Techniques for Program Verification

by Greg Nelson

CSL-81-10 JUNE 1981

© 1980 by Charles Gregory Nelson

Abstract: See page ii.

Some of the material in this paper has appeared previously in JACM and TOPLAS.

CR Categories: 3.69, 4.12, 4.20, 5.21, 5.24, 5.27, 5.30, 5.40, 5.41, 5.7.

Key words and phrases: Program Verification, Mechanical Theorem-proving, Specification Languages, Decision Procedures, Theory of Equality, Theory of Arrays, Theory of Lists, Theory of Linear Inequalities, Congruence Closure, Simplex Algorithm, Automatic Induction, Programming Language Semantics.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Abstract

The main subject of this paper is the detailed description of a mechanical theorem prover for use in program verification, following the approach used successfully in the simplifier/theorem-prover of the Stanford Pascal Verifier: algorithms are developed for theorem-proving in various logical theories, then the algorithms are combined to produce a theorem-prover for the "combination" of the theories. The algorithms described in this paper are refinements of those used in coding the Stanford Verifier.

More precisely, algorithms are described for determining the satisfiability of conjunctions of literals (*i.e.*, atomic formulas or negations thereof) in the following theories: the theory of the real numbers under $+$, $-$, $<$, and $\leq$; the theory of Lisp list structure under car, cdr, cons, atom, and $=$; the theory of uninterpreted function symbols under $=$, and the theory of arrays under operations for accessing and updating elements. A general method for combining such algorithms is described and applied to them to produce a single program that determines the satisfiability of conjunctions of literals containing any of the above functions and predicates.

A second subject of the paper is the problem of combining these "special-purpose" theorem-proving techniques with the two "general-purpose" techniques of matching (or resolution) and induction. Several examples are worked through that suggest that matching in the data structure of the special-purpose algorithms is a viable alternative to matching using ordinary list structure, but the details of an appropriate matcher are not considered. Inductive proofs are considered in more detail: formal semantics are described for a logical system that extends first-order logic with non-deterministic partial recursive function definitions, and an appropriate induction rule is defined and proved to be valid. The rule seems to be suited to the methods of Boyer and Moore; if so, their successful heuristics for proving theorems about total functions can be used with the new rule to prove theorems about partial functions. This is an important extension, since the specifications for programs that manipulate linked data structures are most naturally written using partial (and perhaps non-deterministic) recursive function definitions. An example is given of the use of the system in verifying the correctness of a "destructive" list reversal program.

Finally, this paper addresses the question of the role of program verification in contemporary programming enviornments. It is argued that there are immediate applications for program verification is establishing invariants that are of practical value to a compiler. A programming language is outlined that uses verifiable invariants in place of type declarations, thereby allowing both a higher level of consistency checking than that performed by compilers for conventional hard-typed languages, and also the flexible data structures that are ruled out by strict type systems.

Preface

This paper is a slightly revised version of my Ph.D. thesis. I have rearranged the material to put the tedious parts as near to the end as possible, and included a few new results and discussions (in particular, in sections 4, 8, and 11). I also changed the semantics of the formal system in order to correct errors in the induction rule; thus section 20 is now correct.

Judging from the requests I have received for "a tape of the simplifier", there is a demand in the research community for a good mechanical theorem prover. Instead of preparing a distributable binary program, I have worked on concise descriptions of the algorithms, so that those who want them can implement them easily. Sections 6 through 13 include a cook-book style guide to the parts of the theorem-prover that I understand well and recommend as worth copying.

This work was influenced by the ECL programming project at Harvard, the Pascal Verification project at Stanford, the Analysis of Algorithms group at Stanford, the programming methodology prevalent at the Xerox Palo Alto Research Center, and by the research of Bob Boyer and J Moore. So if the results are not perfectly harmonious, you can see why.

I particularly want to acknowledge the contributions of Derek Oppen, who collaborated with me on some of this research, and of Bob Tarjan, my thesis advisor, who contributed a valuable algorithmic point of view.

I am grateful for the careful criticism of the manuscript offered by Jim Horning, Dick Karp, Don Knuth, Derek Oppen, Wolf Polak, and Bob Tarjan. I have had helpful discussions about this material with Bob Boyer, Steve German, J Moore, Bill Scherlis, and Dan Sleator.

Don Knuth's TEX and METAFONT typesetting and type design systems became available just when I needed them, so that the preparation of this manuscript became a facinating, and in the main enjoyable, lesson in computer typography. I have used several of Knuth's typesetting conventions.

This research was supported by the Fannie and John Hertz Foundation and by the Xerox Corporation.

Contents

1. Introduction

The introduction decribes the goals of this research, defends these goals against criticism that has been made of program verification, summarizes the results of the paper, and gives some examples that illustrate the value of verification in compilation, the kinds of theorems that a verifier must prove mechanically, and the notation that will be used in the rest of the paper.

1. Goals

Programming systems that include mechanical verification as part of the compilation process will make programs more efficient, reliable, and flexible—if they can ever be made practical. The main reason such systems have not yet progressed beyond the experimental stage is that they depend on mechanical theorem provers, which have been too slow and unreliable. The goal of the research described in this paper is the construction of a mechanical theorem prover that is fast and reliable for the class of problems encountered in program verification.

If program verification fulfills its greatest potential, future programmers will write programs that are as reliable as their specifications, and write specifications in so clear a language that they are self-evidently correct. However, research in the last decade has revealed formidable obstacles in the path to this goal, and whether it can ever be reached is a subject of debate. Many people have concluded that verification methods have no immediate practical value in typical software projects, but this conclusion is probably wrong: There is immediate potential for verification methods in establishing invariants that, while not strong enough to show the total correctness of the program, are still of great practical value in compilation.

An *invariant* is a statement that is invariantly true at some point during program execution, for example "whenever the procedure qsort is entered, l will be less than r," or "if the **if** statement at line 1405 succeeds, then $rlink(p) \neq$ nil," or "when the GOSUB returns, i will be between 1 and n." A program verifier's business is to verify the invariants suggested to it by the programmer and compiler, and perhaps to ferret some out by itself in the process.

If appropriate invariants have been established, a compiler can safely dispense with runtime checks on array selections and pointer dereferences, can safely produce code that returns records to the free list instead of waiting for a garbage collector to do so, can simplify polymorphic or otherwise generalized module bodies into the appropriate instance required by the program at hand, and can generally provide greater reliability and flexibility without sacrificing the speed of the compiled program.

It is illuminating to view these compilation-related invariants as generalized type declarations. Conventional type declarations can be viewed as invariants; for example, to declare that p has the type **procedure**(a : **array**[1, 10] **of integer**) is (more or less) to declare that at entry to the body of p, it is invariantly true that a is an array of ten integers. A compiler for a conventional hard-typed language checks that procedure and module interfaces respect the type system; in other words the compiler verifies that the declared invariants really are invariants. The restrictions and complications in existing type systems are introduced to facilitate this verification, and could be removed if more sophisticated verification methods were used. For example, suppose we are writing a procedure p whose argument is an array of integers, $a_1, \ldots, a_n$, and that we do not know n in advance, but will instead assume the existence of a 0th element a_0 that contains the number n.

Using a programming system based on a mechanical theorem-prover, we could label the entry of the procedure with the invariant:

$$\mathbf{array}(a) \wedge \mathbf{lob}(a) = 0 \wedge \mathbf{hib}(a) \geq 0 \wedge \mathbf{hib}(a) = a(0)$$

where **array** is a predicate that is true of arrays, and **lob**(a) and **hib**(a) are the low and high bounds of the array a, respectively. A verifying compiler would refuse to compile a program until it had verified that the entry condition was satisfied at every call to p, and that only valid components of a were accessed in the body of p. For realistic programs, such a verification requires only simple reasoning about inequalities, using automatic techniques that are well-understood and efficient.

Of course, one could add to Pascal or Algol a particular type to handle this problem, though some care must be taken to prevent the program from changing $a(0)$. We might call the type "arbitrarily long but not dynamically variable array of integers." In this way many programming languages have acquired an elaborate collection of types, whose rules of use are intricate and error-prone. PL/I, for example, uses several different types for string variables. The invariants verified by a mechanical theorem prover can provide more precise specifications for procedures and modules, using a type structure that is simple, powerful, and general.

Thus the invariants of a program range from the trivial ("at line 6, the type of x is **integer**") to the very difficult ("at line 32767, the output file is a valid compiled image of the input file"). Existing compilers check the trivial ones; experimental verifiers grapple with the most difficult ones; between these extremes there is a class of invariants, rich by comparison to conventional type systems, that could be managed by a practical verifying compiler. Section 4 contains a sketch of such a type system, and some example programs that use it.

The basic verification problem is to determine whether a program P meets given entry and exit specifications F and G, in the sense that if F is true when P is started, G is invariantly true when P finishes. If P is a large program, it will be composed of smaller pieces, each of which will be specified with its own entry and exit invariants. P will be verified under the assumption that the pieces work correctly, and the pieces will be verified separately. By repeating this process, the general verification problem is reduced to the problem of verifying that some elementary construct of the programming language meets given entry or exit conditions. If the programming language has an appropriate semantic defintion, it will be possible to answer such questions directly (assuming a mechanical theorem prover is available). For example, it might be a semantic rule that $P(x)$ is true at exit from $x \leftarrow t$ if and only if $P(t)$ is true at entry. Such rules for relating predicates before and after execution of some programming language construct are called *predicate transformation semantics* for the construct. Program verification, at least as it is now known, cannot be performed on languages that lack predicate transformation semantics.

Most languages were designed with no consideration for verifiability, and as a consequence it is infeasible to give them predicate transformation semantics. (Their state transformations depend on so many obscure properties of the state that the corresponding predicate transformations are unmanagable.) To obtain the benefits of program verification, it is necessary to plan for them during language design. This is a controversial subject, since many people fear that languages with suitably regular semantics will be unequal to the tasks of real-world programming, or in any case unsuited to the tastes of most programmers. On the opposing side, a number of languages have been proposed, or outlined, with very regular semantics (see for example references [19, 36]) that are (arguably) adequate for general-purpose programming. This paper, while primarily concerned with mechanical theorem-proving, continues this line of research by using a new language for its examples. As a certain amount of discussion of the semantics of the language used for examples is inevitable in any case, outlining a new language is not too much of a digression.

In summary, the main goal of this research is to develop and refine mechanical theorem-proving techniques, and a secondary goal is to outline a programming language and verifying compiler that take advantage of the techniques.

2. Defense of program verification

There are mountainous obstacles to program verification, and many people say flatly, "it will never work." Before summarizing the results of this paper in section 3, let us consider several objections that are frequently raised against program verification, some of them spurious, others serious.

The first objection that we will consider is based on theoretical results. Many logical theories can be shown to be undecidable, or to be decidable only by algorithms with super-exponential time bounds. These results are sometimes construed as obstacles to mechanical theorem-proving, but to do so is to underestimate their scope: such lower bounds apply not only to machine proofs, but to any proofs. For example, [23] proves that there are true statements of length n about geometry whose shortest proofs have length $2^{\Omega(n)}$; thus there is a theorem of geometry that can be squeezed on a page, although any of its proofs would more than fill the universe with fine print. Whether we use pencil and paper or electronic computers, such theorems are secure forever from our attack; but this is irrelevant to the question of whether mechanical theorem provers can find the proofs that mathematicians find. It is also demonstrably difficult to find a winning move in an arbitrary position in the game of checkers (in the precise sense that the problem is P-space complete on an n by n board), but, as anybody who has seen the construction in the proof [25] will agree, it would be mistaken to infer from this that computers cannot play checkers.

The second objection is based on the difficulty of specifying a program. To verify mechanically that a program has some property, the property must be defined in a formal language. Thus the important question arises: can the formal specification of the desired properties of a program be any simpler than the program itself? For example, the APL statement $x \leftarrow \iota\, n$, which sets x to the value $(1, 2, \ldots, n)$, is as simple as any formal specification for it; and at the other extreme, the formal specification of all the properties that are desired of some operating system, compiler or editor might be as formidable as the listing of a program that has the properties. These examples show that it is naïve to talk without qualification about verifying the "correctness" of a program; the best that can be done is to verify that one complicated object (the program) is "consistent," in some complicated technical sense, with another complicated object (the set of specifications). Thus—the objection goes—verification does not lead to reliability, but instead multiplies the possibilities for error.

One answer to this question is that adding redundancy makes it more likely that errors will show up as inconsistencies. A more forceful answer is that the result of a computation can be more clearly specified in a language designed for such specifications than in a language designed for describing computations themselves, because specification languages are not required to be efficiently executable (or even executable at all). Many programming languages sacrifice some efficiency for power, clarity, and ease of expression; but even languages (like APL) that go a long way in this direction do not approach the concise and powerful notation common in mathematical discourse. If all concern for efficiency is abandoned, complicated programs become simpler; if in addition the language they are written in is designed for clarity only, they become simpler still.

It is a surprisingly difficult task to design a specification language, and those of most verifiers are rather cumbersome. Designers of programming languages may think that all their problems would vanish if efficiency w:re no longer required; but it is still a challenge to find a harmonious language that is expressive but not baroque. The expressiveness of the specification language is a critical factor in program verification; perhaps this has not received the emphasis that it deserves.

The third objection is a practical one; it goes something like this: Since everyone knows that errors are omnipresent, the attempt to verify a program will either never succeed, or succeed only erroneously. There is a lot of truth in this objection. There may be errors in the specifications, in the verifier, in the operating system, in the hardware; substitute zero for oh in a few key places and a bank's new verified accounting system will set everyone's balance to zero. Foolish people advertise program verification along these lines: "Think what peace of mind you will have when the verifier finally says '`verified`,' and you can relax in the mathematical certainty that no errors remain."

The answer to the objection is that the purpose of verification is not to produce peace of mind, but to find bugs in programs. A verifier would indeed be useless if failed verifications did not point the way to bugs in the program; but in fact they do. The value of a verifier is in no way contingent on its being bug-free. As it is prudent to assume that there are bugs in everything, the message at the successful exit of program verifiers should be changed from "`Verified`" to "`Sorry, can't find any more errors.`"

In fact, "bug-catcher" is a better name than "verifier" for the kind of system aimed at in this paper. Most compilers provide a crude kind of bug-catching in that they check that the usage of a variable is consistent with its type (and to achieve this they use an unpleasantly rigid type system). Section 4 describes a more expressive type system that can be handled with the aid of a verifier, allowing the system to catch many more errors at compile time than a traditional compiler could ever hope to.

The last objection that we will consider is the main one raised by Richard De Millo, Richard Lipton, and Alan Perlis in their article [18]. They summarize their argument as follows: "The aim of program verification, an attempt to make programming more mathematics-like, is to increase dramatically one's confidence in the correct functioning of a piece of software, and the device that verifiers use to achieve this goal is a long chain of formal, deductive logic. In mathematics, the aim is to increase one's confidence in the correctness of a theorem, and it's true that one of the devices mathematicians *could* in theory use to achieve this goal is a long chain of formal logic. But in fact they don't. ... We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem—and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail."

Briefly, the article argues that program verifications are to programming as "imaginary formal demonstrations" are to mathematics: they have no rôle in practice. The authors quote Poincaré: "... if it requires twenty-seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?" They write "The *Principia Mathematica* was the crowning achievement of the formalists. It was also the deathblow for the formalist view. ... Russell failed, in three enormous, taxing volumes, to get beyond the elementary facts of arithmetic." Formal methods (the authors declare) are important only in theory; in practice mathematicians use informal methods suitable for human social interaction.

It is true that we have more confidence in long-standing theorems than in new results, and that the social process accounts for this. But instead of focusing on a theorem after it is published, or in its formative stages when it may be the subject of excited talk and blackboard discussions, consider the crucial stage during which the theorem is transformed from an intuitive idea, perhaps supported by diagrams or vague scribbles, into the quasi-formal language that is required by scientific journals. This is usually done alone, since it is hard to get anyone to read your error-filled drafts; many mistakes are found in this stage (in fact some results simply evaporate); and it is in this stage that a mathematican, alone with a thick sheaf of definitions and formulas, relies most heavily on formal methods. Since this stage is the one most nearly analogous to the coding of a program, we can expect formal methods to be valuable in coding.

In other words, this objection is based on a questionable view of the rôle of formal methods in mathematics and computer programming. Since the whole thrust of this paper is towards formal methods, we will consider the objection carefully. Let us consider three problems, one from analysis, one from topology, and one from computer programming:

- Determine the area bounded by the x-axis, by the line $x = 1$, and by the parabola $y = x^2$.
- Show that a simple closed curve divides the plane into two regions.
- Show that if two circularly-linked lists of records are "spliced" by exchanging any link in one list with any link in the other list, the result is a circular list containing all records of the original two lists.

The first two problems are much more managable than the third, because of the availability of appropriate formal methods. Thus there are standard formal definitions for the notions of area and of connected region, but it is not obvious how to state the third problem formally. The solutions to the first two problems can also be presented formally: for the first, we have

$$\int_0^1 x^2\,dx = \frac{x^3}{3}\bigg|_0^1 = 1/3;$$

for the second problem, let $\beta(G)$ be the betti number of the group G, f an injection of S^1 into S^2, and $|f|$ the range of f. Then the number of components of the complement of $|f|$ is

$$\beta(H_0(S^2 - |f|)) = 1 + \beta(\tilde{H}_0(S^2 - |f|)) = 1 + \beta(H^1(S^1)) = 1 + \beta(Z) = 2.$$

By contrast, the third problem requires drawing a picture, which is no proof at all, or an inductive proof that runs several pages. Surely this is not because of the inherent difficulty of the problem, but rather because appropriate formal methods have not been developed. It is instructive to remember how difficult the first two problems are to solve from first principles; before the development of symbolic algebra and calculus, Archimedes gave a long geometric construction for the "quadrature of the parabola"; before the development of algebraic topology, the Jordan Theorem was proved by elementary methods, but the proof is notoriously long and tricky.

These examples show that a question like "is the Jordan Theorem within the range of a mechanical theorem-prover" is nonsense. It depends on what formal methods the machine is armed with. Perlis *et al.* assume that a mechanical theorem-prover must start with primitive notions and construct a "foundational" formal proof, in a language like set theory; but this is not true. Such foundational demonstrations are relevant only when formal symbolic methods are themselves the objects of study, as in Hilbert's program or Whitehead and Russell's work. But outside of the foundations of mathematics, formal symbolic methods are used in practical reasoning, as the above examples showed. MACSYMA is an example of a program which does formal manipulation in a practical formal domain.

The case for formal methods is summarized by Hilbert [27], pp. 441–2: "It is an error to believe that rigor in the proof is the enemy of simplicity. On the contrary we find it confirmed by numerous examples that the rigorous method is at the same time the simpler and the more easily comprehended. The very effort for rigor forces us to find out simpler methods for proof. It also frequently leads the way to methods which are more capable of development than the old methods of less rigor. ... I think that wherever, from the side of the theory of knowledge or in geometry, or from the theories of natural or physical sciences, mathematical ideas come up, the problem arises for mathematical science to investigate the principles underlying these ideas and so to establish them upon a simple and complete system of axioms, that the exactness of the new ideas and their applicability to deduction shall be in no respect inferior to those of the old arithmetical concepts."

It is a tough challenge to build a system in which we can reason about programs with the exactness and deductive ease with which we reason about arithmetic. The results in this paper are offered as a step toward this goal.

3. Summary of results

Most of the results in this paper are algorithms for mechanical theorem-proving in the conventional first-order predicate calculus with equality. These results are presented first, since they do not depend on any non-standard logical systems. For reasons described later in this section, the predicate calculus is not expressive enough to describe programs that manipulate linked data structures. Therefore, in section 16 we extend the predicate calculus with recursive function definitions. We defer the semantic definition of the programming language used in the examples until section 17, since its semantics are related to those of recursive functions, and because this masses the formal technical parts of the paper toward the end where they are out of the casual reader's way. Thus, some of the notations used in the first sections of the paper precede their formal definitions; we give informal definitions for them in section 4. The last few sections of the paper discuss the mechanical theorem proving problem in the presence of recursive function definitions.

The algorithms for theorem-proving in the conventional predicate calculus, described in sections 5 to 14, are essentially those used by the Stanford Pascal Verifier [56] to reason about conjunctions of literals. This part of the Verifier's theorem prover was implemented by Derek Oppen and the author. Essentially, the theorem-prover's data stucture is shared by several cooperating "satisfiability procedures." A *satisfiability procedure* for a logical theory is a program that solves the "satisfiability problem" for the theory. The *satisfiability problem* for a theory is the problem of determining the satisfiability of a conjunction of literals (signed atomic formulas) from the theory.

The satisfiability problem is to be distinguished from the *decision problem*, in which quantified statements are allowed. The problem of proving an arbitrary quantifier-free formula F in a theory can be reduced to the satisfiability problem; for example, $A \wedge B \supset C$ is valid if and only if $A \wedge B \wedge \neg C$ is unsatisfiable. If the boolean structure of F is complicated, this reduction may produce exponentially many instances of the satisfiability problem. If we excude such formulas (since no practical way to handle them is known), then the complexity of the satisfiability problem for a theory is a good measure of how difficult it is to reason mechanically in the theory.

The theorem-prover for the Stanford Pascal Verifier contains satisfiability procedures for four logical theories: the theory $\mathcal{R}$ of the real numbers under addition, the theory $\mathcal{E}$ of equality with uninterpreted function symbols, the theory $\mathcal{L}$ of Lisp list structure, and the theory $\mathcal{A}$ of arrays. These theories are defined in section 5; here we briefly describe their satisfiability problems.

The satisfiability problem for $\mathcal{R}$ is equivalent to the linear programming problem. Khachian [29] describes a polynomial-time algorithm for this problem, but the algorithm has not been tested in practice. The satisfiability procedure for $\mathcal{R}$ is based on the simplex algorithm, which takes exponential time in the worst case, but is fast in practice. An example of a conjunction that it will prove inconsistent is

$$y > 2x + 1 \wedge y > 1 - x \wedge y < 0.$$

In general, the simplex algorithm will determine the satisfiability over the reals of any conjunction of linear inequalities or equalities. Although almost all theorem provers used in program verification have special routines for arithmetic formulas, the simplex algorithm has rarely been used, possibly because most implementations of the algorithm are designed to handle large problems arising in operations research, and such implementations are not suitable for mechanical theorem proving.

The implementation of the simplex algorithm is described in section 6. The theorem prover uses heuristics for formulas containing non-linear inequalities and integer-restricted variables, but these will not be described.

The satisfiability programs for the theories $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{A}$ are all based on the "congruence closure" algorithm. Section 6 describes the congruence closure problem and its relation to mechanical theorem proving. Section 7 describes a satisfiability procedure for $\mathcal{E}$ based on a simple $O(n^2)$ congruence closure algorithm. A faster algorithm is described by Downey, Sethi, and Tarjan [20]. Sections 9 and 10 describe the satisfiability procedures for $\mathcal{L}$ and $\mathcal{A}$.

An example of a valid formula in the theory $\mathcal{E}$ of equality with uninterpreted function symbols is $x = y \supset \mathrm{f}(x) = \mathrm{f}(y)$. A more interesting example is

$$\mathrm{g}(\mathrm{g}(\mathrm{g}(x))) = x \wedge \mathrm{g}(\mathrm{g}(\mathrm{g}(\mathrm{g}(\mathrm{g}(x))))) = x \supset \mathrm{g}(x) = x.$$

The theory $\mathcal{L}$ is essentially the theory of ordered pairs under the Lisp functions for constructing them and selecting their components. These functions are defined as follows: $\mathrm{cons}(x, y)$ is the ordered pair (x, y); $\mathrm{car}(z)$ and $\mathrm{cdr}(z)$ are the first and second components respectively of the ordered pair z; and $\mathrm{atom}(z)$ is true if and only if z is not an ordered pair. An example of a theorem in this theory is

$$\neg\,\mathrm{atom}(x) \wedge \neg\,\mathrm{atom}(y) \wedge \mathrm{car}(x) = \mathrm{car}(y) \wedge \mathrm{cdr}(x) = \mathrm{cdr}(y) \supset x = y.$$

The time and space costs of the satisfiability procedures for $\mathcal{E}$ and $\mathcal{L}$ are dominated by the congruence closure algorithm. Thus, using various versions of the algorithm described by Downey, Sethi, and Tarjan [20], these satisfiability problems can be solved in $O(n \log n)$ average time and $O(n)$ space, or $O(n \log^2 n)$ worst-case time and $O(n)$ space, or $O(n \log n)$ worst-case time and $O(n^2)$ space.

Finally, the theorem prover contains a satisfiability procedure for the theory of arrays under operations for updating and selecting elements. Essentially, the literals handled by this satisfiability procedure are equalities and negated equalities between terms containing "array selections" like $v(i)$ as well as "updated arrays" like $v^{(i)}_{e}$, where $v^{(i)}_{e}$ denotes the array each of whose components is equal to the corresponding component of v except for the ith component, which is equal to e. That is, $v^{(i)}_{e}$ is the array to which v is transformed by the assignment $v(i) \leftarrow e$. An example of a statement valid in this theory is

$$a^{(i)}_{e} = b^{(i)}_{e} \wedge a(x) \neq b(x) \supset x = i.$$

The satisfiability problem for $\mathcal{A}$ was first solved by Kaplan [28]. The results of Downey and Sethi [21] show that the problem is NP-hard. Section 10 describes an algorithm for this satisfiability problem that is simpler than Kaplan's. It follows from the results in section 10 that the problem is NP-complete.

The formulas that the theorem prover must prove usually do not fall within any of these naturally-defined theories—they involve "mixed" terms containing functions and predicates from several theories. Satisfiability procedures have been found for some theories with "mixed" terms: [53] gives a satisfiability procedure for Presburger arithmetic with uninterpreted function symbols, and [57] for Presburger arithmetic with arrays and uninterpreted function symbols. Section 5 describes a general method for combining satisfiability procedures for two theories into a single satisfiability procedure for their combination, which contains the functions and predicates of both theories. This method was described previously in Nelson and Oppen [44]. The method is used in the Stanford Pascal Verifier to combine the four satisfiability procedures mentioned above.

The conventional predicate calculus is not expressive enough to reason about programs that manipulate linked data structures. A good solution to this problem is to add partial recursive function definitions to the specification language. This makes it possible to formalize, in a precise but intelligible way, many specifications that are difficult or impossible to formalize in the specification languages of existing automatic verifiers. Section 16 gives gives a formal semantic definition for an extension of the predicate calculus that allows partial recursive function definitions.

This paper makes only a small start on the problem of reasoning mechanically about partial recursive functions. Although most specifications are easy to state, the formal verification that a program meets its specification is likely to be long and complicated. Section 19 describes how to modify the data structure used by the satisfiability procedures in order to accomodate partial recursive functions. Section 20 describes an induction rule for the system. This rule is based on the induction rule described by Boyer and Moore [7], in the hope that their highly successful heuristics for constructing induction arguments in a system based on total recursive functions can be used for the system described in section 16.

Here is an example illustrating the importance of recursive functions in specifying programs that manipulate pointers. Consider formalizing the statement that a is a list linked by the field l that terminates in nil; that is, of formalizing the statement:

$$a = \mathbf{nil} \ \text{ or } \ l(a) = \mathbf{nil} \ \text{ or } \ l(l(a)) = \mathbf{nil} \ \text{ or } \ \cdots. \tag{1}$$

(Ignore whatever specifications are required about the types of the nodes or the other fields in the nodes.) No formula H of the first-order predicate calculus with function symbols and equality is equivalent to (1). For if H entails (1), the infinite conjunction

$$H \wedge a \neq \mathbf{nil} \wedge l(a) \neq \mathbf{nil} \wedge l(l(a)) \neq \mathbf{nil} \wedge \cdots \tag{2}$$

is unsatisfiable. By the compactness of first-order logic, some finite subconjunction of (2) is unsatisfiable; so H entails some finite sub-disjunction of (1), which means that H is much too strong.

Using a time-honored form of definition, the valid lists could singled out as follows:

(a) nil is a valid list.
(b) if $l(x)$ is a valid list, then x is a valid list.
(c) nothing is a valid list unless it is because of (a) and (b). (3)

(There is a tendency to write the converse of (b), erroneously associating l with the successor function on the natural numbers. Instead l is analogous to the predecessor function, although of course l need not be one-to-one.) Points (a) and (b) are easily stated in the predicate calculus, but some extension is needed to formalize the "extremal clause" (c). The specification language extends the predicate calculus by allowing recursive function definitions. To formalize (1), we make the recursive definition

$$\textbf{Function validlist}(x) \equiv [\![\, x = \mathbf{nil} \Rightarrow \mathbf{true};\ \mathbf{validlist}(l(x)) \,]\!]. \tag{4}$$

Then we express (1) by stating that the computation specified by the term validlist(a) terminates with the result true. (The notation on the right means, "if $x =$ nil then true else validlist($l(x)$)".) Obviously if x is a valid list because of (a) and (b), then the computation of validlist(x) terminates with true, since the two branches in the function definition correspond precisely to (a) and (b). Furthermore if validlist(x) is evaluated in any standard way, it will loop unless x is a valid list because of (a) and (b). Thus recursive definitions can be used to formalize definitions containing extremal clauses. A particular evaluation rule is defined in section 16.

The objection may be raised that recursive definitions are essentially programs, and if they are allowed in the specification language, then the specifications may become as hard to understand as the programs themselves. But the whole point of a specification language is to introduce powerful forms of expression. There is no protection from badly-written specifications; any language expressive enough to describe specifications for interesting programs contains formulas that are hard to understand. The more powerful the specification language is, the more likely it will be that a program can be specified simply. To formalize the predicate "n is equal to the factorial of m" in the first-order theory of the natural numbers requires a hideous formula; its recursive definition is simple and natural.

4. Notation and examples

This section consists of several examples whose purposes are to introduce some notation and to illustrate the potential uses of mechanical verification in compilation.

The first example is a program that implements the union and find operations for maintaining an equivalence relation on a set of n items, using a simple algorithm, which requires a single array $a(1), \ldots, a(n)$ whose values are integers in the range $[0, n]$. The invariant maintained by the algorithm is that for any i in $[1, n]$, the sequence i, $a(i)$, $a(a(i))$, ... contains 0, and that the call find(i) returns the last value in this sequence before 0, and finally that two items i and j are equivalent if and only if find(i) = find(j). Under these assumptions, the identity relation can be represented by making $a(i) = 0$ for all i. Furthermore, the equivalence classes represented by find(i) and find(j) can be merged by assigning $a(\text{find}(i)) \leftarrow \text{find}(j)$. Here is a program that implements these operations, together with some specifications for the program:

```
Declare $(\forall i)\ 1 \le i \le n \supset 0 \le a(i) \le n$.

Procedure find($i$) $\equiv$ $[\![ a(i) = 0 \Rightarrow i;\ \text{find}(a(i)) ]\!]$.

Procedure union($i, j$) $\equiv$
  $[\![ i \leftarrow \text{find}(i);$
    $j \leftarrow \text{find}(j);$
    $i = j \Rightarrow [\![\ ]\!];$
    $a(i) \leftarrow j ]\!]$.

Entering find, $1 \le i \le n$.
Exiting find, $1 \le \mathcal{V} \le n$.
Entering union, $1 \le i \le n \wedge 1 \le j \le n$.
```

The code above contains two procedure definitions and three kinds of specifications. We explain the meaning of each construct in turn.

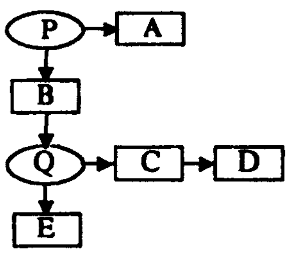
A procedure definition associates an executable statement and a list of formal parameters with a function symbol, in the usual way. Thus the first procedure definition above associates with the function symbol **find** the list containing the single formal parameter i, and the executable statement $[\![a(i) = 0 \Rightarrow i;\ \text{find}(a(i))]\!]$. Once a procedure is defined it may be called in the usual way.

There are five kinds of statements, each of which returns some value (as well as possibly producing side effects): A variable is considered a statement, whose execution returns the value of the variable. An expression $v \leftarrow S$, where v is a variable and S is a statement, is a statement; its effect is to bind v to the value produced by executing S. An expression like $a(i) \leftarrow x$ is allowed as an abbreviation for the assignment statement $a \leftarrow a_i^{(x)}$; see section 3 for the definition of this notation. An expression of the form **with** $v_1 = s_1, \ldots, v_n = s_n$ **do** S, where the v_i are variables and S and the s_i are statements, is a statement; its effect is to execute $s_1, \ldots, s_n$ in turn, producing values $u_1, \ldots, u_n$, then to bind each v_i to u_i, then to execute S, producing a value u, then to restore the original bindings of $v_1 \ldots v_n$, and finally to return u.

An expression $f(s_1, \ldots, s_n)$, where f is a function symbol and the s_i are statements, is a statement. If f is a "built-in function", execution of the statement proceeds with the execution of s_1 through s_n in order, producing values u_1 through u_n; then the built-in function is applied to these values and the result is returned. Otherwise f will have been defined as a procedure with formal parameters $v_1, \ldots, v_n$ and body S; in this case the effect of the statement is identical to that of **with** $v_1 = s_1, \ldots, v_n = s_n$ **do** S. (This implies that all parameters are passed by value.)

The fifth kind of statement is a "conditional statement" of the form $\llbracket C_1; \ldots C_n \rrbracket$, where each C_i is a *command*; a command is either a statement or an expression of the form $P \Rightarrow S$, where P and S are statements. To execute a conditional statement, the commands are executed in order, except that a command of the form $P \Rightarrow S$ is an "exit conditional": if P is true when the exit conditional is reached, then S is executed and the rest of the conditional statement is skipped; if P is not true, then S is ignored and execution continues with the next command. Of course, P is executed to determine its truth value. Here is an example of a conditional statement and equivalent flowchart:

```
⟦P ⇒ A;
 B;
 Q ⇒ ⟦C;D⟧;
 E⟧;
```



As special cases we have the conventional conditional expression

$$\llbracket P_1 \Rightarrow E_1;\ P_2 \Rightarrow E_2;\ \ldots\ P_n \Rightarrow E_n;\ E_{n+1} \rrbracket,$$

with the meaning "if P_1 then E_1 else if P_2 then E_2 else ... if P_n then E_n else E_{n+1}," and the ordinary block $\llbracket S_1; \ldots; S_n \rrbracket$ where no S_i contains $\Rightarrow$, which is equivalent to the Algol begin $S_1; \ldots; S_n$ end. This notation is used in the ECL programming system [61].

There are no iteration primitives; all loops must be created by recursion. Note that the definitions above imply that the scoping of variables is dynamic, as in Lisp. This concludes the description of the imperative features of the language.

The declarative features of the language differ from conventional declarations just as one would expect in a language designed to be compiled with the aid of a mechanical theorem-prover: declarations are expressed in a formal logical language, and are not confined to listing the types of variables, but may declare any invariant that is expressible in the logical language. It is not necessary in this informal introduction to discuss the semantics of the logical language; as declarative notation is more standardized than imperative notation, we can use functions and predicates on numbers, boolean connectives and quantifiers without ambiguity. In section 16 the logic is defined precisely.

To avoid confusing our declarations with those of conventional programming languages, we will call them *specifications.* A specification may be associated with the point of entry or point of exit of any procedure; the syntax for such specifications is shown in the union-find program above. The specification

$$\textbf{Entering find, } 1 \leq i \leq n$$

declares that whenever the body of **find** is entered (note that this is after the formal parameter is bound), i is between 1 and n. Exit specifications are similar, but when describing the exit from a procedure it is desirable to be able to refer to the value returned by the procedure, so we make the convention that in an exit specification, the symbol $\mathcal{V}$ refers to the returned value. Thus the exit specification of **find** declares that the value returned by the procedure is between 1 and n. The third and final way of declaring a specification is with the keyword "Declare", as at the top of the example; this construct declares that the specification is true at every entry and every exit to or from any procedure.

We will use the word *module* to denote a set of procedure definitions and specifications; thus the **union-find** example above is the listing of a module. We say that an entry (or exit) from a procedure is *valid* if the context at entry (or exit) satisfies all specifications associated with the entry (or exit). A module is *consistent* if any valid entry to any procedure of the module leads to a computation in which there are no invalid entries or exits. (We do not require that a valid entry lead to a terminating computation.)

The **union-find** module is consistent. Consider, for example, a valid entry to **find**, and suppose that $a(i) \neq 0$, so that the recursive call is reached. Then the recursive entry will be valid, since we know that $0 \leq a(i) \leq n$ and that $a(i) \neq 0$ from which it follows that $1 \leq a(i) \leq n$. Similarly, one may verify every other "path" in the program from entry to entry or from entry to exit (the path we have just checked is in fact the most difficult); it follows by induction that the module is consistent.

There are close relations between the definition of the logical semantics of the specification language, the definition of the predicate transformation semantics of the programming language, and the definition of consistency; these definitions are all givin formally in sections 16 and 17.

If our only interest in a program were to verify it, then we could end our discussion of semantics with the definition of consistency. But in fact our main interest in a program is our ability to run it, and the predicate transformation semantics do not by themselves say what happens when a program is run, since the predicates are uninterpreted. To define a working programming system, we need to choose a class of finitary values that will be represented in the run-time environment of a program, and to agree that certain predicate symbols denote certain predicates on this class. This choice, which we may loosely call the choice of a type structure, has as much effect on the character of the language as the choice of its predicate-transformation semantics; for example languages like Lisp, APL or SETL could be defined by choosing the class of symbolic expressions, of sets, or of matrices. (It is important to note that the way we have defined things, the predicate-transformation associated with a statement of our language does not depend on the "types" of the values in the statement; this factoring helps keep the semantics manageable.)

It is beyond the scope of this paper to give a precise definition for any type structure, but we will briefly outline one possible structure, in order to give a few examples of the kind of program that one would write for a verifying compiler.

We postulate the existence of a compiler conversant with integers, floating-point numbers, characters, boolean values, arrays, and record-pointers, and we reserve the predicate symbols **int**, **char**, **real**, **bool**, **array**, **recptr** as names for the appropriate predicates. If a is an array, $\mathbf{lob}(a)$

and hib(a) are the lower and upper bounds, respectively, for a's index set, which we assume to be an interval of integers. If p is a record-pointer, we write field(f, p) to denote that f is a valid field selector for the record pointed to by p. (Field selectors are values, not constants; perhaps the compiler represents them as byte offsets. No predicate symbol is reserved for recognizing field-selectors, because no use for one is immediately apparent.) By avoiding the two types *pointer* and *record*, and combining them into one type *record-pointer*, we lose very little flexibility in practice, and gain simplicity in notation (cf. Lisp and Algol W).

Conventional type declarations are effected by using these predicates in entry specifications; for example the type declaration in the Algol procedure "p(x : int)" can be mimicked by the declaration "Entering p, int(x)." The rich specification language allows interfaces to be described more precisely; for example,

$$\text{Entering p, } \mathbf{array}(a) \wedge \mathbf{lob}(a) = 0 \wedge \mathbf{hib}(a) \geq 0 \wedge \mathbf{hib}(a) = a(0)$$

declares that when entering p, a is an array whose high bound is stored in its first element, which is indexed by 0.

To describe linked record structures precisely is a challenge to the specification language (see sections 15 and 16), but it is not difficult to describe the number and general nature of the fields and links after the manner of a conventional type system. For example, consider specifying the interface to a procedure that inserts a record r into an ordered list a, $l(a)$, $l(l(a))$, ..., of records, where the keys of the records are integers contained in the field k. We want to use the procedure on different types of records at different times, but we assume that k and l are global constants and that any time the procedure is entered, a is the first record of an appropriate list. (We do not assume that all the records in the list headed by a have the same structure; only that they each have integers stored in the field whose offset is k, and that they each have links to appropriate records stored in the field whose offset is l.) If we can define in the specification language a predicate node that is the weakest predicate that satisfies the axiom:

$$\begin{aligned}&(\forall x)\, \mathbf{node}(x) \wedge x \neq \mathbf{nil} \supset \\ &\qquad \mathbf{field}(k, x) \wedge \mathbf{int}(k(x)) \wedge \mathbf{field}(l, x) \wedge \mathbf{node}(l(x)).\end{aligned}$$

(I.e., "if x is any node, then either it is nil, or it is a pointer to a record for which k and l are valid field selectors, and $k(x)$ is an integer and $l(x)$ is a node) then we can formulate the entry specification of our procedure as

$$\text{Entering insert, } \mathbf{node}(a) \wedge \mathbf{node}(r).$$

(Instead of requiring that r be a node, we probably would just require that it have an integer k field and any recordpointer for its l field, since we do not care what $l(r)$ points to.)

One way to "define" the predicate node is to simply assume that it satisfies the axiom above. This is what we will do for now, although the specification language is capable of more precise definitions.

We introduce the abbreviation

$$\text{Field } f : P \rightarrow Q$$

for

$$\text{Declare } (\forall x)\, P(x) \wedge x \neq \mathbf{nil} \supset \mathbf{field}(f, x) \wedge Q(f(x)).$$

Thus the axiomatic definition of node above could be written:

$$\text{Field } k : \mathbf{node} \to \mathbf{int}.$$
$$\text{Field } l : \mathbf{node} \to \mathbf{node}.$$

We also introduce the abbreviation

$$\text{Array } A : [l, h] \to Q.$$

for

$$\text{Declare } \mathbf{array}(A) \wedge \mathbf{lob}(A) = l \wedge \mathbf{hib}(A) = h \wedge (\forall x)\, Q(A(x)).$$

(The compiler will not allow a program to access non-existent array elements, so it does no harm to assume that Q holds for these non-existent elements.)

In both of the above abbreviations, we allow Q to be an expression for the form $[a, b]$, in which case $Q(x)$ should be understood to mean $\mathbf{int}(x) \wedge a \leq x \leq b$.

Every value x in the universe has a *size* $\mathbf{size}(x)$; this might be the number of bits or bytes or words required to represent the value; its exact meaning concerns no one but the implementor of the compiler. We assume, though, that all characters have the same size, as do all floating-point numbers, all (that is, both) booleans, and all record-pointers. (The size of any record-pointer is $\mathbf{size}(\mathbf{nil})$.) Different integers may have different sizes, but we assume that if n is positive, and $|m| \leq n$, then $\mathbf{size}(m) \leq \mathbf{size}(n)$.

Every array a has a *component size* $\mathbf{csize}(a)$, and if p is a record-pointer and f is a field of p, then this field has a size $\mathbf{fsize}(f, p)$. We assume that the size of an array is the product of its component size by the number of its elements.

We use the notation $[f_1 : v_1, \ldots, f_n : v_n]$ to denote a record pointer to a record with fields $f_1, \ldots, f_n$, such that the value in the ith field is v_i, and the size of the ith field is $\mathbf{size}(v_i)$. This notation is allowed to appear in programs. Subsequently, the compiler will not allow the program to store a value into a field of the record unless it fits there, so a programmer who intends to store objects of varying sizes in a field of a record must create the record with a sufficiently bulky value in the appropriate field.

We use the notation $[a, b] : v$ to denote an array whose lower and upper bounds are a and b respectively, whose component size is $\mathbf{size}(v)$, and all of whose values are v; the notation is allowed in programs; and the compiler will not store a value in an array if the value's size exceeds the component size of the array.

To illustrate these conventions, here is a module which defines a predicate **string** together with a function **concat** that operates on any two objects that satisfy the **string** predicate. An object is a string if it is a pointer to a record that includes the two fields *strlen* and *strdata*, the first of which is an integer, the second of which is an array of characters whose length is determined by the *strlen* field:

```
Field $strlen : \mathbf{string} \to [0, 32767]$.
Field $strdata : \mathbf{string} \to \mathbf{array}$.
Declare $(\forall x)\, \mathbf{string}(x) \supset$ Array $strdata(x) : [1, strlen(x)] \to \mathbf{char}$.

Procedure $\mathbf{concat}(x, y) \equiv$
 with $xylen = strlen(x) + strlen(y), xy = \mathbf{nil}$ do
  $[\![ xy \leftarrow [strlen : 32767, strdata : [1, xylen] : \text{“ ”}]$;
    $xylen > 32767 \Rightarrow \mathbf{error}()$;
    $strlen(xy) \leftarrow xylen$;
    $\mathbf{fillin}(0, x, 1)$;
    $\mathbf{fillin}(strlen(x), y, 1)$;
    $xy \,]\!]$.
```

```
Procedure **fillin**($offset, source, i$) $\equiv$
  $[\![ i > strlen(source) \Rightarrow [\![\,]\!];$
    $strdata(xy)(offset + i) \leftarrow strdata(source)(i);$
    $fillin(offset, source, i + 1) ]\!].$
```

Entering **concat**, **string**(x) $\wedge$ **string**(y).

Entering **fillin**, $1 \leq i \leq strlen(source) + 1 \;\wedge$
$strlen(xy) - offset \geq strlen(source)$.

The concatenation procedure simply allocates an appropriate record and fills it in. We assume that " " denotes the character code for a space. Notice that there is wasted work in initializing the *strdata* field to contain blanks; there are several ways of avoiding this, including using a kind of reverse flow analysis to determine that the blanks are going to be overwritten. We want the *strdata* offset to be constant for all strings, so we make all *strlen* fields the same size; here we choose **size**(32767), which on a byte-oriented machine will be two bytes. The error check is necessary, since otherwise the compiler would not be able to verify that *xylen* will fit in the field of the record, and would refuse to compile the program.

2. Satisfiability Procedures

Sections 5 through 14 discuss theorem-proving by satisfiability procedures or "special-purpose provers". First a method of combining satisfiability procedures is described, then satisfiability procedures for $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{A}$ are described, all three based on a "congruence closure algorithm", and then a satisfiability procedure for $\mathcal{R}$ is described based on the simplex algorithm. The satisfiability procedures for $\mathcal{L}$ and $\mathcal{A}$ are derived methodically, following a procedure similar to the Bendix-Knuth algorithm. Finally, in section 14, the matching problem is discussed briefly.

5. Combining satisfiability procedures by equality-sharing

Let R be the set $\{+, -, <, \leq, =, \neq, 0, 1\}$ of the common arithmetic constants, functions, and predicates, excluding multiplication; and let E be the set $\{=, \neq, \mathbf{f}, \mathbf{g}, \ldots\}$, containing many "uninterpreted functions" taking various numbers of arguments. Consider the problem of determining the satisfiability over the reals of a conjunction of literals (that is, signed atomic formulas) containing only variables and the symbols in $R \cup E$. This problem is important in mechanical verification, where the formulas that arise often contain both arithmetic and non-arithmetic functions. A typical instance of the problem is to determine the satisfiability of

$$\mathbf{f}(\mathbf{f}(x) - \mathbf{f}(y)) \neq \mathbf{f}(z) \wedge x \leq y \wedge y + z \leq x \wedge z \geq 0. \tag{1}$$

This conjunction is unsatisfiable, since the three inequalities imply that $x = y$ and $z = 0$, hence the disequality is equivalent to $\mathbf{f}(0) \neq \mathbf{f}(0)$.

In general, the *satisfiability problem* for a set of functions, predicates and constants is the problem of determining the satisfiability of a conjunction of literals containing (besides variables) only functions, predicates and constants in the set. (The satisfiability problem is to be distinguished from the *decision problem*, in which quantified statements are allowed.) The satisfiability problem for $R \cup E$ is unusual because it allows "mixed" terms containing functions and predicates from two "natural" theories (the theory of the real numbers and the theory of equality with uninterpreted function symbols) that intuitively have nothing to do with one another. The separate satisfiability problems for R and E were solved long ago—the first by Fourier, the second by Ackermann. But the combined problem was not considered until recently, when it became important in program verification. A solution to the satisfiability problem for $R \cup E$ was first given by Shostak [53]; another solution for a larger class of functions has been given by Suzuki and Jefferson [57].

This section describes a general method for "factoring" the satisfiability problem for $S \cup T$ into the two satisfiability problems for S and T, given certain conditions on S and T. The method gives practical solutions to many satisfiability problems that are important in practice, including that for $R \cup E$. This material represents joint work with Derek Oppen, and was described previously in [44].

Strictly speaking, the satisfiability problem is not defined for a set of functions and predicates, but for the *theory* of these functions and predicates under some axiomatization. For the purposes of this section, an informal definition of a theory is sufficient. The formal definition of a theory is in section 16; the formal proof of correctness of the algorithm described in this section is delayed till section 18, though this section contains an informal proof.

Informally, a theory is determined by a set of function symbols, which will be called the *free functions* of the theory, together with a set of axioms constraining the interpretation of the function symbols. The axioms are written in the first-order predicate calculus with function symbols and equality; thus any theory "gets equality for free." We regard a constant as a special case of a function symbol. We now describe the theories $\mathcal{L}$, $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{E}$.

Let $\mathcal{L}$ be the theory of Lisp list structure with the three axioms

$$(\forall\, x\, y\, u\, v)\ \mathbf{car}(\mathbf{cons}(x,y)) = x \ \wedge\ \mathbf{cdr}(\mathbf{cons}(x,y)) = y$$
$$(\forall\, z)\ \mathbf{atom}(z)\ \Leftrightarrow\ (\forall\, x\, y)\ z \neq \mathbf{cons}(x,y)$$
$$\mathbf{atom}(\mathbf{nil})$$

These axioms imply that $z = \mathbf{cons}(\mathbf{car}(z), \mathbf{cdr}(z))$ whenever z is non-atomic, since in this case $z = \mathbf{cons}(u,v)$ for some u and v (by the second axiom), and it follows from the first axiom that $u = \mathbf{car}(z)$ and $v = \mathbf{cdr}(z)$. Notice that these axioms do not imply $\mathbf{cdr}(x) \neq x$; the axiomatization allows "circular" structure. The free functions of $\mathcal{L}$ are **car**, **cdr**, **cons**, **atom**, and **nil**.

We must introduce some notational conventions for the theory of arrays, since we want to use notation like $a(x)$, where a is a variable instead of a function symbol, and like $a^{(e)}_{i}$, but neither of these are legal in the usual syntax of first-order logic. Thus we adopt the following convention: If t and u are terms, the expression $t(u)$ is an abbreviation for the term $\mathbf{select}(t,u)$. This causes no ambiguity, since the term t cannot be confused with a function symbol. We also adopt the convention that an expression of the form $a^{(i)}_{e}$, where a, e, and i are terms, is an abbreviation for the term $\mathbf{store}(a,i,e)$.

With these conventions, let $\mathcal{A}$ be the theory with the free functions **store** and **select** and the two axioms:

$$(\forall\, x\, a\, e)\ a^{(x)}_{e}(x) = e$$
$$(\forall\, x\, y\, e\, a)\ x \neq y\ \supset\ a^{(x)}_{e}(y) = a(y)$$

These axioms express the obvious properties of **store** and **select**.

Let $\mathcal{R}$ be the theory of the real numbers under addition, whose free functions are $+$, $-$, $<$, and the numerals. We also define the theory $\mathcal{Z}$ of the reals under addition with the additional predicate int recognizing integers, and $\mathcal{R}^{\times}$ and $\mathcal{Z}^{\times}$ that are like $\mathcal{R}$ and $\mathcal{Z}$ respectively except that they include multiplication. For axiomatizations of these theories, see [26] or [59]. A satisfiability procedure for $\mathcal{R}$ is given in section 12. Unfortunately no satisfactory satisfiability procedures are known for $\mathcal{Z}$, $\mathcal{R}^{\times}$, or $\mathcal{Z}^{\times}$, though $\mathcal{Z}$ can be handled reasonably well using heuristics.

It is convenient to define the theory $\mathcal{E}$ of equality with uninterpreted function symbols, whose theorems, like $x = y\ \supset\ \mathbf{f}(x) = \mathbf{f}(y)$, are valid because of the properties of equality. It is not necessary for $\mathcal{E}$ to have any axioms, since the semantics of equality are built into the system. We want all uninterpreted function symbols to be free functions of $\mathcal{E}$. In the context of this paper an "uninterpreted" function symbol is one that is not a free function of $\mathcal{Z}^{\times}$, $\mathcal{L}$, or $\mathcal{A}$, so we define $\mathcal{E}$ to be the theory with no axioms and whose free functions are all function symbols except $+$, $-$, $\times$, $<$, $/$, the numerals, int, **cons**, **car**, **cdr**, **atom**, **nil**, **store**, or **select**.

If $\mathcal{S}$ is a theory, then a term, literal, or formula will be called an $\mathcal{S}$-term, $\mathcal{S}$-literal or $\mathcal{S}$-formula, respectively, if all function symbols appearing in it are free functions of $\mathcal{S}$. For example, $x = y$ and $x \leq y + 1$ are $\mathcal{R}$-literals but $x \leq \mathbf{car}(y)$ is not.

The *satisfiability problem* for a theory $\mathcal{S}$ is the problem of determining the satisfiability in $\mathcal{S}$ of conjunctions of $\mathcal{S}$-literals. This makes the earlier notion of a satisfiability problem precise. The general quantifier-free decision problem for $\mathcal{S}$ can be reduced to the satisfiability problem, since a formula is satisfiable if and only if one of the disjuncts of its disjunctive normal form is satisfiable. A *satisfiability procedure* for $\mathcal{S}$ is an algorithm that solves the satisfiability problem for $\mathcal{S}$.

If $\mathcal{S}$ and $\mathcal{T}$ are two theories with no common free functions, their *combination* is the theory with all the axioms and recursive function definitions of $\mathcal{S}$ and of $\mathcal{T}$. Given satisfiability procedures for $\mathcal{S}$ and $\mathcal{T}$, this section describes a method for constructing a satisfiability procedure for the combination of $\mathcal{S}$ and $\mathcal{T}$.

The method works only for theories with no common free functions. For example, it cannot be used to combine satisfiability procedures for $\mathcal{R}^{\times}$ and $\mathcal{Z}$ into a satisfiability procedure for $\mathcal{Z}^{\times}$. (Which is as it should be, since the satisfiability problems for $\mathcal{R}^{\times}$ and $\mathcal{Z}$ are recursive, but that for $\mathcal{Z}^{\times}$ is not.)

We begin with an example illustrating how the method determines that the conjunction (1) is unsatisfiable. Assume that we have satisfiability procedures for $\mathcal{R}$ and $\mathcal{E}$. We will use the same names "$\mathcal{R}$" and "$\mathcal{E}$" for the satisfiability procedures that we use for the theories themselves.

The first step is to construct two conjunctions $F_{\mathcal{E}}$ and $F_{\mathcal{R}}$ such that $F_{\mathcal{E}}$ is a conjunction of $\mathcal{E}$-literals, $F_{\mathcal{R}}$ is a conjunction of $\mathcal{R}$-literals, and $F_{\mathcal{E}} \wedge F_{\mathcal{R}}$ is satisfiable if and only if (1) is. This is achieved by introducing new variables to represent terms of the wrong "type", and adding equalities defining these new variables. Thus, the first literal is made into an $\mathcal{E}$-literal by replacing the term $\mathfrak{f}(x) - \mathfrak{f}(y)$ with the new symbol g_1, and the equality $g_1 = \mathfrak{f}(x) - \mathfrak{f}(y)$ is added to "define" g_1. The new equality is then made into an $\mathcal{R}$-literal by replacing the $\mathcal{E}$-terms $\mathfrak{f}(x)$ and $\mathfrak{f}(y)$ by the variables g_2 and g_3, which are defined by $\mathcal{E}$-literals. The result is:

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

These two conjunctions are given to $\mathcal{R}$ and $\mathcal{E}$. Each of the conjunctions is satisfiable by itself, so there must be interaction between $\mathcal{R}$ and $\mathcal{E}$ to detect the unsatisfiability. The interaction takes a particular, restricted form: each satisfiability procedure is required to deduce and propagate to the other satisfiability procedure all equalities between variables entailed by the conjunction it is considering. For example, if the conjunction $F_{\mathcal{R}}$ contained the literals $a \leq b$ and $b \leq a$, $\mathcal{R}$ would be required to deduce and propagate the consequence $a = b$.

In the example, $F_{\mathcal{E}}$ does not entail any equalities between variables, but $F_{\mathcal{R}}$ entails $x = y$. (It also entails $z = 0$, but 0 is not a variable, so this equality is not propagated.) $\mathcal{R}$ therefore propagates $x = y$. Given this equality, $\mathcal{E}$ deduces and propagates that $g_2 = g_3$. This enables $\mathcal{R}$ to deduce and propagate $z = g_1$, which enables $\mathcal{E}$ to detect the contradiction.

The method illustrated by this example works for any conjunctions $F_{\mathcal{R}}$ and $F_{\mathcal{E}}$. Obviously, if one of the conjunctions $F_{\mathcal{R}}$ or $F_{\mathcal{E}}$ becomes unsatisfiable as a result of equality propagation, the original conjunction must be unsatisfiable. It is a consequence of the results below that the converse holds as well: if the original conjunction $F_{\mathcal{R}} \wedge F_{\mathcal{E}}$ is unsatisfiable, then one of the conjunctions $F_{\mathcal{R}}$ and $F_{\mathcal{E}}$ will become unsatisfiable as a result of propagations of equalities between variables.

The method will not work as described to combine $\mathcal{E}$ with a satisfiability procedure $\mathcal{Z}$ for the theory of the reals and integers. Suppose, for example, that the conjunctions are

$F_{\mathcal{Z}}$	$F_{\mathcal{E}}$
$\mathrm{int}(x)$	$\mathfrak{f}(x) \neq \mathfrak{f}(a)$
$1 \leq x$	$\mathfrak{f}(x) \neq \mathfrak{f}(b)$
$x \leq 2$	
$a = 1$	
$b = 2$	

Then $F_{\mathcal{Z}} \wedge F_{\mathcal{E}}$ is unsatisfiable, but neither $F_{\mathcal{Z}}$ nor $F_{\mathcal{E}}$ entail any equalities between variables. The problem is that $F_{\mathcal{Z}}$ entails the *disjunction* $a = x \vee b = x$, and each case is unsatisfiable. To handle this problem, the procedure must do case-splitting in certain circumstances.

A formula F is *non-convex* in a theory $\mathcal{T}$ if there exist $2n$ variables, $x_1, y_1, \ldots, x_n, y_n$, $n > 1$, such that in $\mathcal{T}$, F implies the disjunction

$$\bigvee_{1 \leq i \leq n} (x_i = y_i), \tag{2}$$

but for no proper subset S of $\{1, 2, \ldots, n\}$ does F imply the disjunction

$$\bigvee_{i \in S} (x_i = y_i).$$

In this case, (2) is a *split entailed by* F. If no such disjunction exists, F is *convex*. Note that any unsatisfiable formula is convex.

That is, a formula is non-convex if it entails a "proper" disjunction of equalities between variables. When one of the conjunctions in the procedure above becomes non-convex, it is necessary to do a case-split; that is, to "guess" which of the equalities is true, saving the current state so that if the guess is wrong then the state can be restored and another guess tried. This is a straightforward non-deterministic procedure that can be implemented with an extra stack for backtracking. For example, suppose that we are trying to determine the satisfiability of the conjunction of the literals $\text{int}(x)$, $0 \leq x$, $x \leq 1$, $a = 0$, $b = 1$, $f(x) = a$, $f(a) = b$, and $f(b) = b$. They are divided into two conjunctions:

$F_{\mathcal{Z}}$	$F_{\mathcal{E}}$
$\text{int}(x)$	$f(x) = a$
$0 \leq x$	$f(a) = b$
$x \leq 1$	$f(b) = b$
$a = 0$	
$b = 1$	

There are no equalities to propagate, but $F_{\mathcal{Z}}$ is non-convex: it implies the split $x = a \vee x = b$. Suppose that the case $x = a$ is considered first. The procedure puts the triple $(5, 3, x, b)$ on a separate stack, so that it can later return to the state in which there are 5 literals in $\mathcal{Z}$'s conjunction, 3 literals in $\mathcal{E}$'s conjunction, and in this state consider the case $x = b$. It considers the case $x = a$ by adding this equality to both conjunctions. Given $x = a$, $\mathcal{E}$ deduces and propagates $a = b$, and $\mathcal{R}$ detects the inconsistency. At this point the conjunctions are

$F_{\mathcal{Z}}$	$F_{\mathcal{E}}$
$\text{int}(x)$	$f(x) = a$
$0 \leq x$	$f(a) = b$
$x \leq 1$	$f(b) = b$
$a = 0$	$x = a$
$b = 1$	
$x = a$	
$a = b$	

Since the first guess resulted in an unsatisfiable conjunction, the entry $(5, 3, x, b)$ is popped from the stack, conjuncts are removed from $F_{\mathcal{Z}}$ and $F_{\mathcal{E}}$ in a last-in first-out manner until the conjunctions have 5 and 3 literals respectively, and then the $x = b$ case is tried. That is, the following state is created:

$F_{\mathcal{Z}}$	$F_{\mathcal{E}}$
$\mathrm{int}(x)$	$f(x) = a$
$0 \le x$	$f(a) = b$
$x \le 1$	$f(b) = b$
$a = 0$	$x = b$
$b = 1$	
$x = b$	

But $\mathcal{E}$ again propagates $a = b$, so this case is also found to be unsatisfiable. Since there are no more cases to consider, the original conjunction has been shown unsatisfiable.

The following algorithm specifies this process precisely. The algorithm uses two stacks to hold the literals and one stack to hold the untried cases. The *size* of a stack is the number of entries in it.

Algorithm E (*Equality Sharing Procedure*). Given two stacks S_1 and S_2 such that the entries in S_1 are $\mathcal{S}$-literals, the entries in S_2 are $\mathcal{T}$-literals, where $\mathcal{S}$ and $\mathcal{T}$ are theories with no common free functions, this algorithm determines the satisfiability in the combination of $\mathcal{S}$ and $\mathcal{T}$ of the conjunction of all literals in S_1 and S_2. The correctness of the algorithm's answer depends on the assumptions that neither $\mathcal{S}$ nor $\mathcal{T}$ have infinitely many free variables, and that $\mathcal{S}$ and $\mathcal{T}$ are both "stably infinite"; see the correctness proof in section 18 for definitions of these conditions. The algorithm uses another stack T whose entries are lists of the form $(n_1, n_2, u_1, v_1, \dots, u_k, v_k)$, where n_1 and n_2 are integers, $k > 0$, and the u_i and v_i are variables.

E1. [Initialize T.] Make T the empty stack.

E2. [Unsatisfiable?] For $i = 1, 2$, if the conjunction of the literals in S_i is unsatisfiable, then go to step E6.

E3. [Propagate equalities.] For $i = 1, 2$, if the conjunction of literals in S_i entails some equality $u = v$ between variables that is not entailed by S_{3-i}, then push the equality $u = v$ on the stack S_{3-i} and go to step E2.

E4. [Case split necessary?] For $i = 1, 2$, if the conjunction of literals in S_i is non-convex, then let $u_1 = v_1 \vee \dots \vee u_k = v_k$ be a split entailed by S_i, n_1 the size of S_1, and n_2 the size of S_2. Push the entry $(n_1, n_2, u_2, v_2, \dots, u_k, v_k)$ onto the stack T, push the equality $u_1 = v_1$ onto both S_1 and S_2, and go to step E2.

E5. [Satisfiable.] Halt with the answer "`satisfiable`".

E6. [Try next case.] If the stack T is empty, then the algorithm halts with result "`unsatisfiable`"; else let the top entry on T be $(n_1, n_2, u_1, v_1, \dots, u_k, v_k)$. For $i = 1, 2$, pop S_i as many times as necessary in order to reduce its size to n_i. Push $u_1 = v_1$ on both S_1 and S_2. If $k = 1$ then pop T, else replace the top entry of T with $(n_1, n_2, u_2, v_2, \dots, u_k, v_k)$. Go to step E2. ▮

The procedure always halts, since there can be no more than $n - 1$ non-redundant equalities among n variables. It is not difficult to prove that the procedure is correct when it answers "`unsatisfiable`". To guarantee that it is correct when it answers "`satisfiable`" requires the technical conditions that the two theories be "stably infinite" and have only finitely many free variables. These conditions will not be defined until we prove the correctness of the algorithm in section 18; here we make some observations about the use of the algorithm.

To implement this algorithm efficiently, the satisfiability procedures for $\mathcal{S}$ and $\mathcal{T}$ must have several properties. They must be *incremental* and *resettable*; that is, it must be possible to add and remove literals from the conjunctions without restarting the procedures. The procedures must also detect the equalities and splits entailed by the conjunctions. In theory, any satisfiability procedure can be used to find the equalities and splits, simply by testing each of the finitely many equalities and splits to see if it is entailed. In practice, we construct our satisfiability procedures from algorithms that find the equalities and splits for free, or at very little extra cost.

Here is good news: *if F is a conjunction of $\mathcal{E}$-literals, $\mathcal{R}$-literals, or $\mathcal{L}$-literals, then F is convex.* Therefore, case-splitting is never caused by these satisfiability procedures. It is easy to see that this is the case for $\mathcal{R}$: 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.

Satisfiability procedures for $\mathcal{E}$ and $\mathcal{L}$, and proofs that conjunctions of $\mathcal{E}$-literals and $\mathcal{L}$-literals are convex, are described in sections 6 and 9. As described, these satisfiability procedures take time $O(n^2)$ to process a sequence of instructions of the form "assume the literal ..." and "remove the last-assumed literal", where to assume a literal the procedure determines the satisfiability of the resulting conjunction and propagates any equalities that are entailed, and n is the length of the sequence of instructions in characters. The time bounds can be reduced to $O(n \log n)$ by using a different algorithm.

(The claims above that conjunctions of $\mathcal{L}$-literals are convex and can be processed in $O(n \log^2 n)$ time are subject to the caveat that atom may be used only as a predicate, not as a boolean function embedded in terms.)

The satisfiability procedure for $\mathcal{R}$ is based on the simplex algorithm; it is described in section 12. In the worst case the simplex algorithm takes exponential time. Khachian [29] has discovered a polynomial-time linear programming algorithm, which could be used in $\mathcal{R}$, but it is unlikely to be as fast as the simplex algorithm in practice.

The satisfiability problem for conjunctions of $\mathcal{A}$-literals is NP-complete. The algorithm for $\mathcal{A}$ is described in section 10.

It is lucky that $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{L}$ do not require case-splitting, because most theories do. For example, in the theory $\mathcal{R}^{\times}$, $x \times y = 0 \wedge z = 0$ implies $x = z \vee y = z$. We have already seen that Z causes case-splitting. (Notice that, since it is only necessary to propagate equalities between variables, not between variables and constants, conjunctions such as $\mathrm{int}(z) \wedge 1 \leq z \leq 100$ will not cause hundred-way splits—unless each of the numbers 1, ..., 100 is constrained equal to a variable!) The theory of sets is non-convex; for example, $\{a, b, c\} \cap \{d, e, f\} \neq \{\}$ forces a nine-way case split. It is also true, unfortunately, that the theory $\mathcal{A}$ can in the worst case cause quadratically many splits (see section 10).

We now turn to the problem of proving that Algorithm E is correct if it returns "`satisfiable`". The rigorous proof is deferred until section 18. Here we consider a more informal argument, which, although it will not stand up to close scrutiny, conveys clearly why the algorithm is correct.

The *residue* of a formula F is the strongest boolean combination of equalities between variables entailed by F. Here are several examples of residues:

Formula	Residue
$x = \mathrm{f}(a) \wedge y = \mathrm{f}(b)$	$a = b \supset x = y$
$x \leq y \wedge y \leq x$	$x = y$
$x + y - a - b > 0$	$\neg(x = a \wedge y = b) \wedge \neg(x = b \wedge y = a)$
$x = v^{(i)}_{e}(j)$	$i = j \supset x = e$
$x = v^{(i)}_{e}(j) \wedge y = v(j)$	$\llbracket i = j \Rightarrow x = e;\ x = y \rrbracket$

As another example, here are the residues of the formulas $F_{\mathcal{E}}$ and $F_{\mathcal{R}}$ from the first example in this section:

Residue of $F_{\mathcal{R}}$:	$x = y \wedge (g_2 = g_3 \supset g_1 = z)$
Residue of $F_{\mathcal{E}}$:	$(x = y \supset g_2 = g_3) \wedge g_1 \neq z$

Obviously these residues are inconsistent. The reason Algorithm E works is that the residues are always inconsistent if the conjunctions of literals are:

Fact. *Let $\mathcal{S}$ and $\mathcal{T}$ be two theories satisfying the conditions required by Algorithm E. Let F be a conjunction of $\mathcal{S}$-literals, G a conjunction of $\mathcal{T}$-literals. Then the conjunction of F and G is satisfiable if and only if the conjunction of their residues is satisfiable.* ∎

For theories that do not contain recursive function definitions, this fact may be proved using Craig's interpolation lemma [15], which is proved in many standard textbooks on logic; for example [51]. For theories that do contain recursive function definitions, the fact is a consequence of arguments given in section 18. Assuming the fact, we can prove the correctness of Algorithm E:

Let F be the residue of the conjunction of literals in S_1 and G the residue of the conjunction of literals in S_2 at the moment that step E5 returns "satisfiable." By the above fact, the step will be justified if we can prove that $F \wedge G$ is satisfiable. Let V be the set of all variables appearing in either F or G, E the set of equalities $u = v$ such that u and v are in V and $F \supset u = v$, (equivalently, $G \supset u = v$), and D the set of equalities $u = v$ such that u and v are in V but $F \not\supset u = v$ (equivalently, $G \not\supset u = v$). Let Ψ be any interpretation that satisfies every equality in E and no equality in D. (Since F and G are just boolean combinations of equalities between variables, an interpretation for them can be specified by giving the equalities it satisfies.) We claim Ψ satisfies F and G. Suppose it does not satisfy, say, F. Then F entails the disjunction of all the equalities in D. If D is empty, then F is unsatisfiable, which is impossible since step E2 was passed. If D contains a single equality, then F entails the equality, so the equality would be in E, not D. If D contains more than one equality, then F is non-convex, which is impossible, since step E4 was passed. Thus, Ψ does satisfy F and G, so step E5 is correct. ∎

6. The theory $\mathcal{E}$ and the congruence closure problem

The problem of writing a satisfiability procedure for $\mathcal{E}$ is essentially that of determining whether one equality is a consequence of several other equalities; for example, whether $f(f(a,b),b) = a$ is a consequence of $f(a,b) = a$; or, less obviously, whether $f(a) = a$ is a consequence of $f(f(f(a))) = a$ and $f(f(f(f(f(a))))) = a$. In 1954, Ackermann [1] showed that the problem was decidable, but did not give a practical algorithm. The problem appears to have been ignored for the next twenty-four years. In 1976 and 1977, several people attacked the problem, from quite different points of view. Downey, Sethi and Tarjan [20] viewed the problem as a variation of the common subexpression problem, Kozen [35] as the word problem in finitely presented algebras, Shostak [52] and Nelson and Oppen [43] as the decision problem for the quantifier-free theory of equality.

All these problems reduce to the problem of constructing the "congruence closure" of a relation on a graph, which is defined below. The satisfiability procedures for the theories $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{A}$ are all based on the congruence-closure algorithm. In fact, the algorithm's data structure can be used to represent entry and exit assertions, axioms and lemmas, function definitions, and the values of variables during symbolic execution, much as list structure is used in conventional systems.

Downey, Sethi and Tarjan [20] describe a congruence-closure algorithm that yields satisfiability procedures for $\mathcal{E}$ and $\mathcal{L}$ that run in average time $O(n \log n)$, or worst case time $O(n \log^2 n)$ (or worst-case time $O(n \log n)$ at the cost of using $O(n^2)$ space.)

This section describes the congruence closure problem and its relation to the satisfiability problem for $\mathcal{E}$. The material in this section is based on previous joint work with Derek Oppen, which was reported in [43].

Let R be an equivalence relation on some set of terms (in particular, R is a set of ordered pairs whose components are terms). If $(t, u) \in R$, we say that t is *related by* R to u. By the *domain* of R we mean the set of terms t such that R contains the pair (t, u) for some term u. We assume that the domain of R contains every subterm of any of its terms. Two terms are *congruent under* R if they are of the forms $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$, and if t_i is related by R to u_i for each i from 1 to n. (This term was coined by Downey and Sethi [21].) R is *closed under congruences* if any two terms t and u that are congruent under R and both in the domain of R are related by R. The *congruence closure* of a relation R is the smallest equivalence relation containing R that is closed under congruences. (The "smallest" relation is the one containing the fewest pairs, that is, the finest relation.)

For example, let R be the equivalence relation

$$\{(a,b),\ (b,a),\ (a,a),\ (b,b),\ (\mathrm{f}(a),\mathrm{f}(a)),\ (\mathrm{f}(b),\mathrm{f}(b))\}.$$

Then R is not closed under congruences, since the terms $\mathrm{f}(a)$ and $\mathrm{f}(b)$ of the domain of R are congruent under R but not equivalent under R. If these two terms are merged in R (that is, if R is replaced by the smallest equivalence relation that contains R and the pair $(\mathrm{f}(a), \mathrm{f}(b))$, the resulting relation is congruence-closed and thus is the congruence closure of R.

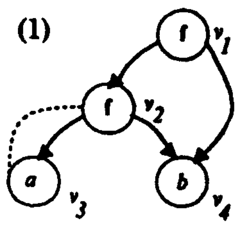
A set of terms can be represented by a labelled directed acyclic graph, and as the nomenclature for directed graphs is often more convenient than that for equivalence relations on terms, we make the following definition: If R is an equivalence relation whose domain is the set of terms V, the *Equality graph* or *E-graph* associated with R is the tuple (V, E, λ, R) where

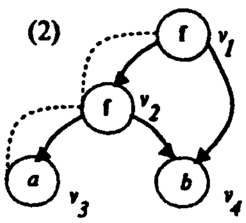
- E is an ordered set of directed edges over V; for each term $f(t_1, \ldots t_n)$ in V, E contains n edges pointing from the term to the terms $t_1, \ldots, t_n$, ordered so that the ith of these edges points at t_i. These are the only edges in E.
- λ labels each term in V with either a function symbol or a variable; $\lambda(f(t_1, \ldots t_n))$ is f, while $\lambda(\mathsf{v}) = \mathsf{v}$ for any variable v.

With these definitions, (V, E, λ) is a labelled directed graph. The elements of V may accurately be called either vertices or terms, since they are vertices of a graph and terms of a formal system. In naming the elements of V, we use "meta" variables like v when emphasizing the term structure, and normal italic variables like v when emphasizing the graph structure. For a vertex v, let $\delta(v)$ be its outdegree, that is, the number of edges leaving v. For $1 \leq i \leq \delta(v)$, let $v[i]$ denote the ith *successor* of v, that is, the vertex pointed to by the ith edge leaving v.

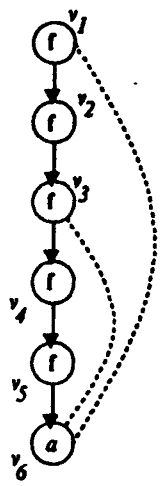
One effect of these definitions is that the congruence closure of R is determined by the graph structure of the E-graph associated with R, since two vertices u and v are congruent if and only if $\lambda(u) = \lambda(v)$, $\delta(u) = \delta(v)$, and $(u[i], v[i]) \in R$ for all i such that $1 \leq i \leq \delta(u)$. As far as computing the congruence-closure goes, we may ignore the internal structure of vertices and view the problem as being defined on any labelled directed graph; but in describing the connection of the problem to mechanical theorem-proving, it is convenient to require that the vertex set actually be a set of terms, whose structure is reflected by the edges and labels.

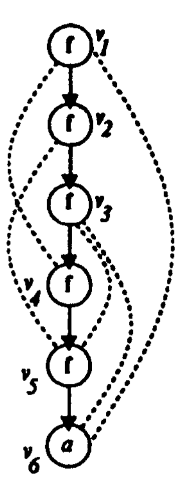
It is laborious to list all pairs in an equivalence relation, so we ordinarily represent an equivalence relation on terms by a picture of its associated E-graph, and connect the related vertices of the graph with dotted lines, omitting redundant lines. For example, let R be the smallest equivalence relation whose domain is the term $\mathrm{f}(\mathrm{f}(a, b), b)$ together with all its subterms, and which includes the pair $(\mathrm{f}(a,b),a)$. Then R is represented by the E-graph (1). Each of the four vertices v_1 to v_4 of the graph is really a term, though we have drawn them as circles containing their labels. The vertices v_1 and v_2 are congruent under R, so the congruence closure of R must include the pairs (v_2, v_3) and (v_1, v_2), as shown in (2). In fact, the finest equivalence relation containing these pairs, namely the equivalence relation with associated partition $\{\{v_1, v_2, v_3\}, \{v_4\}\}$, is closed under congruences and is therefore the congruence closure of R. Notice that this computation is analogous to the deduction of $\mathrm{f}(\mathrm{f}(a,b),b) = a$ from $\mathrm{f}(a,b) = a$ by the substitutivity of equality.

(1)


(2)


As a second example, let R be the smallest equivalence relation whose domain contains $\mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(a)))))$ and all its subterms and which contains the pairs $(\mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(a))))), a)$ and $(\mathrm{f}(\mathrm{f}(\mathrm{f}(a))), a)$. The associated E-graph is shown in (3). Let R' be the congruence closure of R. Vertices v_2 and v_5 are congruent under R, so $(v_2, v_5) \in R'$. Since R' is closed under congruences, $(v_1, v_4) \in R'$. The pairs (v_1, v_4) and (v_1, v_6) are both in R', so $(v_4, v_6) \in R'$. Hence, $(v_3, v_5) \in R'$. (4) shows the pairs we have found; notice that all six vertices are equivalent in the congruence closure. Essentially, we have proved that $\mathrm{f}(\mathrm{f}(\mathrm{f}(a))) = a$ and $\mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(a))))) = a$ together imply $\mathrm{f}(a) = a$. (Thus a congruence closure algorithm can be used to compute greatest common divisors!)

(3)




(4)

That the quantifier-free decision problem for the theory of equality with uninterpreted function symbols can be reduced to the problem of computing congruence closures was proved first by Shostak [52], and independently by Kozen [35] and by Nelson and Oppen [43]. The notation in the following proof is based on [43].

A formula is satisfiable if there exists an interpretation satisfying it (see section 16 for the formal semantics of the logic, which for the purposes of this section is the classical first-order logic with equality). An interpretation Ψ with universe U is a map that assigns to each variable v an element $\Psi(v)$ of U, and to each n-ary function symbol f a map $\Psi(f)$ from U^n to U. An interpretation extends over terms in the obvious way. We also define a *partial interpretation*, which differs from an interpretation in that it may be undefined on some variables, and in that the map it associates with an n-ary function symbol may be undefined on some n-tuples in U^n. Thus if Ψ is a partial interpretation, $\Psi(t)$ will generally be undefined for some terms t.

Let $G = (V, E, \lambda, R)$ be an E-graph such that R is closed under congruences, and for each vertex v of G, let v^* be some cannonical representative of the equivalence class of v under R; for example v^* might be the lexicographically smallest of the vertices equivalent to v. We define a partial interpretation Ψ_G, whose universe is the set of cannonical representatives of equivalence classes of R, as follows. If v is a variable, Ψ_G is defined on v if and only if $v \in V$, and in this case $\Psi_G(v)$ is v^*. If f is a function symbol, $\Psi_G(f)$ is defined on $(v_1{}^*, \ldots, v_n{}^*)$ if and only if there exists a vertex $v \in V$ labelled f such that $v[i]^* = v_i{}^*$ for $1 \leq i \leq n$, and in this case $\Psi_G(f)(v_1{}^*, \ldots, v_n{}^*)$ is v^*. (Notice that $\Psi_G(f)$ is well defined, because if two vertices satisfy this condition, they are congruent, hence equivalent.) In this way we associate a partial interpretation Ψ_G with every E-graph G. (Technically we should write Ψ_G^*, to reflect the fact that the partial interpretation depends not only on the E-graph but also on the projection function $*$, but it rarely matters which cannonical representative is chosen.)

It is trivial to show by induction that if $t \in V$, then $\Psi_G(t)$ is t^*. In general, if $\Psi_G(t)$ is defined, we say that G *represents* t, and that the equivalence class $\Psi_G(t)$ contains a *representative* for t. The set of terms represented may be larger than the set of terms that are vertices; for example the graph for an equivalence relation that relates $f(x)$ to x will represent all terms of the form $f(f(\cdots f(x) \cdots))$.

Theorem E. *Let E be a conjunction of equalities, D a conjunction of disequalities, $G = (V, E, \lambda, R')$ an E-graph where V is the set of terms appearing in E or D, R is the set of pairs (t, u) such that $t = u$ occurs in E, and R' is the congruence closure of R regarded as a relation on V. Then there exists an interpretation satisfying E and D if and only if, for each conjunct $t \neq u$ of D, $\Psi_G(t) \neq \Psi_G(u)$.*

Proof. To prove the "if" part, let Ψ be any interpretation (*not* a partial interpretation) that extends Ψ_G. Then Ψ satisfies both D and E. To prove the "only if" part, we must show that any interpretation satisfying E satisfies all equalities satisfied by Ψ_G; this can be done by a straightforward induction on the number of merges performed in constructing the congruence closure R' from R. ▮

This justifies a simple satisfiability procedure for $\mathcal{E}$: given a conjunction of equalities and disequalities, let R be the smallest congruence-closed relation whose domain is the set of all terms occuring in the conjunction, and which contains the pair (t, u) for each equality $t = u$ in the conjunction. Then the conjunction is satisfible if and only if R contains no pair of terms that are asserted not equal by the conjunction. The next section describes this algorithm more precisely, and in particular describes how to make it incremental.

It follows from Theorem E that $\mathcal{E}$ is convex, since if a conjunction of literals C entails a disjunction of equalities $E_1 \lor \ldots \lor E_n$, then $C \land \neg E_1 \land \ldots \land \neg E_n$ is unsatisfiable, so by Theorem E, one of the disequalities is violated by the E-graph representing the conjunction of the equalities in C; let this disequality be $\neg E_i$, then C entails E_i; hence C is convex. (If the disequality violated is a conjunct of C, then C is unsatisfiable, hence convex.)

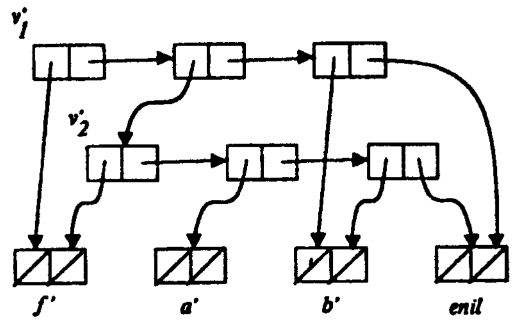
Notice that this solution to the satisfiability problem for $\mathcal{E}$ finds the equalities between variables entailed by the conjunction, as required by the Equality Sharing Procedure of section 5, since the equality $u = v$ is entailed if and only if u and v are in the same equivalence class after the congruence closure is constructed.

7. A congruence-closure algorithm

This section describes a congruence closure algorithm in considerable detail, for the benefit of readers who want to implement the algorithm. Other readers may wish to just skim the section.

The congruence closure algorithm to be described is not the asymptotically fastest but it may be a good choice in practice. Fix a labelled directed graph $G = (V, E)$. The algorithm represents G by a set of linked record structures that will be called *E-nodes*. Each E-node contains nine fields: the *ecar*, *ecdr*, *root*, *pred*, *samecar*, *samecdr*, *size*, *eqclass*, and *back* fields.

The *ecar* and *ecdr* fields are used to represent the graph structure of G, using a version of the standard representation of oriented trees by binary trees. For example, the E-graph of section 6, figure (1) would be represented by the following linked structrue:



(1)

(A diagonal slash through a field means that it contains nil.) It is worth paying attention to the details of this representation, since the algorithm depends on some of its properties that may not be immediately obvious.

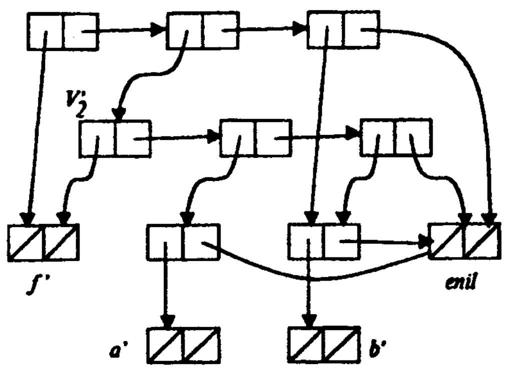
We make two assumptions about the graph G: no vertex with outdegree zero has the same label as a vertex with positive outdegree, and there is at most one vertex of outdegree zero with a given label. These restrictions are certainly satisfied in our application, where all vertices with the same label have the same outdegree, and there is only one vertex for each variable. There is one E-node for each edge, label, and vertex with positive outdegree, and one other distinguished E-node called *enil*. If x is an edge, label, or vertex with positive outdegree, we write x' to denote the E-node that corresponds to it. If v is a vertex of G with outdegree zero, then there is no E-node corresponding to it, but we write v' to mean the E-node corresponding to the label of v. Both *enil* and the nodes corresponding to the labels of G are called *atomic* E-nodes; these have the property that their *ecar* and *ecdr* fields are nil. Let v be a vertex of G with positive outdegree, λ the label of v, and e the first edge leaving v. Then the *ecar* field of v' points at λ' and the *ecdr* field of v' points at e'. For each edge e of G, the *ecar* field of e' points at v', where v is the vertex to which e points; the *ecdr* field of e' points at *enil* if e is the last edge leaving its source vertex; otherwise the *ecdr* field of e' points at f', where f is the next edge after e leaving the source of e.

(Actually, the program would not create the upper-right node in (1), using in its place the node in the middle right, to which it is congruent. But it is simpler to describe the reduction without worrying about this optimization.)

Vertices of outdegree zero are treated specially in order to save space. If they were represented according to the rule used for the other vertices, the structure (1) would be replaced by the structure shown in (2).

We call two E-nodes a and b *congruent* under a relation R on the set of E-nodes if a and b are non-atomic, *ecar*(a) is related by R to *ecar*(b), and *ecdr*(a) is related by R to *ecdr*(b). The congruence closure of a relation on the set of E-nodes is defined in the obvious way. It is easy to show that u and v are equivalent in the congruence closure of $R = \{(u_1, v_1), \ldots, (u_p, v_p)\}$ if and only if u' and v' are equivalent in the congruence closure of $R' = \{(u_1', v_1'), \ldots, (u_p', v_p')\}$.

We call an E-node a *vertex node* if it is v' for some vertex v in G, an *edge node* if it is e' for some edge e of G, and a *label node* if it is neither a vertex node, an edge node, nor *enil*. If a is



(2)

a vertex node or a label node, then $a \not\approx ecdr(b)$ for all nodes b; if a is an edge node or *enil*, then $a \not\approx ecar(b)$, for all nodes b. This is a useful fact, especially in light of the following lemma.

Lemma 1. *If u_1, v_1, ..., u_n, v_n is any set of vertices of G, then each equivalence class of the congruence closure of $R = \{(u_1', v_1'), \ldots, (u_p', v_p')\}$ is either a singleton, or consists entirely of vertex nodes or entirely of edge nodes.*

Proof. Let a be a label node or *enil*; then $\{a\}$ is an equivalence class of the smallest equivalence relation containing R, since a does not represent any vertex of G. But a will never become congruent to any other node during the construction of the congruence closure of of R, since a is an atomic node. Hence $\{a\}$ is an equivalence class of the congruence closure of R.

Each equivalence class of the smallest equivalence relation containing R that is not a singleton consists entirely of vertex nodes; so it suffices to prove that in constructing the congruence closure, no vertex node will ever become congruent to an edge node. But this is obvious, since the *ecar* of a non-atomic vertex node is a label node, while the *ecar* of an edge node is a vertex node. ▮

Notice that if the expression $g(a)$ were a term, then a graph that represented it along with $f(g, a)$ would have a vertex node equivalent to an edge node. We avoid this problem by regarding $g(a)$ as an abbreviation for **select**(g, a).

The *root*, *eqclass*, and *size* fields are used to represent an equivalence relation on the nodes. The *root* field of the node a points at the canonical representative of the equivalence class of a, and the equivalence class is circularly linked by the *eqclass* fields. We will assume that **union** is applied only to canonical representatives (which will hereafter be called *root nodes*). Here is the code for **union**:

```
Procedure union(u, v) ≡
  ⟦ reroot(u, v, u);                                   Make roots of nodes in u's class be v
    size(v) ← size(v) + size(u);                       Set size of v. Now splice class lists:
    (eqclass(u), eqclass(v)) ← (eqclass(v), eqclass(u)) ⟧;
```

```
Procedure reroot(u, newroot. done) ≡
  ⟦ root(u) ← newroot;
    eqclass(u) = done ⇒ ⟦ ⟧;
    reroot(eqclass(u), newroot, done) ⟧;
```

The *size* field is the size of a node's equivalence class. The code below will call **union**(u, v) only when u's equivalence class is smaller than v's; the worst-case cost of n calls to **union** under this scheme is $O(n \log n)$.

Let Q be any equivalence class and S the set of all nodes b such that the *ecar* of b is in Q. Then the nodes in S are circularly linked by their *samecar* fields. The *samecdr* field is used similarly. Thus two nodes are congruent if they are in the same *samecar* circle and the same *samecdr* circle.

A node b is a *predecessor* of an equivalence class Q if either $ecar(b) \in Q$ or $ecdr(b) \in Q$. If Q has no predecessors, then $pred(a) =$ nil, where a is the canonical representative of Q; otherwise $pred(a)$ points at some predecessor of Q. Naturally, we assume that nil is not equal to any E-node.

Given a vertex node or label node a, it follows from Lemma 1 that we can find all predecessors of a's equivalence class by following the *pred* field of $root(a)$ and then following the *samecar* fields around in a circle. To find the predecessors of an equivalence class of edge-nodes, or of the equivalence class containing *enil*, follow the *pred* field of the root of the class and then follow the *samecdr* fields.

This scheme uses three pointer fields per node to represent the predecessors of equivalence classes. If the root of each equivalence class contained a pointer to a list of its predecessors, then each non-atomic node would appear in two lists, and the lists would have two pointers per entry, so five pointers per node would be required.

Suppose Q_1 and Q_2 are equivalence classes of vertex nodes and that the nodes a and b are made congruent by merging Q_1 and Q_2. Then the *ecar* fields of a and b must point at nodes in Q_1 and Q_2 (not necessarily respectively), and the *ecdr* fields of a and b must point at equivalent nodes. Therefore, we can find all such pairs a, b as follows: For each predecessor a of Q_1, store in $root(ecdr(a))$ a backpointer to a. Then, for each predecessor b of Q_2, if $root(ecdr(b))$ contains a backpointer, then b is congruent to the node pointed to by the backpointer. After finding all congruent pairs it is necessary to traverse the predecessors of Q_1 again to remove all the backpointers.

The following procedure **mergepair** takes two nodes, combines their equivalence classes with **union**, finds all new congruent pairs of vertices, and adds each such pair to the list *mergelist*:

```
Procedure mergepair$(u, v) \equiv$
  ⟦$(u, v) \leftarrow (root(u), root(v))$;
   $u = v \Rightarrow$ ⟦ ⟧;
   ⟦$size(u) > size(v) \Rightarrow (u, v) \leftarrow (v, u)$⟧;        Now $v$ has the larger class.
   union$(u, v)$;                                                    Now $v$ is the new root.
   ⟦undo$(u)$; undo$(merge)$⟧;                                       See comments below.
   $pred(u) =$ nil $\Rightarrow$ ⟦ ⟧;                                If $u$ or $v$ has no predecessors,
   $pred(v) =$ nil $\Rightarrow pred(v) \leftarrow pred(u)$;          done, else prepare for loops.
   with $link = samecar, other = ecdr$ do                            $u$ a vertex node $\rightarrow$ predecessors linked by $samecar$
     ⟦⟦$root(ecdr(pred(u))) = u \Rightarrow (link, other) \leftarrow (samecdr, ecar)$⟧;    else by $samecdr$.
       markpred$(pred(u), pred(u))$;                                 Mark predecessors of $u$.
       checkpred$(pred(v), pred(v))$;                                Check predecessors of $v$.
       unmarkpred$(pred(u), pred(u))$;                               Unmark. Finally, spice predecessor lists:
       $(link(pred(u)), link(pred(v))) \leftarrow (link(pred(v)), link(pred(u)))$⟧⟧;
```

The procedure does nothing if u and v are already equivalent. Otherwise, it combines the equivalence classes with **union** and lets v be the new root. The significance of **undo** will be explained later. If u has no predecessors, there are no new congruent pairs and all predecessors of the new class are predecessors of the new root v. If u has predecessors and v does not, then there are no new congruent pairs, but the predecessor field of the new root v must be changed to point into the ring of predecessors of u. If both u and v have predecessors, three loops are used to find any new congruent pairs. There are two versions of each of these loops, one version where the link field is the *samecar* field and the "other" successor of the predecessors is pointed at by the *ecdr* field, and one version where the link field is *samecdr* and the other successor is the *ecar*. The conditional in the middle of mergepair determines whether u is a vertex node or an edge node and sets *link* and *other* to the appropriate field selectors. (Note that the formal system described in section 4 allows "variable field selectors".) The three loops markpred, checkpred, and unmarkpred use *link* and *other* to find the new congruent pairs. Finally, the circular predecessor lists of u and v are spliced.

Here is the code for the three loops:

```
Procedure markpred($v, done$) ≡
   ⟦ $back(other(v)) \leftarrow v$;
     $link(v) = done \Rightarrow$ ⟦ ⟧;
     markpred($link(v), done$) ⟧;

Procedure checkpred($v, done$) ≡
   ⟦ ⟦ $back(other(v)) \neq$ nil $\Rightarrow$
       $mergelist \leftarrow$ cons(cons($v, back(other(v))$), $mergelist$) ⟧;
     $link(v) = done \Rightarrow$ ⟦ ⟧;
     checkpred($link(v), done$) ⟧;

Procedure unmarkpred($v, done$) ≡
   ⟦ $back(other(v)) \leftarrow$ nil;
     $link(v) = done \Rightarrow$ ⟦ ⟧;
     unmarkpred($link(v), done$) ⟧;
```

(The space for the *back* fields—one pointer per node—can be reduced to one bit per node, plus space for an auxiliary stack. The bit is cleared to represent a *back* field that is nil; if the bit is set, then the *pred* field contains the value of the *back* field and the true *pred* field is on the auxiliary stack. The procedure unmarkpred restores the *pred* fields while it clears the bits.)

The actual congruence closure is computed by the procedure mergeloop, which removes pairs from *mergelist* and merges them with mergepair, continuing until *mergelist* is empty, at which time the equivalence relation must be closed under congruences:

```
Procedure merge($u, v$) ≡
   ⟦ $mergelist \leftarrow$ cons(cons($u, v$), $mergelist$); mergeloop() ⟧.

Procedure mergeloop() ≡
   with $p =$ nil do
      ⟦ $mergelist =$ nil $\Rightarrow$ ⟦ ⟧;
        ($p, mergelist$) $\leftarrow$ (car($mergelist$), cdr($mergelist$));
        mergepair(car($p$), cdr($p$));
        mergeloop() ⟧;
```

If the original graph G has m edges, then there are $O(m)$ E-nodes. The number of calls to union is $O(m)$; the worst-case cost for the three loops is $O(m)$, so the cost of the whole algorithm is $O(m^2)$.

Congruence closure algorithms may be classified according to how they find the new congruent pairs given the lists of predecessors of the two equivalence classes being combined. The most obvious method is a double loop over all pairs of predecessors, with cost $O(ab)$, where a and b are the lengths of the predecessor lists. Interestingly enough, an $O(m^2)$ algorithm can be obtained using this double loop. (See Nelson and Oppen [43].) The program above finds the new congruent pairs in $O(a+b)$ time. Another way to find the pairs in $O(a+b)$ time is by lexicograpically sorting the predecessor lists. By using lexicographic sorting and not reducing the problem to the binary case, an $O(nm)$ algorithm can be constructed, but it would probably be slower in practice than the method described here, because of the overhead of the lexicographic sorting. In the algorithm of Downey, Sethi and Tarjan [20], each node a is kept in a hash table or binary tree under the key $(root(ecar(a)), root(ecdr(a)))$. When merging two equivalence classes, only the predecessors in the shorter list need to be rehashed; thus the cost of finding new congruent pairs is $\min(a, b)$ accesses to the hash table or binary tree. The average time for $O(m)$ merges using their algorithm with a hash table is $O(m \log m)$; the worst case time using a balanced tree is $O(m \log^2 m)$; the worst case time using a "trie" is $O(m \log m)$, but the trie requires $O(m^2)$ storage.

The algorithm of Downey, Sethi, and Tarjan has been implemented by the author in an experimental verifier and by Derek Oppen in the Stanford Pascal Verifier. The experimental verifier never became stable enough to support meaningful experiments. In the Pascal verifier, the $O(m \log m)$ algorithm was slightly slower than the $O(m^2)$ algorithm on the ten or fifteen problems on which it was tried. Since the structure of the graph depends on many volatile factors, such as the matching algorithm (see section 14) and the interface with the arithmetic routines (see section 13), it is not yet clear what variation of the congruence-closure algorithm is the best choice.

The worst case for the algorithm described here is $O(m^2)$, but if most of the merges are "balanced", in the sense that the two classes being combined have approximately the same number of predecessors, then the total time required will be $O(m \log m)$. The average behaviors of several algorithms for maintaining an equivalence relation under the **union** operation, assuming various distributions for the merges, have been computed by Doyle and Rivest [22], Andrew Yao [63], and Knuth and Schönhage [33]. These results do not apply to the congruence closure algorithm, because postulating a distribution for the merges that come "from the outside" does not determine, in any obvious way, the distribution of merges that are performed by the algorithm. For example, suppose that the distribution of the calls to **union** from the outside is that analyzed by Doyle and Rivest, in which the probability that two classes are merged is independent of their sizes. This distribution tends to make naïve algorithms look good, because it leads to lots of balanced merges—in fact, it is not difficult to show that if *all* the merges obeyed this distribution, the average case of the algorithm described here would be $O(n \log n)$. Unfortunately, even if the merges from the outside obeyed this distribution, presumably a large class would be more likely to be merged by the congruence closure algorithm than a small class. If we assume that the probability that two classes are merged is proportional to the product of their cardinalities, then the results of Knuth and Schönhage [33] show that the average time of the algorithm becomes $O(n^2)$.

The procedure **mergepair** provides an incremental congruence-closure algorithm, but to obtain an incremental satisfiability procedure for $\mathcal{E}$, it is necessary to handle disequalities incrementally. Given a disequality $t \neq u$, we must add structure to the graph so that if the nodes representing t and u ever become equivalent, the program will detect it.

The obvious way to do this is to store with the root of each equivalence class a circular list of nodes that are forbidden to join the class. Given two root nodes a and b that are to be made "unmergeable", we add, say, a to the list associated with b. Whenever two roots u and v are merged, we check if u is the root of anything in v's list, or if v is the root of anything in u's list; if so, the context is unsatisfiable; otherwise we splice the two lists.

The worst case cost of this scheme is $O(n^2)$ (where n is now the total length of the input conjunction, including equalities and disequalities), and it is not very good in practice, either (at least for the formal system described in this paper), because the list of things that are forbidden to join the equivalence class of **true** is likely to become very long. A reasonable scheme in practice is to forbid the merge of a and b by adding a to b's list if b's list is shorter than a's, otherwise adding b to a's list. This method still has worst-case cost $O(n^2)$. To hold the worst-case to $O(n \log n)$, forbid the merge of a and b by adding a to b's list *and* adding b to a's list. Then to test if the merge of u and v violates any disequalities, it it only necessary to search the shorter of u's list and v's list. That the cost of this method is $O(n \log n)$ follows from the fact that every time a list is traversed, it is as least doubled in size. The cost of two pointer fields per disequality is very reasonable, so this method is probably best in practice.

We will describe the modifications to the program above explicitly. The procedure **forbidmerge** takes two E-nodes and makes them "unmergeable"; it sets the variable *inconsistent* if the two nodes are already equivalent. The new version of **mergepair** will set *inconsistent* if the requested merge has been forbidden. We assume that E-nodes contain the field *forbid*, which is either nil, or points at a circular list of nodes with the two fields *enode* and *link*. We assume the existence of a list *avail* of available nodes of this form. Here is the code for **forbidmerge**:

```
Procedure forbidmerge($u, v$) $\equiv$
  [[ $(u, v) \leftarrow (root(u), root(v))$;
     $u = v \Rightarrow inconsistent \leftarrow$ true;
     addforbid($u, v$);
     addforbid($v, u$);
     [[ undo($u$); undo($forbidmerge$) ]] ]].

Procedure addforbid($u, v$) $\equiv$
  with $temp$ = nil do
    [[ $avail$ = nil $\Rightarrow$ error();
       $(temp, avail) \leftarrow (avail, link(avail))$;
       $(enode(temp), link(temp)) \leftarrow (v, temp)$;
       $forbid(u)$ = nil $\Rightarrow forbid(u) \leftarrow temp$;
       $(link(forbid(u)), link(temp)) \leftarrow (link(temp), link(forbid(u)))$ ]].
```

The procedure **mergepair** is modified by adding the following code fragment just before the call **union**(u, v) (note that this call will make v the new root); the fragment checks that the merge has not been forbidden, and arranges that the forbid-list of the new root v will contain the concatenation of the old lists for u and v:

```
[[ $forbid(u)$ = nil $\Rightarrow$ [[ ]];
   $forbid(v)$ = nil $\Rightarrow forbid(v) \leftarrow forbid(u)$;
   checkforbid($forbid(u), forbid(u), forbid(v), forbid(v)$) $\Rightarrow inconsistent \leftarrow$ true;
   $(link(forbid(u)), link(forbid(v))) \leftarrow (link(forbid(v)), link(forbid(u)))$ ]];
$inconsistent \Rightarrow$ [[ ]];
```

where the procedure **checkforbid** is defined by:

```
Procedure checkforbid($p, endp, q, endq$) $\equiv$
  [[ $eroot(enode(p)) = v \lor eroot(enode(q)) = u \Rightarrow$ true;
     $link(p) = endp \lor link(q) = endq \Rightarrow$ false;
     checkforbid($link(p), endp, link(q), endq$) ]].
```

The calls to the function **undo** record the changes made to the E-graph so as to make it possible to mark the state of the data structure and later return to the marked state by "undoing" the merges performed since the mark. Notice that such a facility is required by the equality sharing procedure of section 5, since in step E6 the procedure pops the stack of assumed literals, which is equivalent to undoing the effect of the last call to **mergepair** or **forbidmerge**. The function **undo** does nothing but push its argument on a stack:

```
Procedure undo(x) ≡
  ⟦ undoptr = hib(undostack) ⇒ error();
    undoptr ← undoptr + 1;
    undostack(undoptr) ← x ⟧.
```

The procedures **mergepair** and **forbidmerge** each push two entries on *undostack*, one of which is a pointer to an E-node, the other of which is a conventional mark (either *merge* or *forbid*, which are assumed to be distinct values) whose purpose is to distinguish the entries made by the two procedures. To save the state of the data structure, call **undo**(*push*); that is, push the conventional mark *push* on stack. (We assume *push* is distinct from *merge* and *forbid*.) To restore the last saved state, call the function **pop**, which removes and processes entries on the stack until it gets to an entry *push*, when it stops. It processes *merge* entries and *forbid* entries by undoing the effect of the corresponding procedures. Here is the code:

```
Procedure pop() ≡
  with temp = undostack(undoptr) do
    ⟦ undoptr ← undoptr − 1;
      temp = push ⇒ ⟦ ⟧;
      temp = merge ⇒
        ⟦ undomerge(undostack(undoptr)); undoptr ← undoptr − 1; pop() ⟧;
      temp = forbid ⇒
        ⟦ undoforbid(undostack(undoptr)); undoptr ← undoptr − 1; pop() ⟧ ⟧.
```

where the procedures **undomerge** and **undoforbid** are defined by:

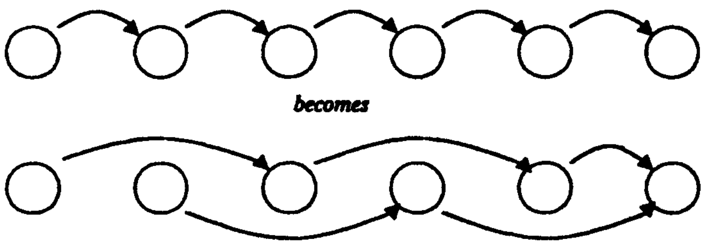
```
Procedure undomerge(u) :
  ⟦ v ← root(u);                                          Now u and v are as at end of merge.
    size(v) ← size(v) − size(u);                          Restore v's old size.
    (eqclass(u), eqclass(v)) ← (eqclass(v), eqclass(u));  Undo splice.
    reroot(u, u, u);                                      Reset root pointers in u's class.
    ⟦ forbid(u) = nil ⇒ ⟦ ⟧                               Do nothing if u had no forbidlist.
      forbid(u) = forbid(v) ⇒ forbid(v) ← nil             If v had empty forbidlist, restore it.
      (link(forbid(u)), link(forbid(v))) ←                Else unsplice.
         (link(forbid(v)), link(forbid(u))) ⟧;
    pred(u) = nil ⇒ ⟦ ⟧;                                  Exit if u had no predecessors.
    pred(u) = pred(v) ⇒ pred(v) ← nil;                    Exit if v had no predecessors.
    link ← ⟦ u = root(ecar(pred(u))) ⇒ samecar; samecdr ⟧;        Else unsplice.
    (link(pred(u)), link(pred(v))) ← (link(pred(v)), link(pred(u))) ⟧.
```

```
Procedure undoforbid(u) ≡
  with v = enode(forbid(u)) do ⟦ undoaddforbid(u); undoaddforbid(v) ⟧.
```

```
Procedure undoaddforbid(u) ≡
  with temp = forbid(u) do
    ⟦ link(temp) = temp ⇒ (avail, forbid(u), link(temp)) ← (forbid(u), nil, avail);
      (avail, link(temp), link(link(temp))) ← (link(temp), link(link(temp)), avail) ⟧.
```

Note that *undostack* contains both pointer entries and "tag" entries that presumably are integers from some ennumerated type. In later sections, we will place groups of entries on the undo stack that consist of several different pointers and integer counters, followed by a distinguished tag to indicate the type of the group. This is similar to a typical control stack that contains different kinds of records of different sizes, but is such that the type of the top record can always be determined by some tag field. This would be illegal in a typical hard-typed language, but using the approach described in section 4, it is easy to attach mechanically verifiable invariants that establish to the compiler that the program uses the type system consistently. This is a good example of a situation where program verification can liberate the programmer from excessively rigid type systems, without compromising reliability.

The algorithm used for union is particularly "undoable", which is one of the reasons it was chosen. The algorithm analysed by Tarjan [58] is faster in the worst case, but it uses incremental path compression, which is difficult to undo. One form of incremental path compression which can be undone reasonably efficiently is "path-halving", in which the path x, $r(x)$, $r(r(x))$ from a node x to its root is split into the two paths x, $r(r(x))$, $r(r(r(r(x))))$, …, and $r(x)$, $r(r(r(x)))$, $r(r(r(r(r(x)))))$, …, by replacing each link that used to go from a node n to $r(n)$ by a link going from n to $r(r(n))$:



This operation can be undone by "zipping" the two paths together again.

8. A method for finding satisfiability procedures

Before describing the satisfiability procedures for $\mathcal{L}$ and $\mathcal{A}$, we illustrate the method used to derive the procedures by applying it to the problem of discovering a satisfiability procedure for the simple theory with the two axioms:

$$(\forall x\, y) \quad \mathrm{car}(\mathrm{cons}(x, y)) = x \tag{1}$$

$$(\forall x\, y) \quad \mathrm{cdr}(\mathrm{cons}(x, y)) = y \tag{2}$$

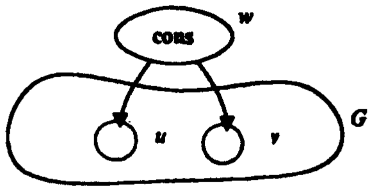
Recall that in section 6 we associated a partial interpretation Ψ_G with each E-graph G. We must define what it means for a partial interpretation to "satisfy" a formula. If F is a quantifier-free formula and Ψ is a partial interpretation such that $\Psi(t)$ is defined for all terms t appearing in F, then the truth value of F may be defined in Ψ in the obvious way. We say that a partial interpretation Ψ *satisfies* a formula F if (a) F is quantifier-free, Ψ is defined on every term appearing in F, and the truth-value of F in Ψ is *true*, or (b) F is of the form $(\forall x)\ G$, and for all u in the universe of Ψ, G is satisfied by $\Psi^{(x)}_{u}$, or (c) F is of the form $(\exists x)\ G$, and there exists a value u in the universe of Ψ such that G is satisfied by $\Psi^{(x)}_{u}$. If Ψ happens to be total, then this definition agrees with the conventional definition of what it means for an interpretation to satisfy a formula. (This definition only works for "prenex" formulas in which all quantifiers are on the outside; this will be sufficient for our purposes.)

Given an E-graph G, it is straightforward to check whether the partial interpretation Ψ_G satisfies (1) and (2), because, for example, Ψ_G violates (*i.e.* fails to satisfy) axiom (1) if and only if one of its equivalence classes contains a representative for a term of the form car(cons(t, u)), but does not contain a representative for t. An equivalence class contains a representative for a term of the form car(cons(t, u)) if and only if it contains a vertex labelled **car** whose successor's equivalence class contains a vertex labelled **cons**; this condition is easy to test. Thus, our problem would be solved if the following assumption were true:

Optimistic Assumption. *Let G be an E-graph such that Ψ_G satisfies (1) and (2). Then there exists an interpretation Ψ that extends Ψ_G and satisfies (1) and (2).* ∎

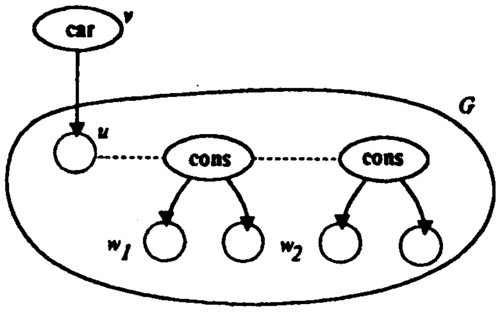
If this assumption were true, we could determine the satisfiability of a conjunction $E \wedge D$, where E is a conjunction of equalities and D is a conjunction of disequalities, by contructing an E-graph representing all terms occuring in E and D; merging as required to satisfy E, (1), and (2); and testing whether the resulting partial interpretation satisfies D.

There is a systematic method for testing the truth of assumptions like the one above: test whether the partial interpretation can be extended over a slightly larger E-graph without violating (1) or (2); if so, the extention can be repeated indefinitely, and the assumption will be proved. For example, let G be an E-graph such that Ψ_G satisfies (1) and (2), u and v vertices of G, and let G' be formed from G by adding a vertex w labelled **cons** with successors u and v:



If w is congruent to some vertex of G, then the "extension" was a no-op, and $\Psi_{G'}$ certainly satisfies (1) and (2). But if w is not congruent to an existing vertex, then w will represent no term of the form car(cons(t, u)) or cdr(cons(t, u)), hence $\Psi_{G'}$ will satisfy (1) and (2). So far so good.

Again let Ψ_G satisfy (1) and (2), and consider extending G to G' by adding a vertex v labelled **car** over an existing vertex u. This creates a representative for a new instance of (1) for every vertex in u's equivalence class labelled **cons**; and these are the only new instances of (1) or (2). Here is an illustration of the case where there are two vertices equivalent to u labelled **cons**:



In order to make the interpretation $\Psi_{G'}$ satisfy (1), we must merge v with both w_1 and w_2. If these two vertices are already equivalent in G, there is no problem; but if they are not, this merge prevents $\Psi_{G'}$ from being an extension of Ψ_G, since the terms w_1 and w_2 are interpreted differently by Ψ_G, but identically by $\Psi_{G'}$. Thus, our optimistic assumption is false; we have found a graph with no contradictions of (1) or (2) that cannot be extended as required. But this "obstruction" will be avoided if we require that Ψ_G satisfy:

$$(\forall x\, y\, u\, v) \quad \mathbf{cons}(x, y) = \mathbf{cons}(u, v) \supset x = u. \tag{3}$$

Similarly, considering extentions by cdr, we require that G satisfy:

$$(\forall x\, y\, u\, v) \quad \mathbf{cons}(x, y) = \mathbf{cons}(u, v) \supset y = v. \tag{4}$$

Thus we try:

Revised Assumption. *Let G be an E-graph such that Ψ_G satisfies (1), (2), (3), and (4). Then there exists an interpretation Ψ that extends Ψ_G and satisfies (1), (2), (3), and (4).* ▮

This assumption turns out to be true, as the reader can verify by showing that G can be extended with a vertex labelled **car** or **cdr** over any existing vertex, or by a vertex labelled **cons** over any existing pair of vertices, without merging any existing equivalence classes, and without introducing any violations of (1), (2), (3), or (4). That this suffices to prove the assumption is demonstrated formally at the end of this section.

The revised assumption justifies a simple satisfiability procedure for the theory axiomatized by (1) and (2): To determine if a conjunction $E \wedge D$ is satisfiable in this theory, where E is a conjunction of equalities and D is a conjunction of disequalities, construct an E-graph representing all terms in E and D and merge as necessary to satisfy E, (1), (2), (3), and (4). The conjunction is satisfiable if and only if no disequality in D is contradicted by these merges.

If the revised assumption were false, the E-graph which proved it false would provide a new formula to add to the list (1), (2), (3), and (4); thus we get a sequence of sharper and sharper assumptions; if the sequence is finite, we will have found a satisfiability procedure. (Unfortunately, the method usually diverges; as described below.)

This method is similar to the Bendix-Knuth procedure (see [32]) for "completing" a set of rewrite rules. The Bendix-Knuth procedure shows that (1) and (2) constitute a complete set of rewrite rules by themselves. Thus, developing a complete set of rewrite rules does not suffice to give a satisfiability procedure. The Bendix-Knuth procedure would not help to prove the theorem

$$\mathbf{cons}(x, y) = \mathbf{cons}(u, v) \supset x = u.$$

It would be interesting to investigate the relation between the method of the section, the Bendix-Knuth procedure, and the various techniques for extending resolution theorem-proving to handle equality.

The method described above works on the axioms (1) and (2), but on most theories, even simple ones, the method fails. For example, if it is applied the axioms of $\mathcal{A}$, starting with:

$$a^{(e)}_{i}(i) = e \tag{5}$$

$$i \neq j \supset a^{(e)}_{i}(j) = a(j) \tag{6}$$

it produces in the first "round"

$$a^{(e)}_{i} = b^{(f)}_{i} \supset e = f \tag{7}$$

$$a^{(e)}_{i} = b^{(f)}_{j} \wedge k \neq i \wedge k \neq j \supset a(k) = b(k) \tag{8}$$

$$k \neq i \wedge k \neq j \supset a^{(x)}_{i}(k) = a^{(y)}_{j}(k) \tag{9}$$

and in subsequent rounds it becomes clear that the procedure will run endlessly, as it elaborates (8) into such useless conditions as

$$a^{(e)}_i = b^{(f)}_j \wedge a^{(g)}_l = c^{(g)}_m \wedge k \neq i \wedge k \neq j \wedge k \neq l \wedge k \neq m \supset b(k) = c(k). \tag{10}$$

It is, unfortunately, possible for a graph to contradict (10) without contradicting any of (5) through (9). Notice, however, that if the graph contained representatives for the terms $b^{(j)}_f(k)$, $c^{(m)}_g(k)$, and $a(k)$ then the equality $b(k) = c(k)$ of (10) would be forced by four instantiations of (6). This suggests that we require of the graphs not only that their associated partial interpretations satisfy some set of formulas, but also that they "have enough vertices" in some sense. In fact, the satisfiability procedure described in section 10 for $\mathcal{A}$ introduces vertices labelled select and then looks for violations of (5) and (6); because of the added vertices it is not necessary to check (7) through (9) or any of the other formulas that the method would produce.

Similarly, the procedure seems to diverge if it is applied to the full set of axioms for $\mathcal{L}$, but by introducing vertices labelled **car** and **cdr** over vertices labelled **cons**, a simple satisfiability procedure can be obtained.

The satisfibility procedures for $\mathcal{L}$ and $\mathcal{A}$ were discovered by applying the above method by hand and observing what vertices should be added to the graph to make the extension step work. It is not clear how this process can be mechanized.

The following lemma justifies the method outlined in this section:

Lemma X. *Let F be a formula, G_0 an E-graph such that Ψ_{G_0} satisfies F, and suppose that for any E-graph G such that Ψ_G satisfies F, for any function symbol f, and for any terms $t_1, \ldots, t_n$ that are represented in G, where n is the arity of f, there exists an E-graph G' such that $\Psi_{G'}$ extends Ψ_G, $\Psi_{G'}$ satisfies F, and $\Psi_{G'}(f)(t_1, \ldots, t_n)$ is defined. Then F is satisfiable.*

Proof. Let T be the set of all terms t such that every variable appearing in t appears in F. Well-order T in any way such that a term is preceded by all its subterms. Construct G_1, G_2, ... by the following rule: Let t_i, for $i = 0, 1, \ldots$, be the first term in the well-ordering that is not represented in G_i. Let G_{i+1} be an E-graph that represents t_i, satisfies F, and is such that $\Psi_{G_{i+1}}$ extends Ψ_{G_i}; that such a graph exists follows from the condition of the theorem. The union of a chain of functions, each of which extends the previous one, is itself a function; let Ψ be the union of the Ψ_{G_i}. By construction, $\Psi(f)$ is total for all function symbols f. Thus Ψ fails to be an interpretation only in that it is still undefined on variables that do not appear in F; let Ψ' have the same universe as Ψ and extend Ψ by interpreting variables not appearing in F arbitrarily. Then Ψ' is an interpretation that satisfies F; hence F is satisfiable. ∎

9. The satisfiability procedure for $\mathcal{L}$

The theory $\mathcal{L}$ may be axiomatized as follows:

$$(\forall x\, y)\ \mathbf{car}(\mathbf{cons}(x, y)) = x \tag{1}$$
$$(\forall x\, y)\ \mathbf{cdr}(\mathbf{cons}(x, y)) = y \tag{2}$$
$$(\forall x)(\exists y\, z)\ \neg\mathbf{atom}(x) \supset x = \mathbf{cons}(y, z) \tag{3}$$
$$(\forall x\, y)\ \neg\mathbf{atom}(\mathbf{cons}(x, y)) \tag{4}$$
$$\mathbf{atom}(\mathbf{nil}) \tag{5}$$

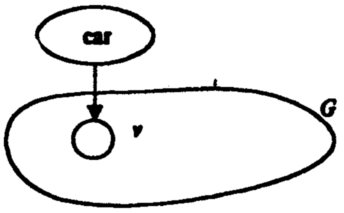
The satisfiability procedure for $\mathcal{L}$ rests on the following theorem:

Theorem L. *Let E be a conjunction of equalities, D a conjunction of disequalities, G_0 an E-graph that represents every term appearing in E or D, and suppose that Ψ_{G_0} satisfies E, D, (1), (2), (3), (4), and (5), and that every vertex of G_0 labelled* **cons** *has predecessors labelled* **car** *and* **cdr**. *Then $E \wedge D$ is satisfiable in the theory $\mathcal{L}$.*

Proof. Let $\mathcal{G}$ be the class of E-graphs that satisfy the conditions of the theorem. We show that for any $G \in \mathcal{G}$, for any function symbol f, and for any n vertices $v_1, \ldots, v_n$ of G, where n is the arity of f, there exists an E-graph $G' \in \mathcal{G}$ such that $\Psi_{G'}$ extends Ψ_G and G' represents $f(v_1, \ldots, v_n)$. This suffices to prove the theorem, given Lemma X.

It is trivial to show that the extension if possible if f is **nil** or if f is not a free function of $\mathcal{L}$. Thus it suffices to prove the extension is possible for the function symbols **car**, **cdr**, and **cons**.

Let v be a vertex of G such that G does not represent the term **car**(v). Let G' be formed from G by adding a new vertex **car**(v) in its own equivalence class:

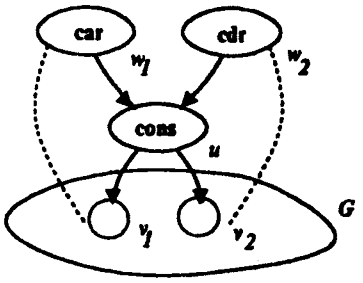


Then $\Psi_{G'}$ obviously satisfies (2), (3), (4), and (5). Since we are assuming that vertices labelled **cons** have predecessors labelled **car**, and that **car**(v) is not represented in G, it follows that v is not equivalent to any vertex labelled **cons**, therefore every term of the form **car**(**cons**(t, u)) represented in G' is already represented in G; thus $\Psi_{G'}$ satisfies (1), because Ψ_G does. Hence $G' \in \mathcal{G}$.

Similarly, we can extend the graph with a vertex labelled **cdr**.

Let v be a vertex of G such that G does not represent the term **atom**(v), and let G' be formed from G by adding a vertex u labelled **atom** over the vertex v. If the equivalence class of v contains a vertex labelled **cons**, then inspection reveals that $\Psi_{G'}$ satisfies (1) through (5). If the equivalence class of v contains no vertex labelled **cons**, then to satisfy (3) we merge u with the vertex for **true** (adding a vertex for **true** if necessary); let this merge be performed on G'. Since u is not congruent to any existing vertex, this merge adds a term to an existing equivalence class without merging any existing classes, thus the cannonical representative of the augmented class can be preserved, so that $\Psi_{G'}$ extends Ψ_G. Inspection reveals that $\Psi_{G'}$ also satisfies (1), (2), (4), and (5).

Finally, let G' be formed from G by adding a vertex u labelled **cons** over the two vertices v_1 and v_2; and simultaneously adding vertices w_1 and w_2, labelled **car** and **cdr** respectively, over u; and merging w_1 with v_1 and w_2 with v_2:



Inspection reveals that $\Psi_{G'}$ satisfies (1) through (5). This completes the proof of the theorem. ∎

Corollary. *Let C be a conjunction of atomic formulas each of which has one of the forms $t = u$, $t \neq u$,* **atom**$(t) =$ **true**, *or* **atom**$(t) \neq$ **true**, *for some term t or terms t and u that are composed of variables and the function symbols* **car**, **cdr**, **cons**, *and* **nil**. *Then C is convex.*

Proof. Construct an E-graph G as follows: initialize G to be the E-graph associated with the identity relation on the set of terms appearing in C. For each conjunct $t = u$ of C, merge t and u in G. For each vertex **cons**(t, u) of G add the vertices **car**(**cons**(t, u)) and **cdr**(**cons**(t, u)) to G (if they do not already exist) and merge them with the vertices t and u. Now Ψ_G satisfies the equality conjuncts of C (including the conjuncts **atom**(t), which are abbreviations for equalities involving **true**); if Ψ_G fails to satisfy some disequality conjunct of G (note that conjuncts $\neg$**atom**(t) are disequalities involving **true**), then C is unsatisfiable, hence convex. Suppose to the contrary that C is satisfiable, and that C entails a disjunction $v_1 = w_1 \vee \ldots \vee v_n = w_n$ of equalities between variables. Then $C \wedge v_1 \neq w_1 \ldots \wedge v_n \neq w_n$ is unsatisfiable. The E-graph G' obtained by adjoining vertices for the v_i and w_i to G must therefore fail to satisfy the conditions of Theorem L; inspection reveals that $\Psi_{G'}$ must violate one of the disequalities $v_i \neq w_i$; hence C entails $v_i = w_i$. Thus C is convex. ∎

Thus, a conjunction of $\mathcal{L}$-literals is convex, provided that **atom** is used only as a predicate. If occurrences of **atom** are embedded in terms or equated to terms other than **true**, conjunctions of $\mathcal{L}$-literals may be non-convex; for example, the conjunction

$$x = \mathbf{cons}(\mathbf{car}(y), \mathbf{cdr}(y)) \wedge z = \mathbf{atom}(y) \wedge w = \mathbf{true}$$

entails the split $x = y \vee z = w$. In fact, if **atom** is allowed to be embedded in terms, the satisfiability problem for $\mathcal{L}$ becomes NP-complete. To see this, note that

x is atomic or y is atomic or z is non-atomic

is equivalent to

not (x is non-atomic and y is non-atomic and z is atomic)

whose satisfiability is equivalent to that of

$$\mathbf{f}(\mathbf{cons}(\mathbf{car}(x), \mathbf{cdr}(x)), \mathbf{cons}(\mathbf{car}(y), \mathbf{cdr}(y)), \mathbf{atom}(z)) \neq \mathbf{f}(x, y, \mathbf{true}).$$

Thus, the 3-CNF satisfiability problem can be reduced to the satisfiability problem for $\mathcal{L}$, if **atom** is allowed embedded in terms. Algorithm L below is fast in the case that **atom** is used only as a predicate, and correct (but not necessarily fast) for the general case.

The satisfiability procedure for $\mathcal{L}$ uses the procedures **merge** and **forbidmerge** of section 7 to assert equalities and disequalities in an E-graph. We require that the procedure **mergepair** be modified to set the global variable *inconsistent* if the requested merge would cause the E-graph to violate axiom (4); that is, if the merge would create a pattern in which some vertex **atom**(t) is equivalent to **true**, and t is equivalent to some vertex labelled **cons**. The necessary modifications are described at the end of this section.

Algorithm L (*Satisfiability procedure for $\mathcal{L}$*). Let E be a conjunction of equalities that includes **atom**(**nil**) $=$ **true**, and let D be a conjunction of disequalities; this algorithm determines whether $E \wedge D$ is satisfiable in the theory $\mathcal{L}$. Let G be the E-graph associated with the identity relation on the set of terms appearing in E and D, together with the term **true**; let $a(1), a(2), \ldots, a(n)$ be the list of those vertices of G labelled **atom**, and let S be an empty stack. Let $i = 1$. During the execution of the algorithm, the entries on S will be triples of the form (v, i', G'), where G' is an E-graph, i' is an integer between 1 and n, and v is a vertex of G' labelled **atom**.

L1. [Satisfy E and D.] For each conjunct $t = u$ of E, merge the vertex t with the vertex u. For each disequality $t \neq u$ of D, forbid the merge of t and u. If *inconsistent* is set by any of these actions, answer "unsatisfiable".

L2. [Add vertices over **cons** nodes.] For each vertex $\mathbf{cons}(t_1, t_2)$ of G, add to G the two vertices $\mathbf{car}(\mathbf{cons}(t_1, t_2))$ and $\mathbf{cdr}(\mathbf{cons}(t_1, t_2))$ (if they are not already present), and merge them with the vertices t_1 and t_2 respectively. If *inconsistent* is set by any of these actions, answer "unsatisfiable".

L3. [Guess next atom.] If $i > n$, answer "satisfiable", otherwise set $v \leftarrow a(i)$, $i \leftarrow i + 1$; v will be of the form $\mathbf{atom}(t)$. If v is equivalent to **true**, or if t is equivalent to some vertex labelled **cons**, then repeat this step. Otherwise push (t, i, G) on S and merge v with **true**. If now *inconsistent* is true, go to step L4; otherwise repeat this step.

L4. [Current context inconsistent.] Pop an entry (t, i', G') from S, set $i \leftarrow i'$, $G \leftarrow G'$, and *inconsistent* $\leftarrow$ **false**. Find or create a vertex for $\mathbf{cons}(\mathbf{car}(t), \mathbf{cdr}(t))$ and merge it with t. If now *inconsistent* is false, go to step L3. If *inconsistent* is true, then repeat this step unless S is empty, in which case answer "unsatisfiable". ∎

The correctness of the algorithm is proved by establishing that at entry to step L3 we have the following invariants:

I. The conjunction $E \wedge D$ is satisfiable if and only if either (a) there exists an interpretation Ψ that extends Ψ_G and satisfies $E \wedge D$, or (b) there exists an interpretation Ψ and an entry (t, i', G') on the stack S such that Ψ extends $\Psi_{G'}$ and $\Psi_G(\mathbf{atom}(t)) \neq \Psi_G(\mathbf{true})$.

II. If $\mathbf{atom}(t)$ is a vertex of G (of G' for some stack entry (t', i', G')), and $\mathbf{atom}(t)$ is not equivalent to **true**, and t is not equivalent to any vertex labelled **cons**, then $\mathbf{atom}(t)$ appears in $a(i)$, $a(i+1)$, ..., $a(n)$ (in $a(i')$, $a(i'+1)$, ..., $a(n)$).

III. If (t, i', G') is an entry on the stack S, then $i \geq i'$, and furthermore $i'' \geq i'$ for any entry (t', i'', G'') that is less deep in the stack than (t, i', G').

It is not difficult to prove that these invariants hold, and that they establish the correctness of the algorithm. For example, when (t, i, G) is pushed on S and v is merged with true, the invariant is preserved because in any interpretation $\mathbf{atom}(t)$ is either true or not true. That the algorithm is correct when it answers satisfiable follows from Theorem L. (Note that invariant II says that every potential violation of axiom (3) appears in the list $a(i), \ldots, a(n)$, so if $i > n$, then the graph satisfies (3). By construction it satisfies (1), (2), (5), E, and D, and since *inconsistent* is false, it also satisfies (4).)

This algorithm may take exponential time in the worst case, but if the only occurrences of the function symbol **atom** are in literals of the form $\mathbf{atom}(t) = \mathbf{true}$ or $\mathbf{atom}(t) \neq \mathbf{true}$, then the stack never has more than one entry on it, and the time for the algorithm is dominated by the time for the congruence closure computation.

Of course, the "undo" mechanism described in section 7 is used to save and restore the state of the E-graph; it would be too slow to follow the algorithm literally and copy the whole graph onto a stack. The same stack (namely *undostack*) can be used to save i and t as well as the list of operations required to restore the E-graph. The command "push (t, i, G) on S" of step L3 is implemented by calling undo(t), undo(i), and undo(*push*), thus pushing t and i on the undo stack immediately beneath the conventional mark *push*. The code for pop is modified to restore t and i to the values on the stack after it gets to the conventional mark. We also reset *inconsistent* in pop; so the effect of the call pop() is exactly that of the first sentence of step L4.

It remains to describe the modifications necessary to make merge detect violations of (4). (Readers who are uninterested in implementation details may wish to skip the rest of the section.) These require seven bits in each E-node; the seven one-bit fields are the *hasatom* field, which is set in the root of each equivalence class containing a vertex labelled **atom**; the *hascons* field, set in the root of each equivalence class containing a vertex labelled **cons**; the *hastrue* field, set in the root of the equivalence class containing the vertex **true**; the *atomic* field, set in the root of each equivalence class that has been asserted atomic, in the sense that it has a predecessor labelled **atom** that is equivalent to **true**; and the *resetatomic*, *resethascons*, and *resethasatom* fields, which are included to facilitate "undoing" operations on the data structure.

After calling **union**(u, v) (thereby merging the equivalence classes of u and v and making v the new root), the new version of **merge** updates the *atomic* and *hascons* fields of v by ORing in the values of the corresponding fields of u. If this results in both fields being set, the procedure sets *inconsistent* and exits. Also, if either field of v was changed from **false** to **true** by this merge, a one-bit *reset* field in v is set, so that when this merge is "undone", the appropriate field of v can be cleared. Here is the fragment of code to be added:

```
$[\![ atomic(u) \wedge \neg atomic(v) \Rightarrow (atomic(v), resetatomic(u)) \leftarrow (\mathbf{true}, \mathbf{true}) ]\!]$;
$[\![ hascons(u) \wedge \neg hascons(v) \Rightarrow (hascons(v), resethascons(u)) \leftarrow (\mathbf{true}, \mathbf{true}) ]\!]$;
$hascons(v) \wedge atomic(v) \Rightarrow inconsistent \leftarrow \mathbf{true}$;
```

To undo these operations, we add to the procedure **undomerge** the following fragment, after it has undone the call to **union** (or at any convenient place):

```
$[\![ resetatomic(u) \Rightarrow (atomic(v), resetatomic(u)) \leftarrow (\mathbf{false}, \mathbf{false}) ]\!]$;
$[\![ resethascons(u) \Rightarrow (hascons(v), resethascons(u)) \leftarrow (\mathbf{false}, \mathbf{false}) ]\!]$;
```

It is also necessary, whenever an equivalence class is merged with **true**, to find each vertex in the class of the form **atom**(t) and set the *atomic* bit in the root node for t. By keeping, for each equivalence class, a circular list of the vertices labeled **atom** in that class, this can be done efficiently; here we suggest keeping one bit to say if there are *any* vertices labeled **atom**, and traversing the class to find them if necessary. This adds a factor of n to the worst case of the algorithm, but is likely to be satisfactory in practice, since the equivalence class of a vertex **atom**(t) that has not yet been merged with **true** is likely to be small (in fact it is likely to be a singleton). Here is the required code, which should be added to the procedure **merge** next to the previous addition:

```
$[\![ hasatom(u) \wedge hastrue(v) \Rightarrow \mathbf{findatoms}(u, u)$;
   $hasatom(v) \wedge hastrue(u) \Rightarrow \mathbf{findatoms}(v, v) ]\!]$;
$inconsistent \Rightarrow [\![\,]\!]$;
$[\![ hastrue(u) \Rightarrow hastrue(v) \leftarrow \mathbf{true} ]\!]$;
$[\![ hasatom(u) \wedge \neg hasatom(v) \Rightarrow (hasatom(v), resetatom(u)) \leftarrow (\mathbf{true}, \mathbf{true}) ]\!]$;
```

where the procedure **findatoms** is defined by:

```
Procedure $\mathbf{findatoms}(p, endp) \equiv$
  $[\![\, [\![ ecar(p) = atomnode \Rightarrow$
      with $y = eroot(ecar(ecdr(p)))$ do
        $[\![ atomic(y) \Rightarrow [\![\,]\!]$;
          $hascons(y) \Rightarrow inconsistent \leftarrow \mathbf{true}$;
          $atomic(y) \leftarrow \mathbf{true}$;
          $[\![ \mathbf{undo}(y);\ \mathbf{undo}(atomic) ]\!] ]\!] ]\!]$;
    $inconsistent \vee eqlink(p) = endp \Rightarrow [\![\,]\!]$;
    $\mathbf{findatoms}(eqlink(p), endp) ]\!]$.
```

(We assume that *atomnode* is the E-node representing the function symbol **atom**.)

It is easy to undo these operations. The pop procedure must recognize a new token (*atomic*), which signifies that the next object on the stack is a pointer to an E-node whose *atomic* field should be cleared. The *hasatom* field is restored just as the *hascons* and *atomic* fields are. The *hastrue* field is easier to restore: since **true** can be a member of only one equivalence class at once, the following fragment, added to **undomerge**, suffices:

$$[\![\, hastrue(u) \Rightarrow hastrue(v) \leftarrow \mathbf{false} \,]\!].$$

Instead of explicitly representing the conjunct **atom(nil)** = **true**, as implied by Algorithm L, the procedure that converts terms into E-nodes just sets the *atomic* bit in the vertex node representing **nil()**.

10. The satisfiability procedure for $\mathcal{A}$

The satisfiability problem for $\mathcal{A}$ was first solved by Kaplan [28]. Downey and Sethi [21] considered the special case of determining whether two $\mathcal{A}$-terms are equivalent, and showed that this problem was NP-complete. Hence the general problem is NP-hard. A careful analysis of Kaplan's algorithm would probably show that the general problem is in NP, but the algorithm is complicated, and apparently no such analysis has been performed. The algorithm below is simpler, and shows that the problem is in NP.

Recall that the theory $\mathcal{A}$ has the two axioms:

$$(\forall a\, i\, e)\; a^{(e)}_{i}(i) = e \tag{1}$$
$$(\forall a\, i\, j\, e)\; i \neq j \supset a^{(e)}_{i}(j) = a(j) \tag{2}$$

where the notation abbreviates terms involving the function symbols **store** and **select** as described in section 5.

The satisfiability procedure for $\mathcal{A}$ rests on the following theorem:

Theorem A. *Let E be a conjunction of equalities, D a conjunction of disequalities, G an E-graph such that Ψ_G satisfies E, D, (1), and (2), and suppose that:*

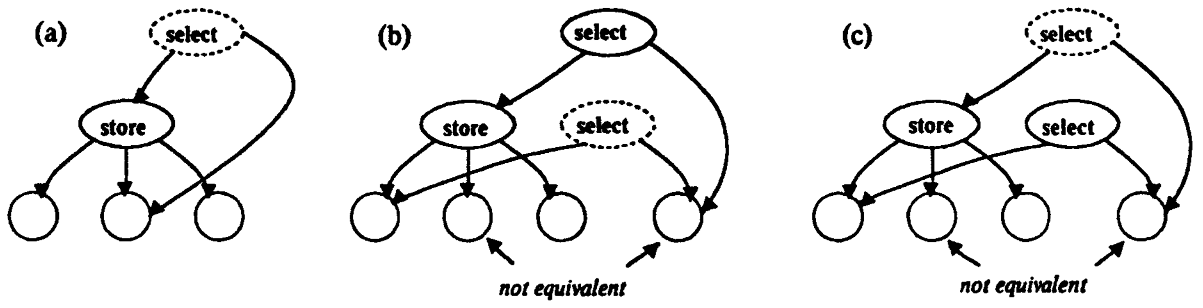
(3) *G represents $a^{(e)}_{i}(i)$ whenever G represents $a^{(e)}_{i}$.*

(4) *G represents $a(j)$ whenever G represents $a^{(e)}_{i}(j)$ and $\Psi_G(i)$ is different than $\Psi_G(j)$.*

(5) *G represents $a^{(e)}_{i}(j)$ whenever G represents both $a^{(e)}_{i}$ and $a(j)$, and $\Psi_G(i)$ is different than $\Psi_G(j)$.*

Then $E \wedge D$ is satisfiable in the theory $\mathcal{A}$.

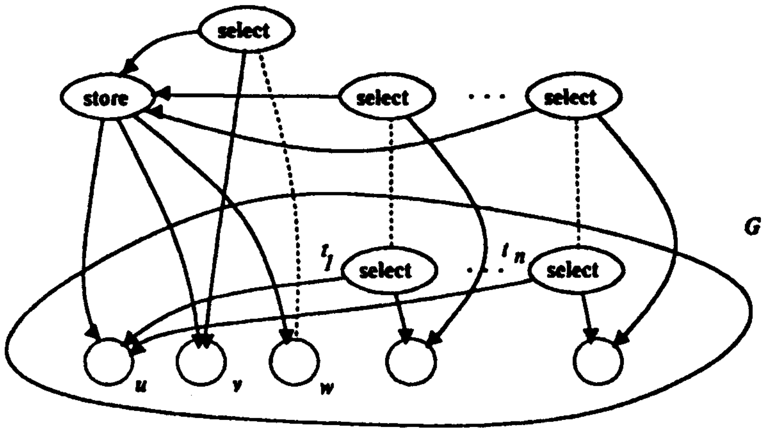
Proof. Conditions (3), (4) and (5) are summarized by the following diagram, in which the dotted nodes are required to exist whenever the solid nodes exist:



Because Ψ_G is required to satisfy (1) and (2), the nodes labelled **select** will be equivalent in (b) and (c); and in (a), the **select** node will be equivalent to the third successor of the **store** node. The diagram is imprecise in that it ignores the equivalence relation; an instance of the solid part of (b) need not be a **select** node whose first successor is a **store** node; it is enough that its first successor be equivalent to some **store** node.

The proof of the theorem is similar to the proof of Theorem L. Let $\mathcal{G}$ be the class of E-graphs that satisfy the conditions of the theorem. We show that for any $G \in \mathcal{G}$, for any function symbol f, and for any n vertices $v_1, \ldots, v_n$ of G, where n is the arity of f, there exists an E-graph $G' \in \mathcal{G}$ such that $\Psi_{G'}$ extends Ψ_G and G' contains a representative for $f(v_1, \ldots, v_n)$. The theorem then follows from Lemma X. The proof of extension is trivial for all function symbols except **store** and **select**.

To add a node labelled **store**. Let u, v, and w be vertices such that G does not represent the term **store**(u, v, w). Let $\{t_1, \ldots, t_n\}$ be the set of vertices of G that are labelled **select**, have first successors that are equivalent to u, and have second successors that are not equivalent to v. To construct G', add to G the vertex **store**(u, v, w); add the vertex **select**(**store**$(u, v, w), v)$) and merge it with w; and for each t_i, add the vertex **select**(**store**$(u, v, w), t_i[2]$) and merge it with t_i. Here is a diagram showing the additions to the graph:



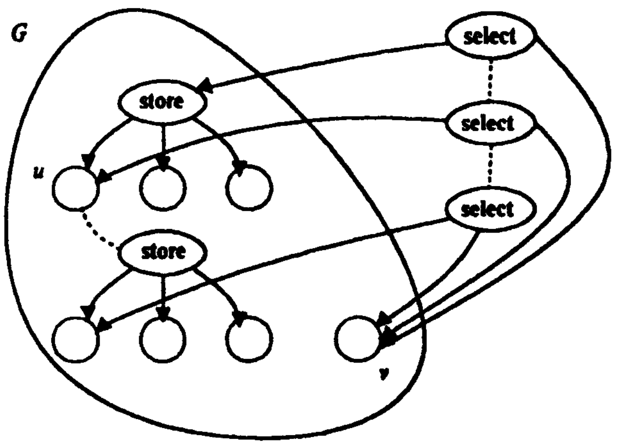
Inspection shows that $\Psi_{G'}$ satisfies (1) through (5) and extends Ψ_G.

To add a node labelled **select**. Let u and v be two vertices of G such that G does not represent the term **select**(u, v). We will add the vertex **select**(u, v) and additional vertices labelled **select** in order to maintain (1) through (5).

Let U be the smallest set of vertices of G such that (a) $u \in U$; (b) if $x \in U$, x is labelled **store**, $x[2]$ is not equivalent to v, then $x[1] \in U$; (c) if $x[1] \in U$, x is labelled **store**, and $x[2]$ is not equivalent to v, then $x \in U$; (d) if $x \in U$ and y is equivalent to x, then $y \in U$.

For any $u' \in U$, G has no vertex representing **select**(u', v); for if such a vertex existed, then by repeated application of (4) and (5) we would find a vertex for **select**(u, v), which we are assuming does not exist.

Let G' be obtained from G by adding a vertex for **select**(u', v), for each $u' \in U$, and merging all the resulting vertices, as shown here:



By the remark above, all the vertices merged are new ones, so $\Psi_{G'}$ is an extension of Ψ_G. Inspection shows that $\Psi_{G'}$ satisfies (1) to (5).

This completes the proof of Theorem A. ▮

The satisfiability procedure for $\mathcal{A}$ uses a modified version of the procedure merge of section 7. The new merge detects any patterns that appear in the E-graph that may require a case-split, and enters the patterns in a global array b, which is regarded as a potentially infinite list of pairs of vertices; the current value of which is $b(l)$, $b(l+1)$, ..., $b(r)$. To add a pair (u, v) to b, merge sets $r \leftarrow r+1$; $b(r) \leftarrow (u, v)$. We assume that whenever two equivalence classes Q_1 and Q_2 are merged, the procedure **merge**

- adds to the list b all pairs (u, v) such that $\lambda(u) =$ **store**, $\lambda(v) =$ **select**, and either $u \in Q_1$ and $v[1] \in Q_2$, or $u \in Q_2$ and $v[1] \in Q_1$.
- adds to the list b all pairs (u, v) such that $\lambda(u) =$ **store**, $\lambda(v) =$ **select**, and either $u[1] \in Q_1$ and $v[1] \in Q_2$, or $u[1] \in Q_2$ and $v[1] \in Q_1$.

In other words, b keeps track of potential violations of (4) and (5). The required modifications to merge are similar to those described in the last section, and will not be described in detail. A double loop can be used to find all the pairs. By using six bits in each enode to keep track of which equivalence classes contain vertices labelled **store** or vertices that are first successors of vertices labelled **store** or **select**, the double loop can be avoided whenever it would find no pairs.

Algorithm A_(*The Satisfiability Procedure for* $\mathcal{A}$). Given a conjunction of equalities E and a conjunction of disequalities D, this algorithm determines whether $E \wedge D$ is satisfiable in the theory $\mathcal{A}$. The algorithm uses an E-graph G, which is initialized to represent the identity relation on the set of terms appearing in E and D; the global array b described above, together with its indices l and r; and a stack S, initially empty, which during the execution of the algorithm will contain tuples of the form (u, v, l', r', G'), where u and v are vertices of G', G' is an E-graph, and l' and r' are integers. Initially l is 1 and $b(1)$, ..., $b(r)$ is the list of pairs (u, v) of vertices of G such that $\lambda(u) =$ store, $\lambda(v) =$ **select**, and either $u = v[1]$ or $u[1] = v[1]$.

A1. [Satisfy E and D.] For each conjunct $t = u$ of E, merge the vertex t with the vertex u. For each disequality $t \neq u$ of D, forbid the merge of t and u. If *inconsistent* is set by any of these actions, answer "`unsatisfiable`".

A2. [Add select vertices.] For each vertex store(u, v, w) of G, add the vertex select(store$(u, v, w), v$) and merge it with w. If *inconsistent* is set by any of these actions, answer "unsatisfiable".

A3. [Perform a split.] If $l > r$, answer "satisfiable," else let (u, v) be $b(l)$ and set $l \leftarrow l + 1$. If $v[2]$ is equivalent to $u[2]$, repeat this step; else push (u, v, l, r, G) on S and merge $u[2]$ with $v[2]$. If now *inconsistent* is set, go to step A4, else repeat this step.

A4. [Try next case.] Pop an entry (u, v, l', r', G') from S and set $l \leftarrow l'$, $r \leftarrow r'$, $G \leftarrow G'$, and *unsatisfiable* $\leftarrow$ false. If u is equivalent to $v[1]$, let w be a vertex that represents select$(u[1], v[2])$, else let w be a vertex to represent select$(u, v[2])$. In either case, create w if it is not already present. Merge w with v. If now *inconsistent* is false, go to step A3, else if S is empty, answer "unsatisfiable," else repeat this step. ▮

The correctness of this algorithm follows from Theorem A in essentially the same way that the correctness of Algorithm L follows from Theorem L. Note that just before restoring $r \leftarrow r'$ in step A4, the entries $b(r' + 1)$, $b(r' + 2)$, ... $b(r)$ are irrelevant. If b is represented as a linked list, which is a reasonable choice, then the storage for these entries should be returned to the avaliable list at that point.

Each vertex labelled select added in step A4 has for its successors two vertices in the original graph; this is easy to prove by induction and inspection of (b) and (c) above. It follows from this that in the worst case, the algorithm requires $O(n^2)$ space and $2^{O(n^2)}$ time, where n is the length of the original conjunction in characters.

It is not clear whether this algorithm takes $2^{\Theta(n^2)}$ time. To see what can happen, consider applying it to the conjunction

$$a\,\overset{(x)}{\underset{i_1}{\cdot}}\,\overset{(x)}{\underset{i_2}{\cdot}}\cdots\overset{(x)}{\underset{i_n}{\cdot}} = a\,\overset{(x)}{\underset{j_1}{\cdot}}\,\overset{(x)}{\underset{j_2}{\cdot}}\cdots\overset{(x)}{\underset{j_{n-1}}{\cdot}} \;\wedge\; a(i_1) \neq x \;\wedge\; a(i_2) \neq x \;\wedge\; \ldots \;\wedge\; a(i_n) \neq x.$$

The array on the left of the equality differs from a at i_1, i_2, ..., i_n. But these n indices cannot all be distinct, since the array on the right of the equality differs from a at no more than $n - 1$ places. Thus the conjunction entails

$$\bigvee_{j \neq k} i_j = i_k.$$

Thus the theory $\mathcal{A}$ is badly non-convex; a conjunction of length n may cause a split with $\Theta(n^2)$ branches. However, Algorithm A will stop at the first satisfiable branch, so it will discover the satisfiability of the above conjunction rapidly. It would cost $2^{\Theta(n^2)}$ time if each branch were unsatisfiable; for example if the conjuncts $b(i_j) \neq b(i_k)$ were included for all j and k with $1 \leq j < k \leq n$. But then the length of the conjunction becomes n^2.

It would be useful to have a satisfiability procedure for the theory obtained by adding to $\mathcal{A}$ the additional axiom

$$(\forall\, a\, b)\; a \neq b \supset (\exists\, x)\, a(x) \neq b(x).$$

11. An algorithm for finding disjunctive normal forms

Before going on to the satisfiability procedure for $\mathcal{R}$, which uses data structures that are very different from the E-graph, we will in this section describe an algorithm that extends the material in the last five sections in three ways: The algorithm combines the satisfiability procedures for $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{A}$; it determines a "simplest form" for its input instead of just determining if its input is satisfiable, and its input is an arbitrary formula instead of a conjunction of literals.

It is clear that to combine the satisfiability procedures for $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{A}$ all we have to do is perform the operations of Algorithms L and A together in one algorithm; it is not necessary to propagate equalities from one procedure to the other, because both procedures represent equalities in the E-graph, so an equality discovered by one procedure will automatically be available to the other.

Since the algorithms for $\mathcal{L}$ and $\mathcal{A}$ have already introduced case splitting, it will not be difficult to handle explicit disjunctions in the input formula: such disjunctions can be handled by entries on the same stack used to handle the splits demanded by $\mathcal{L}$ and $\mathcal{A}$.

The "simple form" produced by Algorithm D of this section is a version of "disjunctive normal form." A disjunctive normal form for a formula F is a formula G that is a disjunction of conjunctions of literals, such that F is equivalent to G and each disjunct of G is satisfiable. Thus F is satisfiable if and only if G is not the empty disjunction; the disjuncts of G describe the satisfying contexts of F. To find such a formula G requires no substantial change in approach: both Algorithm L and Algorithm A essentially search for a satisfying context for the input; the new algorithm searches in the same way, but instead of halting with the answer "satisfiable" when it finds a satisfying context, it outputs the conjunction representing the context and continues searching for other satisfying contexts.

For example, consider the formula

$$f(f(f(f(f(f(x)))))) = x \land (f(f(f(f(x)))) = x \lor f(f(f(x))) = x).$$

By case-splitting in the obvious way, the algorithm finds two E-graphs that satisfy the formula; the first E-graph is constructed by merging $f^6(x)$ with x and $f^4(x)$ with x; the second is constructed by merging $f^6(x)$ with x and $f^3(x)$ with x. By using a method described at the end of this section for simplifying conjunctions of equalities, it is possible to simplify these two contexts and compute the disjunctive normal form $f(f(x)) = x \lor f(f(f(x))) = x$.

We assume the existence of a procedure **add**(t, G) that adds a representative for the term t to the E-graph G, if one is not already present, and that has the following properties: The addition is "undoable", in the sense that calling **pop** will remove all vertices added by **add**. Also, for each vertex **cons**(u, v) created, **add** also creates vertices **car**(**cons**(u, v)) and **cdr**(**cons**(u, v)), and merges them with u and v respectively. Similarly, for each vertex **store**(u, v, w) created, **add** also creates the vertex **select**(**store**$(u, v, w), v)$ and merges it with v. (Notice that neither of these merges can create any inconsistency, since new vertices are just being added to old equivalence classes.) Finally, we assume that each vertex labelled **atom** created by **add** is added to the global list a (in the sense of section 9), and that for each vertex v labelled **select** created by **add**, a pair (v, w) is added to the global list b (in the sense of section 10) for each w labelled **store** that is either equivalent to $v[1]$, or is such that $w[1]$ is equivalent to $v[1]$.

Algorithm D (*Find Disjunctive Normal Form*). Given a conjunction $F_1 \land \ldots \land F_n$ of quantifier-free formulas, this algorithm finds all essentially different E-graphs G such that Ψ_G satisfies the conjunction. By using methods described at the end of the section, the graphs discovered by the algorithm can be turned back into simple conjunctions of literals; thus a disjunctive normal form can be constructed for the original conjunction. We assume that the only boolean connectives appearing in the input are $\land$ and $\neg$; this entails no loss of generality, since the other connectives can be defined in terms of these. The algorithm uses the queue $a(al), \ldots, a(ar)$ that is identical to the queue $a(i), \ldots, a(n)$ of Algorithm E, the queue $b(bl), \ldots, b(br)$ that is identical to the queue $b(l)$, $\ldots, b(r)$ of Algorithm A, and the queue $c(cl), \ldots, c(cr)$, whose entries are pairs $(G, flag)$ where G is a formula (actually a subformula of the input conjunction), and $flag$ is either **true** or **false**. Such an entry represents an assumption that G has the truth-value of $flag$. Initially, $cl = 1$, $cr = n$, and $c(i) = (F_i, \textbf{true})$ for $1 \leq i \leq n$. The algorithm also uses a stack S, whose entries are tuples of the form $(X, al', ar', bl', br', cl', cr', G')$, where X is either one of the conventional marks *amark* or *bmark*, or else is a formula; G' is an E-graph; and the other six components are integers. Finally, the algorithm uses the global E-graph G, which initially is the empty E-graph. The queues a and b are initially empty; that is, $al = bl = 1$, $ar = br = 0$.

D1. [Get next conjunct.] If $cl > cr$, go to D3. Otherwise, set $(G, flag) \leftarrow c(cl)$, $cl \leftarrow cl + 1$, and go on to D2.

D2. [Process conjunct.] If G is of the form $\neg G'$, set $G \leftarrow G'$, $flag \leftarrow \neg flag$, and repeat this step. If G is of the form $G_1 \wedge G_2$ and $flag = \textbf{true}$, set $cr \leftarrow cr + 1$, $c(cr) \leftarrow (G_2, \textbf{true})$, $G \leftarrow G_1$, and repeat this step. If G is of the form $G_1 \wedge G_2$ and $flag = \textbf{false}$, push the entry $(G_2, al, ar, bl, br, cl, cr, G)$ on S, set $G \leftarrow G_1$, and repeat this step. Otherwise G will be an atomic formula $u = v$; call **add**(u, G), **add**(v, G). If $flag = \textbf{true}$, call **merge**(u, v), else call **forbidmerge**(u, v). If *inconsistent* is set, go to D5; else go to D1.

D3. [Atom split.] If $al > ar$, go to D4, else set $v \leftarrow a(al)$, $al \leftarrow al + 1$; v will be of the form **atom**(t). If v is equivalent to **true**, or if t is equivalent to some vertex labelled **cons**, then repeat this step. Otherwise, push $(amark, al \dot{-} 1, ar, bl, br, cl, cr, G)$ on S and merge v with **true**. If now *inconsistent* is true, go to D5, else repeat this step.

D4. [Store split.] If $bl > br$, output the current E-graph and go to D5. Otherwise set $(u, v) \leftarrow b(bl)$ and $bl \leftarrow bl + 1$. If $v[2]$ is equivalent to $u[2]$, repeat this step; else push $(bmark, al, ar, bl - 1, br, cl, cr, G)$ on S and merge $v[2]$ with $u[2]$. If now *inconsistent* is set, go to D5, else repeat this step.

D5. [Backtrack.] If S is empty, the algorithm halts. Otherwise, pop an entry $(X, al', ar', bl', br', cl', cr', G')$ from S and set $al \leftarrow al'$, $ar \leftarrow ar'$, $bl \leftarrow bl'$, $br \leftarrow br'$, $cl \leftarrow cl'$, $cr \leftarrow cr'$, and $G \leftarrow G'$. If X is a formula, then set $G \leftarrow X$, $flag \leftarrow \textbf{false}$, and go to step D2. If $X = amark$, then $a(al)$ will be of the form **atom**(t); call **add**(**cons**(**car**(t), **cdr**(t))), merge **cons**(**car**(t), **cdr**(t)) with t, and then if *inconsistent* is true, repeat this step, else set $al \leftarrow al + 1$ and go to D3. Finally, if $X = bmark$, $b(bl)$ will be a pair (u, v). If u is equivalent to $v[1]$, call **add**(**select**$(u[1], v[2])$) and merge **select**$(u[1], v[2])$ with v; else call **add**(**select**$(u, v[2])$) and merge **select**$(u, v[2])$ with v. In either case, if *inconsistent* is set, repeat this step, else set $bl \leftarrow bl + 1$ and go to D4. ∎

The proof that each E-graph output by the algorithm satisfies the input conjunction is straightforward. It is not difficult to formalize the notion of a *minimal* satisfying E-graph as one whose vertex set and equivalence relation are both as small as possible, and to show that in fact the algorithm finds all minimal satisfying E-graphs. (Of course, the algorithm may find the same graph more than once; for example, if the input conjunction is $\mathbf{P} \vee \mathbf{P}$.)

It is straightforward to incorporate other satisfiability procedures into the algorithm above. The algorithm provides a general framework for theorem-proving with resettable data structures. Section 13 describes the modifications necessary to incorporate the satisfiability procedure for $\mathcal{R}$.

When this algorithm is used in program verification, the conjunction $F_1 \wedge \ldots \wedge F_n$ represents all the assumptions made by the symbolic evaluator on its path to some program point, together with the negation of some invariant that is to be proved to hold at that point. (The stack S contains additional entries that allow the symbolic evaluator to retrace its steps and try other paths through the program.) Thus the outputting of the E-graph in step D4 corresponds to a failure to verify some invariant. By associating with the conjuncts F_i the entities to which they correspond (namely entry assumptions, program branch points, or negated assumptions), the verifier can output a coherent account of where it failed and what potential counterexample context is troubling it. (Of course, one or more steps can be added between D4 and D5 that bring more powerful theorem-proving methods to bear, such as matching or automatic induction.)

It is very important in practice to put off case splits when possible. For example, suppose that the input conjunction is

$$(a \vee b) \wedge (c \vee d) \wedge (e \vee f) \wedge g \wedge \neg g.$$

The algorithm as described would consider eight cases, each of which would of course turn out to be unsatisfiable. It is easy to change the algorithm so that it asserts all literals before it processes any disjunctions: step D2 must save disjunctions on another global queue instead of doing the split, and an additional step can perform the saved splits after all atomic conjuncts have been asserted.

It is possible to work even harder to avoid case-splits, for example, consider the conjunction

$$(a_1 \vee b_1) \wedge \ldots \wedge (a_n \vee b_n) \wedge (c \supset d) \wedge (d \supset e) \wedge (e \supset \neg c) \wedge c. \qquad (1)$$

After asserting the lone literal, c, it is necessary to choose one of $n+3$ case splits. However, one of these case splits (namely $c \supset d$, which is the case split $\neg c \vee d$), is illusionary in that one of its branches ($\neg c$) is immediately unsatisfiable. If this is detected, a new literal is discovered that can be asserted, namely d, then similarly e is discovered to be true, and then a contradiction is detected. This suggests the following strategy: before doing any case splits, check that every case split we are faced with is in fact "proper", in the sense that none of its branches will lead to immediate inconsistency. This strategy can be applied to the splits required by $\mathcal{A}$ and $\mathcal{L}$ as well as those required by explicit disjunctions. Unfortunately the obvious implementation of this rule is expensive, since the only way to be sure that a branch is not manifestly inconsistent is to assert the branch and test *inconsistent*, and this test will have to be repeated whenever a new assertion is made. Thus it would take a complete pass through (1) to ferret out each new literal. The possibilities for more sophisticated implementations are unclear. (In special cases, we can do much better; for example, see [3] for a linear-time algorithm for the 2-CNF satisfiability problem that is relevant to this problem.)

We now turn to the problem of converting an E-graph into the conjunction of literals that it represents. The heart of this is the following combinatorial problem: Given an E-graph $G = (V, E, \lambda, R)$, where R is congruence-closed, find a set R' of pairs of vertices $\{(u_1, v_1), \ldots, (u_k, v_k)\}$, such that the congruence closure of R' is R, and such that k is as small as possible. We define the *height* of a vertex of an E-graph as follows: the height of a variable is zero, and the height of $f(t_1, \ldots, t_n)$ is one greater that the height of the tallest of $t_1, \ldots, t_n$. Note that the height of a vertex is independent of the equivalence relation.

Lemma M. *Let $G = (V, E, \lambda, R)$ be an E-graph such that R is congruence-closed, $Q_1, \ldots, Q_q$ the equivalence classes of R, and for $1 \le i \le q$ let $Q_{i,1}, Q_{i,2}, \ldots, Q_{i,k_i}$ be the partition of Q_i into congruence classes. (Two vertices are in the same congruence class if they are congruent.) For $1 \le i \le q$, $1 \le j \le k_i$, let $v_{i,j}$ be a vertex in $Q_{i,j}$ whose height is less than or equal to the height of any other vertex in $Q_{i,j}$. Let R' be the relation*

$$\begin{array}{l}\{(v_{1,1}, v_{1,2}), (v_{1,2}, v_{1,3}), \ldots, (v_{1,k_1-1}, v_{1,k_1}),\\ \phantom{\{}(v_{2,1}, v_{2,2}), (v_{2,2}, v_{2,3}), \ldots, (v_{2,k_2-1}, v_{2,k_2}),\\ \qquad\qquad\vdots\\ \phantom{\{}(v_{q,1}, v_{q,2}), (v_{q,2}, v_{q,3}), \ldots, (v_{q,k_q-1}, v_{q,k_q})\}.\end{array}$$

Then the congruence closure of R' is R, and no relation with fewer pairs than R' has this property.

Proof. We prove that for all n, two vertices v and w each of height less than or equal to n are equivalent in R only if they are equivalent in R', by induction on n.

Base case. If u and w each have height 0, then they are variables, and no vertex is congruent to either of them. Therefore the congruence classes containing them are singletons. If they are both in the same Q_i, it follows that $u = v_{i,j}$ and $w = v_{i,j'}$ for some j, j'; hence they are equivalent in R'.

Induction step. Let u and w be vertices whose heights do not exceed n, and which are both contained in Q_i. Let $v_{i,j}$ be congruent to u in R, and $v_{i,j'}$ congruent to w in R. Since the height of $v_{i,j}$ does not exceed the height of u, the successors of $v_{i,j}$ have height strictly less than n, as do the successors of u. From the induction hypothesis and the fact that u is congruent to $v_{i,j}$ in R, it follows that corresponding successors of $v_{i,j}$ and u are equivalent in the congruence closure of R'; hence u is equivalent to $v_{i,j}$ in the congruence closure of R'. Similarly w is equivalent to $v_{i,j'}$ in the congruence closure of R'. Since $v_{i,j}$ and $v_{i,j'}$ are obviously equivalent in the congruence closure of R', the induction step is complete.

It remains to prove that R' contains the minimum number of pairs. This follows immediately from the following fact: Let R'' be any equivalence relation such that if any two vertices u and v are equivalent in R'', they are equivalent in R, and if any two vertices u and v are congruent in R, they are equivalent in R''. (Such a relation is obtained from R by breaking up the equivalence classes of R into blocks that preserve the congruence classes of R.) Then R'' is closed under congruences. This is trivial: if u and v are congruent under R'', their corresponding successors are equivalent under R'', hence under R, hence u and v are congruent under R, hence equivalent under R''. It follows from this fact that any set of pairs whose congruence closure is R must include a "spanning tree" connecting all the congruence classes in each equivalence class, hence must contain as many pairs as R'. ▮

(It makes sense to talk above the congruence closure of a relation on a labelled *cyclic* graph, and in fact the algorithm of section 7 works for this case. But Lemma M is false for cyclic graphs; for example, if G consists of just the two vertices u and v, both with outdegree 1 and with the same label, and if $u[1] = v$, $v[1] = u$, and u and v are equivalent, then the single equivalence class consists of a single congruence class, so Lemma M would say that no merges were necessary to produce this graph, which is false since the identity relation is congruence-closed on the graph. It would be interesting to find a counterpart for Lemma M for graphs that were not acyclic.)

Having found a minimal set of pairs whose congruence-closure is the given relation, it is necessary to choose a terms representing the vertices in order to produce a conjunction of equalities, and it is natural to want to choose the simplest such term. A straightforward dynamic programming algorithm will, in one bottom-up pass, compute a term t_Q for each equivalence class Q, such that Q contains a representative for t_Q but not for any smaller term. Having performed this algorithm, we can convert a pair $(v_{i,j}, v_{i,k})$ of the relation R' of Lemma M into an equality conjunct as follows: if $v_{i,j}$ is a variable, let $t = v_{i,j}$; if $v_{i,j}$ is of the form $f(t_1, \ldots, t_n)$, let $Q(i)$ be the equivalence class of t_i, for $1 \leq i \leq n$, and let t be the term $f(t_{Q(1)}, \ldots, t_{Q(n)})$. Similarly form u from $v_{i,k}$. Then $t = u$ may be taken as the equality conjunct corresponding to the pair $(v_{i,j}, v_{i,k})$. In this way a minimal conjunction of equalities between minimal-sized terms is obtained that can be used in the output of the simplifier.

Instead of performing a dynamic programming computation, it is possible (and simple) to incrementally keep track of the minimum-height vertex in each equivalence class, using two extra fields in each E-node (one to hold the height of the node, the other to point from the root of a class to the shortest node in the class). This method is not guaranteed to produce minimum-sized conjuncts, however, and it may actually be slower than the first method, since it substitutes work in **mergepair** and **undomerge** for work performed only when a satisfying context is found.

Of course, it is necessary to find the disequalities that belong in the conjunction, too. A pass through the list $c(1)$, $c(2)$, $\ldots$, $c(cr)$ will find all atomic formulas that have been asserted **false**, and it is straightforward to prune them to a minimal set, since a disequality is redundant only if it is equivalent, given the current equivalence relation, to some other disequality.

Finally, we must deal with the fact that some of the merges represent instances of the axioms of $\mathcal{L}$ and $\mathcal{A}$, and should not produce equality conjuncts in the simplified output. For example, suppose some equivalence class Q contains congruence classes C_1, C_2, C_3, and C_4. Ordinarily we would include three equalities in the output, to link up the four congruence classes, but if, say, C_1 contains a vertex u and C_2 contains a vertex v such that u and v were merged to satisfy an axiom of $\mathcal{A}$ or $\mathcal{L}$, (in the sense that $u = \mathrm{car}(\mathrm{cons}(v, w))$ or $u = \mathrm{select}(\mathrm{store}(a, i, v), i)$ or the like) then including an equality between u and v in the output would produce a conjunction one of whose equality conjuncts is a consequence of the axioms of $\mathcal{A}$ and $\mathcal{L}$. Therefore, in this case, C_1 and C_2 may be treated as a single congruence class, and only two equalities need be included to link up Q. There are only three kinds of merges to worry about, namely when a vertex labelled **car**, **cdr**, or **select** is constructed and merged with a vertex below it; it is not difficult to keep track of these merges and prevent them from appearing in the output.

As mentioned above, Algorithm D may find the same satisfying context more than once. More generally, it may find two contexts, one of which subsumes the other, and it is expensive to check for this. It seems to be very difficult to compute or even define any workable cannonical form for formulas with equality. For example, the formula

$$(i = 0 \wedge \mathrm{p}(0)) \vee \mathrm{p}(i)$$

is equivalent to $\mathrm{p}(i)$, but it is hard to imagine how to perform the simplification mechanically.

12. The satisfiability procedure for $\mathcal{R}$

The satisfiability problem for $\mathcal{R}$ is essentially the problem of verifying that one linear inequality is a consequence of several other linear inequalities—for example, that $x \geq 0$ is a consequence of $x \geq z$ and $x \geq -z$, or that $y < 2x + 1$ is a consequence of $y < x$ and $y > -x$. This is equivalent to the general linear programming problem, but (at least according to the conventional view) the instances of the problem arising in verification work differ significantly from those arising in operations research, so verification researchers have designed many new methods, and modified traditional linear programming methods, to handle the cases they encounter in practice. For example, Pratt [48] reports that most of the inequalities encountered in verification work are of the form $x \leq y + k$ (where x and y are variables and k is a constant); the solution of sets of inequalities of this form is equivalent to the problem of detecting negative cycles in weighted directed graphs. Shostak [54] generalizes Pratt's method to allow inequalities of the form $ax + by \leq k$ (or $ax + by < k$) where x and y are variables and a, b, and k are numerals. The author [45] and Aspvall and Shiloach [4] also describe algorithms for the case considered by Shostak. Some theorem provers have used the Fourier-Motzkin elimination method for the general case, some have used the Sup-Inf method of Bledsoe (see Bledsoe [6] and Shostak [55]), and some have used heuristic methods (see Deutsch [17] and King [30]). None of these methods has been entirely satisfisfactory.

(The corresponding satisfiability problem for non-linear inequalities is even harder; it boils down to determining whether a multivariate polynomial has a real root. The problem is obviously NP-hard; presumably it it NP-complete, but no easy proof suggests itself. The lack of results on this satisfiability problem is surprising considering the amount of attension received by the corresponding decision problem–see for example [12, 13, 26, 50, 62]. Practical methods are as scarce as theoretical results. Facinating difficultites also arise when integer variables are allowed; though the integer linear inequalities that arise in program verification can be handled quite satisfactorily by heuristics. The satisfiability problem for non-linear integer inequalities is unsolvable [41].)

This section describes a satisfiability program for $\mathcal{R}$ that is based on the simplex algorithm [16]. The program is *incremental*; that is, it accepts linear inequalities one by one, maintains a representation for the conjunction of those it has received, and sets a flag whenever the conjunction becomes unsatisfiable. It also detects the equalities between variables that are consequences of the inequalities it has received; for example, if it were given $x \leq y$ and $y \leq x$, it would "propagate" the equality $x = y$. Finally, the program is *resettable*; that is, the last inequality "asserted" to the program can be removed, and the program will restore the data structure to the state it was in before the last inequality was asserted. It is as though there is a stack where inequalities can be pushed or popped; the program represents the conjunction of the inequalities in the stack.

Almost all descriptions of the simplex algorithm emphasize the optimization problem instead of the satisfiability problem, and the variables are usually restricted to be non-negative. There are standard methods for reducing the satisfiability problem for $\mathcal{R}$ to the canonical linear programming optimization problem with non-negative variables, but if these methods are applied blindly they increase the size of the problem by unnecessarily introducing many extra variables. Thus the treatment below differs from the standard treatment of the simplex algorithm in that we are interested in the satisfiability problem instead of the optimization problem, our variables are not restricted to be non-negative, and the program must be incremental, resettable, and must propagate equalities. Any of these differences taken by itself is minor, but their combined effect is sufficiently significant that it seems unfair to refer the reader to reference works on linear programming that describe the algorithm from a very different point of view. Therefore, the algorithm will be described "from scratch", as it appears in our application.

We will assume that all the inequalities are of the form $L \geq 0$, where L is an affine combination of variables. This assumption can be made without loss of generality, since the equality $L = M$ can be represented as the conjunction of the two inequalities $L - M \geq 0$ and $M - L \geq 0$; the strict inequality $L > M$ can be represented as the conjunction of $L - M \geq 0$ and $L - M \neq 0$; and disequalities like $L \neq M$ can be represented in the E-graph. (The actual program does not use these reductions; it accepts equalities and strict inequalities directly. But all the important ideas in the program are illustrated by inequalities of the form $L \geq 0$. The handling of the other cases will be sketched at the end of this section.)

The data structure used by the program will be called a *tableau*. A tableau consists of two one-dimensional arrays of variables, $x(1)$, $x(2)$, ... and $y(1)$, $y(2)$, ...; one one-dimensional array of rational numbers, $c(1)$, $c(2)$, ...; one two-dimensional array of rational numbers $a(1,1)$, $a(1,2)$, ..., $a(2,1)$, ...; one set of variables, R; and three integers n, m, and *dcol*. The arrays are all regarded as potentially infinite; but at any moment $x(j)$, $y(i)$, $a(i,j)$, and $c(i)$ are relevant only for $1 \leq i \leq n$, $1 \leq j \leq m$. Furthermore, the variables in R will always be a subset of $\{x(1), \ldots, x(m), y(1), \ldots, y(n)\}$. The variables in R will be called *restricted*; the other variables *unrestricted*. We will use subscripts to specify components of the arrays; we write a_{ij} for $a(i,j)$. (In some places below this may be confusing; remember that the expression "x_i" does not denote a fixed variable but an element of a dynamically varying array of variables.)

The tableau will be displayed in the following format:

$$\begin{array}{c|cccc} & x_1 & \cdots & x_m & \\ \hline y_1 & a_{11} & \cdots & a_{1m} & c_1 \\ \vdots & & \vdots & & \vdots \\ y_n & a_{n1} & \cdots & a_{nm} & c_n \end{array} \qquad (1)$$

where the variables x_i and y_i are superscripted with the sign $\geq$ if they are restricted. The value of *dcol* is always between 0 and m; to indicate its value, a small triangle will be drawn between column *dcol* and column *dcol* $+ 1$.

For example, here is a typical tableau in which *dcol* is zero:

$$\begin{array}{c|cccc} & {}^{\triangledown} m^{\geq} & n^{\geq} & x & \\ \hline y^{\geq} & 1 & -1 & 2 & 0 \\ l & -1 & -3 & 0 & -1 \end{array} \tag{2}$$

The tableau (2) represents the five constraints:

$$\begin{gathered} m \geq 0 \qquad n \geq 0 \qquad y \geq 0 \\ y = m - n + 2x \\ l = -m - 3n - 1. \end{gathered}$$

We will assume for awhile that *dcol* is zero, since the important ideas in the program are illustrated by this case. Under this assumption, the tableau (1) represents the n *row constraints*:

$$y_i = a_{i1}x_1 + \cdots + a_{im}x_m + c_i, \qquad i = 1, 2, \ldots, n$$

as well as a *sign constraint* $v \geq 0$ for each restricted variable v. The variables $x_1, \ldots, x_m$ are the *column variables*; the variables $y_1, \ldots, y_n$ are the *row variables*, x_i is the *owner* of column i, y_i the *owner* of row i. The column $c_1, \ldots, c_n$ is the *constant column*. The tableau will always have the property that the variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ are distinct.

We identify points in R^{n+m} with assignments of values to the variables of the tableau (1) by choosing some ordering of its variables. To be definite, let $(v_1, \ldots, v_{n+m})$ be the variables of (1) listed in alphabetical order; then the point $(r_1, \ldots, r_{n+m}) \in R^{n+m}$ will be identified with the assignment that makes each v_i equal to r_i. A point *satisfies* the tableau if the corresponding assignment satisfies all the constraints of the tableau. The *solution set* of the tableau is the set of points that satisfy it. If the solution set is not empty, the *solution flat* of the tableau is the smallest affine set containing the solution set. The *sample point* of the tableau (1) is the point obtained by setting each column variable x_i to zero, and each row variable y_i to c_i. Obviously the sample point satisfies all the row constraints of the tableau, but it may violate the sign constraints of some of the row variables. A tableau is *feasible* if its sample point satisfies all its sign constraints.

For example, the solution set of

$$\begin{array}{c|ccc} & {}^{\triangledown} x^{\geq} & y^{\geq} & \\ \hline a^{\geq} & -1 & 0 & 1 \\ b^{\geq} & 0 & -1 & 1 \end{array}$$

is a square; its solution flat is a two dimensional affine subspace of R^4, and its sample point is $x = y = 0$, $a = b = 1$.

The satisfiability problem for $\mathcal{R}$ is equivalent to the problem of determining whether a tableau has a solution, since a constraint like $x + y \geq 0$ can be represented by introducing a new variable v, making v the owner of a row representing $x + y$ (thus creating a row constraint $v = x + y$), and then restricting v. Of course, the actual program does not generate named variables, but it is convenient to imagine that it does.

Let T be a feasible tableau and y a row variable of T. Then y is *manifestly maximized* in T if every non-zero entry in y's row (outside of the constant column) is negative, and lies in a column owned by a restricted variable. For example, l is manifestly maximized in

$$\begin{array}{c|cccc}
\nabla & m^{\geq} & n^{\geq} & x & \\
\hline
y & 1 & -1 & 2 & 0 \\
l & -1 & -3 & 0 & -1
\end{array} \tag{3}$$

If a variable is manifestly maximized in T, then its maximum value over the solution set of the T is the constant entry in its row. For example, the maximum value of l over the solution set of (3) is -1, since, over the solution set of T, $l = -m - 3n - 1$ and m and n are non-negative.

Let T be a feasible tableau and x a column variable of T. Then x is *manifestly unbounded* in T if every negative entry in the column owned by x is in a row owned by an unrestricted variable.

For example, x is manifestly unbounded in

$$\begin{array}{c|ccc}
\nabla & x & u^{\geq} & \\
\hline
l^{\geq} & 1 & 1 & 0 \\
y & -1 & -1 & 1 \\
z & -1 & -2 & -1 \\
m^{\geq} & 0 & 1 & 2
\end{array} \tag{4}$$

Obviously if x is manifestly unbounded in T, then the solution set for T contains points where x takes on arbitrarily large values. For example, (4) is satisfied by its sample point $x = u = l = 0$, $y = 1$, $z = -1$, $m = 2$; if we move this point by increasing x, leaving u zero, and determining l, y, z, and m as required by the row constraints, then no matter how far we move the point neither l nor m will become negative: m will stay 2 and l will always be equal to x.

The *pivot* operation transforms a tableau into another with the same solution set. A tableau is pivoted by choosing a pivot row and a pivot column such that the entry in the pivot row and column is not zero, transforming the arrays a_{ij} and c_i by the rule

$$\begin{array}{c|cc}
 & \text{Pivot Column} & \text{Any Other Column} \\
\hline
\text{Pivot Row} & a & b \\
\text{Any Other Row} & c & d
\end{array}
\quad \longrightarrow \quad
\begin{array}{c|cc}
 & \text{Pivot Column} & \text{Any Other Column} \\
\hline
\text{Pivot Row} & 1/a & -b/a \\
\text{Any Other Row} & c/a & d - bc/a
\end{array}$$

and finally exchanging the owners of the pivot row and pivot column. For example,

$$\begin{array}{c|ccc}
\nabla & l & y & \\
\hline
x & 2 & 1 & -1 \\
m & 2 & 1 & 3
\end{array} \tag{5}$$

is transformed by pivoting l and x into

$$\begin{array}{c|ccc}
\nabla & x & y & \\
\hline
l & 1/2 & -1/2 & 1/2 \\
m & 1 & 0 & 4
\end{array} \tag{6}$$

The pivot operation amounts to solving the equation corresponding to the pivot row for the variable owning the pivot column, and then eliminating the column variable in terms of the row variable in each row equation. For example, (5) is transformed into (6) by rewriting $x = 2l + y - 1$ as $l = x/2 - y/2 + 1/2$, and then using this equation to rewrite $m = 2l + y + 3$ as $m = x + 4$. Using this fact it is easy to check that the pivot operation preserves the solution set of the tableau.

Here is the fact on which the simplex algorithm is based: If T is a tableau that is satisfied by its sample point, and v is anv variable of T, then T may be transformed by a sequence of pivot operations into a tableau T' in which v is either manifestly unbounded or manifestly maximized. Any of several simple rules will generate an appropriate sequence of pivots; no backtracking is needed. All of the commonly-used rules have been shown to require exponentially many pivots in the worst case, but in practice the worst case does not arise. A satisfactory rule is to choose any pivot that increases the sample value of v and produces a feasible tableau.

For example, consider the inequalities

$$y \leq x+1 \quad x+y \geq -3 \quad x \leq -4.$$

They are satisfiable if and only if there is a point in the solution set of the following tableau at which c is non-negative:

$$\begin{array}{c|rrr} & \nabla\;\; x & y & \\ \hline a^{\geq} & 1 & -1 & 1 \\ b^{\geq} & 1 & 1 & 3 \\ c & -1 & 0 & -4 \end{array} \tag{7}$$

Notice that c has not yet been restricted. It is technically convenient to deal only with feasible tableaux; so the restriction will not be added unless it is determined that it is consistent to do so.

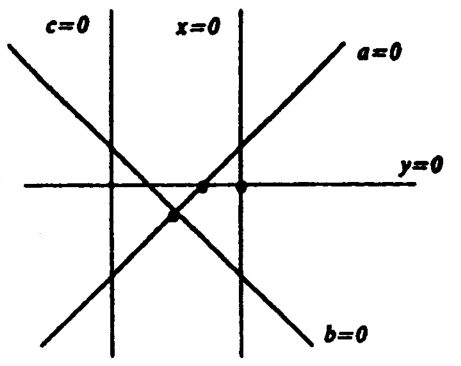
Our goal is to transform (7) into a tableau in which c is manifestly maximized or manifestly unbounded. We restrict our attention to pivots that increase the sample value of c and produce a feasible tableau. Only one pivot for (7) satisfies these restrictions: the pivot of x and a. This pivot produces the tableau:

$$\begin{array}{c|rrr} & \nabla\; a^{\geq} & y & \\ \hline x & 1 & 1 & -1 \\ b^{\geq} & 1 & 2 & 2 \\ c & -1 & -1 & -3 \end{array} \tag{8}$$

Since c is neither manifestly maximized nor manifestly unbounded, we continue pivoting. Again there is only one reasonable pivot, the pivot of y and b, which produces:

$$\begin{array}{c|rrr} & \nabla\;\; a^{\geq} & b^{\geq} & \\ \hline x & 1/2 & 1/2 & -2 \\ y & -1/2 & 1/2 & -1 \\ c & -1/2 & -1/2 & -2 \end{array} \tag{9}$$

Now c is manifestly maximized at -2, so the original inequalities are unsatisfiable. (Notice that x is manifestly *minimized*, in an obvious sense, in (9); if we had known that the inequalities would be unsatisfiable we could have avoided adding the row for c, and instead minimized x directly.)



(A)

Figure (A) shows the solution flat of (9), which is a two-dimensional affine subset of R^5. The hyperplanes $x = 0$, $y = 0$, $a = 0$, $b = 0$, and $c = 0$ intersect the solution flat in five lines. If T' is any tableau produced from (9) by a sequence of pivots, then the sample point of T' is the intersection of the two lines in the diagram that correspond to the column variables of T'. Futhermore, for each intersection point there is such a tableau T'. If T' and T'' are two such tableaux, then T' can be obtained from T'' (if we ignore the order of the rows and columns of T'') in a single pivot if and only if the points corresponding to T' and T'' both lie on one of the lines—the line where the common column variable of T' and T'' is zero.

The solution set of (9) is the quarter-plane to the right of the lines $a = 0$ and $b = 0$. The sequence of pivots above moved the sample point first left and then down and to the left, till the extreme point of the solution set in the direction of increasing c was reached. This example should make it intuitively clear that a sequence of pivots in the direction of increasing y will lead to a tableau in which y is either manifestly maximized or manifestly unbounded.

Let us now consider how the pivots are chosen. This is described in every book on linear programming, but generally under the assumption that all variables are restricted, so it is necessary to make a few simple modifications to handle the unrestricted variables.

If we are trying to increase the value of the row variable x, then pivoting the row variable y with the column variable z will help if the sample value of z is increased and x has a positive "correlation" with z, or if z is decreased and x has a negative correlation with z. Since the sample value of a column variable is zero and it is illegal to make a restricted variable negative, the column owned by z is a potential place to pivot if either (a) The entry in z's column and x's row is positive, or (b) the entry in z's column and x's row is negative, and z is an unrestricted variable. The first step in finding a pivot is to choose a column satifying either (a) or (b). Notice that if no such column exists, then x is manifestly maximized.

The rule for choosing the pivot row depends on whether (a) or (b) holds. Suppose that (a) holds; then we are trying to raise the value of z in order to raise the value of x. We will raise z as much as we can without making the constant entry in any restricted row negative. If row i is owned by a restricted variable, and z owns column j, then z can be raised to $-c_i/a_{ij}$ before the value of the owner of row i becomes zero. Note that we are assuming c_i is non-negative (since the tableau is satisfied by its sample point) and that a_{ij} is negative, so $-c_i/a_{ij}$ is nonnegative as it should be. This leads to the pivot rule: If there are any rows owned by restricted variables with negative entries in the pivot column, choose from these rows that row i such that $-c_i/a_{ij}$ is smallest; and pivot in that row.

If the pivot column was chosen to satisfy (b) instead of (a), the choice of the pivot row is essentially the same, but since z is being lowered instead of raised, it is the restricted rows with positive entries in z's column that pose restrictions.

Algorithm F (*Find a pivot*). Find a pivot that increases the sample value of y_i. If y_i is manifestly maximized, return $(0,0)$. Otherwise return a pair of indices (i',j') such that pivoting at (i',j') will increase the sample value y_i.

F1. [Find pivot column.] If there is some column j such that x_j is not restricted and a_{ij} is non-zero, then set $j' \leftarrow j$, $sgn \leftarrow a_{ij}$, and go to step F2. Otherwise, if there is some column j such that a_{ij} is positive, set $j' \leftarrow j$, $sgn \leftarrow +1$, and go to step F2. Otherwise return $(0,0)$.

F2. [Initialize *champ* and *score*.] (If $sgn > 0$, find a pivot in column j' to raise the sample value of $x_{j'}$; if $sgn < 0$, find a pivot in column j to lower the sample value of $x_{j'}$.) Set $champ \leftarrow i$, $score \leftarrow \infty$.

F3. [Find pivot row.] For each row i' such that $i' \neq i$, the owner of row i' is restricted, and $sgn \times a_{i'j'} < 0$, if $|c_{i'}/a_{i'j'}| < score$, then set $champ \leftarrow i'$, $score \leftarrow |c_{i'}/a_{i'j'}|$.

F4. [Return answer.] Return (i',j'). ▮

(If the final value of *score* is zero, the pivot does not increase y_i; in this case any ties should be broken by one of the rules that avoid "cycling"—for a discussion of such rules, see any textbook on linear programming.)

Two observations about this algorithm will be useful later.

First, if the algorithm returns a pivot (i',j') such that $i' = i$, then $x_{j'}$ can be moved indefinitely in the direction required to increase $y_{i'}$, without violating any sign constraints. Obviously then, there are points in the solution set of the tableau where $y_{i'}$ takes on arbitrarily large values. In fact it is easy to show that in this case, $y_{i'}$ will become manifestly unbounded after pivoting at (i',j').

Second, given a feasible tableau T and a column variable x_j of T such that the column owned by x_j is not identically zero, then there is a pivot a_{ij} in column j such that after pivoting at a_{ij}, the sample point of the resulting tableau satisfies every sign constraint except possibly the sign constraint associated with y_i. (y_i is the variable that used to be x_j.) For, if there is a positive entry in the column; then set $sgn \leftarrow +1$ and apply steps F2, F3, and F4 to the column; if there is no positive entry; then there must be a negative entry; apply steps 2, 3, and 4 with $sgn = -1$.

We now consider how to determine the equalities that need to be propagated. A tableau is *minimal* if it is satisfiable and the dimension of its solution flat is equal to the number of its columns. An example of a non-minimal tableau is:

$$\begin{array}{c|cc} & \nabla & x^{\geq} \\ \hline y^{\geq} & -1 & 0 \end{array}$$

The solution set (hence also the solution flat) of this tableau is the single point $x = y = 0$.

Lemma 1. *Let u and v be any two variables of a minimal tableau. Then the equality $u = v$ is satisfied by every solution of the tableau if and only if either (a) u and v are both row variables, and the rows they own are identical; or (b) u owns a row, v owns a column, and every entry in the row owned by u is zero except the entry in v's column, which is 1; or vice-versa.*

Proof. Let H_i, for i from 1 to n, be the hyperplane which is the solution set of the ith row constraint. These hyperplanes are in general position, since the row variables are distinct. Thus the intersection of the H_i is m-dimensional, and therefore must be the solution flat. The hyperplane $u = v$ is affine, so it is satisfied at every point in the solution set if and only if it is satisfied at every point in the solution flat. In other words, $u = v$ is true at every point in the solution set if and only if $u = v$ is a consequence of the row constraints. But $u = v$ is a consequence of the row constraints if and only if the expression $u - v$ is a linear combination of the n expressions $y_i - a_{i1}x_1 - \cdots - a_{im}x_m - c_i$, for i from 1 to n. Obviously this cannot happen if three or more of the coefficients in the linear combination are non-zero, since then three or more y's will appear in the combination. If the linear combination has two non-zero coefficients, the only way the result can be $u - v$ is if (a) holds. If the linear combination has one non-zero coefficient, the only way the result can be $u - v$ is if (b) holds. ∎

This lemma shows that it is straightforward to propagate equalities if the tableau is minimal. It is therefore desirable to keep the tableau minimal. This is where *dcol* comes in.

When we are propagating equalities it will be convenient to have a variable whose value is known to be zero. Therefore, we will assume that every tableau contains an identically zero row owned by the distinguished variable Z.

Consider the constraints:

$$x \leq 1, \quad y \leq 1, \quad x + y \geq 2.$$

These are satisfiable if and only if there is a point in the solution set of the following tableau where c is non-negative:

	▽ x	y	
Z	0	0	0
$a^{\geq}$	-1	0	1
$b^{\geq}$	0	-1	1
c	1	1	-2

We maximize c as before. The first pivot will either be of a and x or of b and y; suppose it is of a and x. The result is:

	▽ $a^{\geq}$	y	
Z	0	0	0
x	-1	0	1
$b^{\geq}$	0	-1	1
c	-1	1	-1

Now the only reasonable pivot is of b and y, with result:

	▽ $a^{\geq}$	$b^{\geq}$	
Z	0	0	0
x	-1	0	1
y	0	-1	1
c	-1	-1	0

(10)

Thus a sample point is found where c is non-negative, and the constraints are shown to be satisfiable. But if the restriction $c \geq 0$ is added to the tableau, it becomes non-minimal. This is obviously associated with the fact that c is manifestly maximized at zero. If $c \geq 0$, it must be that $c = 0$. More important, it must be that a and b are zero, since if either of them were positive then c would be negative. Thus the two "independent" variables of the tableau are fixed at zero, so the tableau is obviously non-minimal.

The most obvious way to proceed would be to propagate the equalities $a = Z$, $b = Z$, delete the columns owned by a and b, producing the minimal tableau

$$\begin{array}{c|c} & \triangledown \\ \hline Z & 0 \\ x & 1 \\ y & 1 \\ c & 0 \end{array} \tag{11}$$

and then search this tableau for entailed equalities in the manner suggested by lemma 1, thus propagating $x = y$ and $c = Z$.

The only trouble with minimizing the tableau in this way is that there is no easy way to "undo" the assertion that caused the columns to be deleted. Therefore, instead of deleting the columns, the satisfiability procedure regards them as "dead columns" and ignores them. Columns 1, ..., *dcol* are the dead columns, the other columns are "live" columns. Instead of (11), (10) is transformed into

$$\begin{array}{c|rrr} & a & b & \triangledown \\ \hline Z & 0 & 0 & 0 \\ x & -1 & 0 & 1 \\ y & 0 & -1 & 1 \\ c & -1 & -1 & 0 \end{array}$$

which for almost all purposes is regarded as the same as (11). In particular, after the two columns are "killed" in this way, the equalities $x = y$ and $c = Z$ are propagated, because the corresponding pairs of rows are equal if the entries in the dead columns are ignored.

The preceding definitions in this section were all for the case $dcol = 0$; they must be changed in an obvious way if $dcol \neq 0$. For example, the set of constraints represented by a tableau must be extended to include $x_1 = 0, \ldots, x_{dcol} = 0$; a tableau is minimal if the dimension of its solution flat is the number of its live columns, etc. A pivot may not be chosen in a dead column, but the entries in the dead columns are updated during pivoting in the usual way. Thus columns can be "revived" simply by decrementing *dcol*.

We will now describe the two basic algorithms of the satisfiability procedure. Algorithm A adds rows and/or columns to the tableau in the obvious way, and Algorithm R restricts variables of the tableau to be non-negative. Both algorithms make entries on the global list *undolist*, in order that the changes they make to the tableau can be undone. The entries on the undolist contain the constant tag values $A1$, $R1$, and $R2$.

Algorithm A (*Add a row to the tableau*). Let L be the formal affine sum $k_0 + k_1 v_1 + \cdots + k_p v_p$, where the v_i are variables and the k_i are rationals. This algorithm looks for a variable of the tableau that is equal to L at every point satisfying the tableau. If such a variable exists, it is returned. Otherwise a new row is added to represent L, a new variable is created to own the row, and the new variable is returned.

A1. [Add empty row.] Set $n \leftarrow n + 1$, let g be a variable that appears nowhere in the tableau, make g the owner of row n, make row n identically zero, and set $c_n \leftarrow k_0$.

A2. [Add new columns.] For i from 1 to p, if v_i is not a variable of the current tableau, set $m \leftarrow m + 1$, make v_i the owner of column m, make column m identically zero, and put an entry $(A1, v)$ on *undolist*.

A3. [Compute entries in new row.] For i from 1 to p, if v_i owns a live column, let j be the index of the column and set $a_{nj} \leftarrow a_{nj} + k_i$; otherwise v_i owns some row of T; let it be row i' and do the following: set $c_n \leftarrow c_n + k_i c_{i'}$, and for j from 1 to m, set $a_{nj} \leftarrow a_{nj} + k_i a_{i'j}$.

A4. [Is row trivial?] If c_n is zero and there exists a j such that every entry in row n is zero except for a_{nj}, which is 1, then set $n \leftarrow n - 1$ and return the owner of column j.

A5. [Is row really new?] If row n is identical to row i, for some $i \neq n$, then set $n \leftarrow n - 1$ and return the owner of row i.

A6. [Return owner of new row.] Add an entry $(A1, g)$ to *undolist* and return the variable g. ▮

For example, if Algorithm A is applied to the formal linear sum $0 + 1x$ then it will return x, adding a new column owned by x if necessary.

Algorithm R. (*Restrict a tableau variable*). Given a variable v of the current tableau, this algorithm determines whether it is consistent to make v restricted. If so, it restricts v, makes the resulting tableau minimal, and propagates equalities if necessary.

R1. [Is the restriction trivially redundant?] If v is already restricted, or if v owns a dead column, return `satisfiable`.

R2. [Guarantee that v owns a row.] If v owns a column, choose a pivot that brings v into a row and produces a tableau that is satisfied by its sample point, and perform the pivot. (This is always possible, as remarked after Algorithm F above.)

R3. [Determine sign of maximum of v.] Let i be the index of the row owned by v. If c_i is positive, add v to R (that is, make v restricted), put an entry $(R1, v)$ on *undolist*, and return `satisfiable`. Otherwise, use Algorithm F to find a pivot to increase the value of v; let (i', j') be the result. If $(i', j') = (0, 0)$, go to step R4. If $i' = i$, add v to R, put an entry $(R1, v)$ on *undolist*, and return `satisfiable`. Otherwise pivot at (i', j') and repeat this step.

R4. [Maximum negative?] (At this point v is manifestly maximized and its value is not positive.) If $c_i < 0$, return `unsatisfiable`.

R5. [Minimize tableau.] Now the tableau is satisfiable but non-minimal; we make it minimal and propagate equalities. Put an entry $(R2, dcol)$ on *undolist*. For $j \leftarrow dcol + 1, dcol + 2, \ldots, m$, if a_{ij} is non-zero, then do the following: propagate an equality between x_j and Z, exchange column j with column $dcol + 1$, and set $dcol \leftarrow dcol + 1$.

R6. [Propagate equalities.] For each j, k such that $1 \leq j < k \leq n$, if row j and row k are identical, ignoring the dead columns, then propagate an equality between the owners of row j and row k. For each j, $1 \leq j \leq n$, if row j has only a single non-zero entry outside the dead columns, and the entry is in column k, which is not the constant column, and the non-zero entry is one, then propagate an equality between the owner of row j and the owner of column k. ▮

If we were only concerned with satisfiability and did not need to propagate equalities, then step R2 would be unnecessary—when a column variable is restricted, it cannot make the context inconsistent, since the variable is zero at the sample point.

It is very easy to "undo" these operations on the tableau. To save a state, put a conventional mark on *undolist*. To return to the last marked state, go down the list of pairs until the most recent mark is reached; for each pair $(R1, v)$, v will be a restricted variable; make it unrestricted; for each pair $(R2, d)$, set $dcol \leftarrow d$; for each pair $(A1, v)$, if v owns a row of T, delete the row; if v owns a column of T, pivot v into a row and delete the row, unless the column is identically zero, in which case delete the column.

To assert an equality $v = 0$, where v is a variable of the tableau, we would like to pivot the variable into a column and kill the column. But this strategy may not be valid, since if the hyperplane $v = 0$ does not intersect the relative interior of the solution set of the tableau, then the assertion lowers the dimension of the solution set by more than one. To detect this situation we pivot in order to determine the sign of the smallest and largest values of v. If these are negative and positive respectively, then it is all right to pivot v into a column and kill the column. If both are negative, or both are positive, then the assertion is inconsistent. If both the maximum and minimum of v are zero, the assertion is redundant. (This will never happen in a minimal tableau unless v owns an empty row or a dead column.) The two cases that remain are similar; suppose for example that v's maximum is zero and its minimum negative. Then in pivoting to maximize v, a tableau will be reached where v is manifestly maximized at zero. At this point we can perform steps R5 and R6, just as if the assertion had been $v \geq 0$.

For an example of an equality constraint, suppose that the tableau represents the contraints $0 \leq x \leq 1$, $0 \leq y \leq 1$, and $0 \leq z \leq 1$, that the sample point is the origen, and that the equality $x + y + z = 3/2$ is added. Then one of the variables (say x) will be pivoted into a row and given the value 1, then one of the other variables (say y) will be pivoted into a row. It is feasible to pivot y with the slack variable that represents $y \leq 1$, and thereby raise the value of y to 1; of $x + y + z$ to 2. Since this is larger than necessary to satisfy the new equality contraint, we exchange y with the slack variable of the new contraint, and thereby raise y to 1/2. Thus we have pivoted along one edge of the solution set, then half way along another, to reach the hexagon sliced out of the cube by the new equality contraint. The slack for the equality constraint now owns a column, which can be swapped into the region for dead columns and ignored until it is necessary to "undo" the equality contraint. To undo the constraint, it suffices to pivot in any feasible place in the column, thereby making the slack own a row, and then to delete the row.

The program actually represents restrictions of the form $v \leq 0$, $v > 0$, and $v < 0$ directly, using obvious modifications of the algorithms above. Also, a bit is set in a header node if it is restricted to have an integral value; thus strict inequalities between integers can be handled correctly.

The entries a_{ij} are represented using the linked structure described in Knuth [34], pp. 299–302. The rows are linked from right to left; the columns from bottom to top; entries that are zero are not represented. The matrix is usually sparse, so this saves both time and space. The one-dimensional arrays x and y do not contain variables, but record structures which serve as header nodes for the row and column lists. These record structures also contain pointers at enodes if they represent real variables or arithmetic terms; if they just represent "slack variables," the pointer is nil.

The sparse matrix representation speeds up the execution of step A5 and step R6. To check if a row is equal to some other row, traverse some column in which the row has a non-zero entry; usually the column list will have very few entries compared to the number of rows. In step R6, it is often the case that only one column is killed; in this case only rows with a non-zero entry in the column have changed. Thus only these rows need to be checked against the other rows.

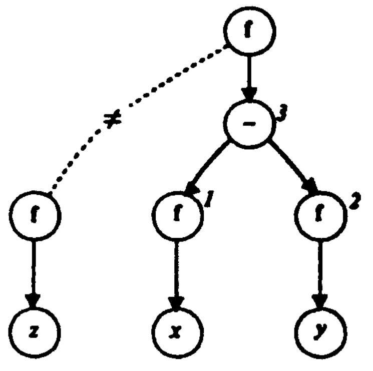
13. An example with $\mathcal{E}$ and $\mathcal{R}$

Algorithm D of section 11 can be modified to incorporate the satisfiability procedure for $\mathcal{R}$; essentially by propagating equalities between the E-graph and the simplicial tableau as in Algorithm E of section 5. We will not describe the modification in detail, but consider one example: Suppose that Algorithm D is applied to the conjunction:

$$\mathrm{f}(\mathrm{f}(x) - \mathrm{f}(y)) \neq \mathrm{f}(z) \wedge (z = \mathrm{f}(x) - \mathrm{f}(y) \vee z \geq 0) \wedge x \leq y \wedge y + z \leq x.$$

This is a slight variation of the example in section 5: if the $z \geq 0$ case of the disjunction is considered, we have the unsatisfiable conjunction from that section; if the $z = \mathrm{f}(x) - \mathrm{f}(y)$ case is considered, the conjunction is obviously unsatisfiable.

The first execution of step D1 creates E-nodes for the terms $\mathrm{f}(\mathrm{f}(x) - \mathrm{f}(y))$ and $\mathrm{f}(z)$, and forbids the merge of the two nodes. The procedure that creates an E-node to represent a term will, if the term contains free functions of $\mathcal{R}$, add one or more rows to the simplicial tableau and establish links between the E-graph and the tableau; thus after the disequality $\mathrm{f}(\mathrm{f}(x) - \mathrm{f}(y)) \neq \mathrm{f}(z)$ has been asserted, the data structures are as follows:



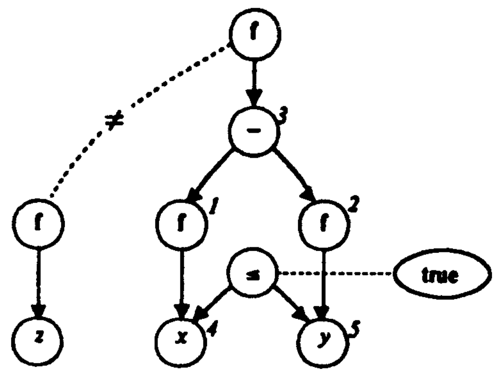
,

$$\begin{array}{c|cc} & \nabla\boxed{1} & \boxed{2} \\ \hline \boxed{3} & 1 & -1 \end{array} \qquad (1)$$

Here is an expanation of (1). The E-graph contains vertices that represent the terms $\mathrm{f}(x)$, $\mathrm{f}(y)$, and $\mathrm{f}(x) - \mathrm{f}(y)$ in the obvious way. We could proceed as in section 5 by "labelling" these terms with variables g_1, g_2, and g_3 and representing the equality $g_3 = g_1 - g_2$ in the tableau. But it is awkward and somewhat inefficient to introduce new variables. Instead, we will use no variables as headers in the simplex tableau at all; the headers will be record structures that point at (and are pointed at by) E-nodes. The record structures are represented in the figure by squares with numbers in them; the numbers have no significance but that of distinct labels. The superscripts on the E-nodes for $\mathrm{f}(x)$ and $\mathrm{f}(y)$ indicate that these nodes contain pointers at the tableau header nodes $\boxed{1}$ and $\boxed{2}$. The tableau header nodes also contain backpointers to the E-nodes, although these are not indicated explicitly in the figure. To create an E-node for the term $\mathrm{f}(x) - \mathrm{f}(y)$, Algorithm A was applied to the formal linear sum $1 \cdot \boxed{1} - 1 \cdot \boxed{2}$; the algorithm created and returned the new tableau header node $\boxed{3}$. This header node was associated with the E-node for $\mathrm{f}(x) - \mathrm{f}(y)$. The rest of Figure 9 represents the disequality $\mathrm{f}(\mathrm{f}(x) - \mathrm{f}(y)) \neq \mathrm{f}(z)$ in the obvious way. (A dotted line labelled with $\neq$ represents a disequality or forbidden merge.)

When step D1 is repeated, it considers the disjunction $z \geq 0 \vee z = \mathrm{f}(x) - \mathrm{f}(y)$. We will assume that the modifications described in section 11 have been made, so that the theorem prover puts off case splits as long as possible. Thus this disjunction is saved away, and step D1 is repeated a third time.

This time the conjunct to be processed is $x \leq y$, which is an abbreviation for $(x \leq y) = \mathbf{true}$. E-nodes are created for **true** and for $x \leq y$, and then the nodes are merged. Just as an equivalence class being merged with the equivalence class of **true** is scanned for vertices labelled **atom** (see section 9), so is it scanned for vertices labelled $<$, $>$, $\leq$, or $\geq$. In our example, when the vertex for $x \leq y$ is merged with **true**, tableau header nodes $\boxed{4}$ and $\boxed{5}$ are created and associated with x and y, a "slack variable" is created to represent $y - x$, and the slack variable is restricted. Here is the state of the data structure just before Algorithm R is applied to restrict the new slack variable:

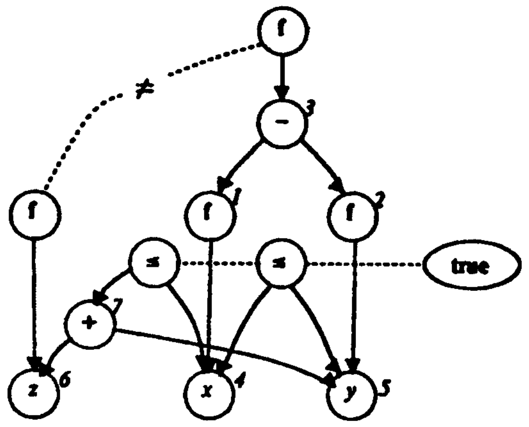


$$\begin{array}{c|cccc}
 & \nabla\boxed{1} & \boxed{2} & \boxed{4} & \boxed{5} \\
\hline
\boxed{3} & 1 & -1 & & \\
\boxed{} & & & -1 & 1
\end{array} \tag{2}$$

Algorithm R performs one pivot, changing the matrix (2) into:

$$\begin{array}{c|cccc}
 & \nabla\boxed{1} & \boxed{2} & \boxed{4} & \boxed{}^{\geq} \\
\hline
\boxed{3} & 1 & -1 & & \\
\boxed{5} & & & 1 & 1
\end{array} \tag{3}$$

The next assertion is $y + z \leq x$. When the E-node labelled $+$ is created to represent $y + z$, tableau header nodes $\boxed{6}$ and $\boxed{7}$ are created, representing z and $y + z$ respectively. (Note that y is already represented by $\boxed{5}$.) When the node for $y + z \leq x$ is merged with true, a "slack variable" is created to represent $x - y - z$ and is then restricted. Just before it is restricted, the data structure looks like this:



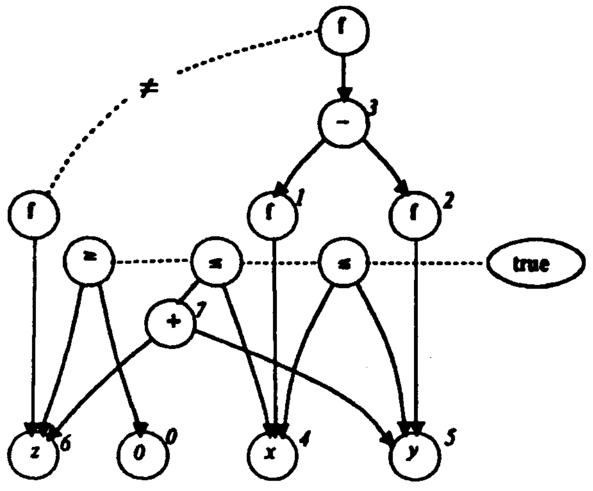
$$\begin{array}{c|ccccc}
 & \nabla\boxed{1} & \boxed{2} & \boxed{4} & \boxed{}^{\geq} & \boxed{6} \\
\hline
\boxed{3} & 1 & -1 & & & \\
\boxed{5} & & & 1 & 1 & \\
\boxed{7} & & & 1 & 1 & 1 \\
\boxed{} & & & & -1 & -1
\end{array} \tag{4}$$

Note that the row owned by $\boxed{7}$ was obtained by copying the row owned by $\boxed{5}$ and adding 1 to the copy in the column owned by $\boxed{6}$. When the new slack is restricted, it is pivoted with $\boxed{6}$ to produce:

$$\begin{array}{c|ccccc}
 & \nabla\boxed{1} & \boxed{2} & \boxed{4} & \boxed{}^{\geq} & \boxed{}^{\geq} \\
\hline
\boxed{3} & 1 & -1 & & & \\
\boxed{5} & & & 1 & 1 & \\
\boxed{7} & & & 1 & & -1 \\
\boxed{6} & & & & -1 & -1
\end{array} \tag{5}$$

At this point Algorithm D considers the split $f(x) - f(y) = z \lor z \geq 0$. An entry is made on *undostack* to save the current state; then $f(x) - f(y) = z$ is asserted. Since E-nodes exist for $f(x) - f(y)$ and for z, these two nodes are merged. Since both of the E-nodes are associated with tableau header nodes, the program recognizes that it is necessary to propagate an equality to $\mathcal{R}$, namely $\boxed{3} = \boxed{6}$. But for efficiency's sake, the E-graph is closed under congruences before any work is done in the simplicial tableau. Thus, the equality $\boxed{3} = \boxed{6}$ is saved on a list (it is similar to *mergelist*, so let us call it *rmergelist*), and the graph is closed under congruences. This immediately sets *inconsistent*, since the merge of $f(z)$ and $f(f(x) - f(y))$ has been forbidden. Thus *mergelist* and *rmergelist* are reset to nil, and pop returns to consider the case $z \geq 0$.

When an E-node is created for a numeral, a row must be added to the simplicial tableau to represent the corresponding number. Thus when the term $z \geq 0$ is added to the E-graph, a node $\boxed{0}$ is added to represent zero. There is already a header node for z, so $\boxed{0}$ is the only new header node. When $z \geq 0$ is asserted, Algorithm A is used to create a slack variable for $z - 0$; it returns the existing header node $\boxed{6}$. This is all right, since any header node can be restricted. Thus, Algorithm R is applied to restrict node $\boxed{6}$ in the following position:



$$\begin{array}{c|ccccc}
 & {}^{\nabla}\boxed{1} & \boxed{2} & \boxed{4} & \boxed{}^{\geq} & \boxed{}^{\geq} \\
\hline
\boxed{3} & 1 & -1 & & & \\
\boxed{5} & & & 1 & 1 & \\
\boxed{7} & & & 1 & & -1 \\
\boxed{6} & & & & -1 & -1 \\
\boxed{0} & & & & &
\end{array}$$

(6)

But node $\boxed{6}$ is manifestly maximized in this tableau, so Algorithm R kills the columns owned by the two slack variables by swapping them to the left and incrementing *dcol*, thus producing the following tableau:

$$\begin{array}{c|ccccc}
 & \boxed{}^{\geq} & \boxed{}^{\geq} & {}^{\nabla}\boxed{4} & \boxed{2} & \boxed{1} \\
\hline
\boxed{3} & & & & -1 & 1 \\
\boxed{5} & 1 & & 1 & & \\
\boxed{7} & & -1 & 1 & & \\
\boxed{6} & -1 & -1 & & & \\
\boxed{0} & & & & &
\end{array}$$

(7)

The last step of Algorithm R then scans the rows of the matrix and detects the equalities $\boxed{4} = \boxed{5} = \boxed{7}$ and $\boxed{6} = \boxed{0}$. These are propagated to $\mathcal{E}$, which uses $\boxed{4} = \boxed{5}$ to deduce $f(x) = f(y)$, and propagates the deduction to $\mathcal{R}$ in the form $\boxed{1} = \boxed{2}$. (The three equalities $\boxed{5} = \boxed{7}$, $\boxed{4} = \boxed{7}$, and $\boxed{6} = \boxed{0}$ cause two merges in the E-graph, but the merges do not lead to any congruences.)

So, finally, here is an example showing how $\mathcal{R}$ handles the propagated equalities that it receives. To assert $\boxed{1} = \boxed{2}$, $\mathcal{R}$ constructs a slack that represents either $\boxed{1} - \boxed{2}$ or $\boxed{2} - \boxed{1}$, and "kills" the slack. In this case there is already a node (namely $\boxed{3}$) representing $\boxed{1} - \boxed{2}$, so if we are lucky enough to form the difference this way, no new nodes will be created. Let us suppose, though, that we are unlucky and create a slack for $\boxed{2} - \boxed{1}$; then before the new slack is killed, the matrix looks like this:

$$\begin{array}{c|ccccc}
 & \boxed{}^{\geq} & \boxed{}^{\geq} & {}^{\triangledown}\boxed{4} & \boxed{2} & \boxed{1} \\
\hline
\boxed{3} & & & & -1 & 1 \\
\boxed{5} & 1 & & 1 & & \\
\boxed{7} & & -1 & 1 & & \\
\boxed{6} & -1 & -1 & & & \\
\boxed{0} & & & & & \\
\boxed{} & & & & 1 & -1
\end{array} \tag{8}$$

The new node is pivoted with node $\boxed{1}$, and then its column is killed by swapping it with the column owned by $\boxed{4}$ and incrementing *dcol*. The result is:

$$\begin{array}{c|ccccc}
 & \boxed{}^{\geq} & \boxed{}^{\geq} & \boxed{} & {}^{\triangledown}\boxed{2} & \boxed{4} \\
\hline
\boxed{3} & & & -1 & & \\
\boxed{5} & 1 & & & & 1 \\
\boxed{7} & & -1 & & & 1 \\
\boxed{6} & -1 & -1 & & & \\
\boxed{0} & & & & & \\
\boxed{1} & & & -1 & 1 &
\end{array} \tag{9}$$

This causes $\mathcal{R}$ to propagate $\boxed{3} = \boxed{0}$ and $\boxed{1} = \boxed{2}$. The first of these equalities allows the congruence closure algorithm to deduce $f(z) = f(f(x) - f(y))$, and so the formula is found to be unsatisfiable.

14. Matching

The matching problem is outside the scope of this paper, but it would be misleading to ignore it, so this section gives an overview of the problem. It will be convenient to introduce the matching problem by describing the correctness verification of a sorting program, thus illustrating the symbolic evaluator, matcher, and satisfiability procedures all in action together.

The matcher's job is to choose instances of definitions, lemmas, and axioms that will make a given proof succeed. This is equivalent to the problem of guiding a resolution theorem-prover or using a set of rewrite rules; no satisfactory solution is known, though many approaches have been tried. The point of this section is not to advertise any particular new heuristics for the problem, but only to illustrate the differences between matching against ordinary list structure and matching against the E-graph and simplex tableau. Most heuristics that have been used for matching with ordinary list structure can be transferred to the matching problem with the new data structures.

The E-graph helps to solve the problem of unification in the presence of equalities. For example, if it is assumed that $g(a) = b$ and $g(b) = a$, then $p(a, a)$ can be proved from $(\forall x)\, p(g(g(x)), x)$ by instantiating x to a. But an ordinary matching algorithm would not consider $p(a, a)$ an instance of $p(g(g(x)), x)$. Even if the goal had been $p(g(g(a)), a)$ instead of $p(a, a)$, many systems would use the two equalities to simplify it to $p(a, a)$ before resorting to matching, and the match would never be found. This is not a problem if an E-graph is used, because the congruence closure algorithm does not "use" an equality $a = b$ by choosing one of a and b to replace all occurences of the other; it simply "notes" the equality and does no replacement. The matcher does not match $p(g(g(x)), x)$

against the list structure for $p(a, a)$, but against the congruence-closure algorithm's data structure for $p(a, a)$. The match does not fail when $g(g(x))$ is found to coincide with a, because the a in the data structure is equivalent to $g(b)$. Matching with the congruence closure algorithm's data structure is more expensive than matching with list-structure, but the advantages are worth the cost.

We begin with a fairly long example that illustrates the interaction between the matcher and the satisfiability procedures: the partial verification of a sorting program. The two most important things to verify about a sorting program are that its output is ordered, and that its output is a permutation of its input. In this section we verify only the first of these properties; the second can be handled by the techniques described in sections 15 and 16.

The procedure call selection-sort(a, n) sorts the numbers $a(1)$, $a(2)$, ..., $a(n)$ by finding the smallest and exchanging it with $a(1)$, then finding the second smallest and exchanging it with $a(2)$, and so forth. Here is the program:

```
Procedure selection-sort(a, n) ≡ sortloop(1).

Procedure sortloop(i) ≡                          sortloop(i) sorts a(i)...a(n):
    [[i ≥ n ⇒ [ ]];                              Nothing to sort? Then return.
     with j = minloop(i, a(i), i + 1)            j gets index of least of a(i)...a(n).
        do (a(i), a(j)) ← (a(j), a(i));          Exchange a(i) and a(j).
     sortloop(i + 1)]].

Procedure minloop(mindex, minval, k) ≡             mindex is index of minimum
    [[k > n ⇒ mindex;                                minval is value of minimum
     a(k) < minval ⇒ minloop(k, a(k), k + 1);        a(k)...a(n) are untested
     minloop(mindex, minval, k + 1)]].
```

We will use the notation

$$\text{Predicate } P(x_1, \ldots, x_n) \equiv F$$

to abbreviate

$$\text{Declare } (\forall x_1, \ldots, x_n)\, (P(x_1, \ldots, x_n) \Leftrightarrow F)$$

This can be taken as an axiomatic definition of P. Without going into details, we remark that there are simple conditions (amounting essentially to the requirement that P not occur in F) that guarantee that the definition is eliminable and that the "axiom" introduces no inconsistency.

To specify that the program sorts the numbers, we define the predicate

$$\begin{aligned}&\text{Predicate ordered}(a, n) \equiv \\ &\qquad (\forall i\, j)\ \text{int}(i) \wedge \text{int}(j) \wedge 1 \leq i \leq j \leq n \supset a(i) \leq a(j).\end{aligned}$$

We wish to prove that the program meets the following specifications:

Entering selection-sort, int(n).

Exiting selection-sort, ordered(a, n).

Ideally, the system would be able to prove this without assistance, but the theorem proving methods we are describing are not powerful enough. To help the system along, we describe what is true at the entry and exit points of **minloop** and **sortloop**.

When entering **sortloop**, the values $a(1)$, $a(2)$, ..., $a(i-1)$ are ordered, and furthermore they are all smaller than any of the values $a(i)$, $a(i+1)$, ..., $a(n)$. To specify this, we define another predicate:

$$\text{Predicate } \mathbf{leftordered}(a, i, n) \equiv$$
$$(\forall j\, k)\, \mathbf{int}(j) \wedge \mathbf{int}(k) \wedge 1 \leq j \leq i \wedge j \leq k \leq n \supset a(j) \leq a(k).$$

Now specifications for **sortloop** can be written:

Entering **sortloop**, $\mathbf{leftordered}(a, i-1, n) \wedge \mathbf{int}(i) \wedge \mathbf{int}(n) \wedge i \geq 1$.

Exiting **sortloop**, $\mathbf{ordered}(a, n)$.

When entering **minloop**, *minval* is the minimum of the values $a(i)$, $a(i+1)$, ..., $a(k-1)$, and *mindex* is the index of *minval* in the sequence a. To specify this, we define another predicate:

$$\text{Predicate } \mathbf{lowerbound}(x, a, m, n) \equiv (\forall k)\, \mathbf{int}(k) \wedge m \leq k \leq n \supset x \leq a(k).$$

Notice that **lowerbound** does not prohibit x from being strictly smaller than all of $a(m)$, ..., $a(n)$, because the entry specification for **minloop** explicitly states that $minval = a(mindex)$:

Entering **minloop**,
$\mathbf{lowerbound}(minval, a, i, k-1) \wedge minval = a(mindex) \wedge \mathbf{int}(k)$
$\wedge\ \mathbf{int}(mindex) \wedge i \leq k \leq n+1 \wedge i \leq mindex \leq n \wedge \mathbf{int}(n)$.

The value returned by **minloop** is the index of the smallest of $a(i)$, $a(i+1)$, ..., $a(n)$. Recall that by convention, the symbol $\mathcal{V}$ appearing in a procedure's exit specification denotes the value returned by the procedure. Thus the exit specification of **minloop** is

Exiting **minloop**, $\mathbf{lowerbound}(a(\mathcal{V}), a, i, n) \wedge i \leq \mathcal{V} \leq n \wedge \mathbf{int}(\mathcal{V})$.

It is annoying to have to include in the specifications the trivial facts that i, j, k and $\mathcal{V}$ are integers and the obvious inequalities between them, but they are required for the verification to succeed. There are techniques for adding such conjuncts to invariants automatically; for example, see Wegbreit [60] and Cousot and Halbwachs [14].

To prove that a procedure meets its specifications, the theorem prover's data structures are initialized to represent the procedure's entry specifications, then the procedure body is symbolically evaluated. The symbolic evaluator directs the theorem prover to make assumptions reflecting the path taken through the program. When the symbolic evaluator comes to the exit of the procedure, or an entry to another procedure, it directs the theorem prover to prove the appropriate specifications.

For example, to verify **sortloop**, the theorem prover's data structure is initialized to represent the conjunction of the following literals:

$\mathbf{int}(n)$
$\mathbf{int}(i)$
$i \geq 1$
$\mathbf{leftordered}(a, i-1, n)$

The theorem prover's data structure is a stack of literals that grows and shrinks as symbolic evaluation proceeds. (It is the data structure $c(1), \ldots, c(cr)$ of Algorithm D of section 11.) The symbolic evaluation of the body of **sortloop** first tries the $i \geq n$ branch of the statement by adding $i \geq n$ to the theorem prover's conjunction and symbolically evaluating the no-op $[\![\ ]\!]$. Now it must prove the exit specification **ordered**(a, n); what actually happens is that the theorem prover assumes $\neg$**ordered**(a, n), and then tries to show that what it has assumed is inconsistent. The assumption stack at this point is:

$$\begin{array}{l}
\textbf{int}(n)\\
\textbf{int}(i)\\
i \geq 1\\
\textbf{leftordered}(a, i-1, n)\\
i \geq n\\
\neg\,\textbf{ordered}(a, n)
\end{array}$$

Algorithm D cannot show the inconsistency of this conjunction, since the inconsistency depends on the definitions of **leftordered** and **ordered**. The theorem prover uses the definition of **ordered** first (in general it uses the definitions of negated predicates in the way about to be described): Since **ordered** is defined to be true if a certain clause is true for all i and j, and we have assumed **ordered**(a, n) is false, there must be some particular i_0 and j_0 for which it fails. Thus the theorem prover adds conjuncts to produce the following stack:

$$\begin{array}{l}
\textbf{int}(n)\\
\textbf{int}(i)\\
i \geq 1\\
\textbf{leftordered}(a, i-1, n)\\
i \geq n\\
\neg\,\textbf{ordered}(a, n)\\
\textbf{int}(i_0)\\
\textbf{int}(j_0)\\
1 \leq i_0 \leq j_0 \leq n\\
a(i_0) > a(j_0)
\end{array}$$

(The conjunct $\neg$**ordered**(a, n) is not removed from the stack, but it is flagged as "used", so that the last step will not be uselessly repeated with i_1, j_1, etc.) The theorem prover next uses the definition of **leftordered**, as follows. Since **leftordered**$(a, i-1, n)$ is true, the body of the predicate definition is known to be true for all j and k. The matcher looks for instantiations of j and k that are helpful. In this case, the right choice is to instantiate the j and k from **leftordered**'s definition to i_0 and j_0 respectively. Thus we get:

$$\begin{array}{l}
\textbf{int}(n)\\
\textbf{int}(i)\\
i \geq 1\\
\textbf{leftordered}(a, i-1, n)\\
i \geq n\\
\neg\,\textbf{ordered}(a, n)\\
\textbf{int}(i_0)\\
\textbf{int}(j_0)\\
1 \leq i_0 \leq j_0 \leq n\\
a(i_0) > a(j_0)\\
(\textbf{int}(i_0) \wedge \textbf{int}(j_0) \wedge 1 \leq i_0 \leq i \wedge i_0 \leq j_0 \leq n \supset a(i_0) \leq a(j_0))
\end{array}$$

Finally, we have a set of facts which Algorithm D can show are inconsistent, so the verification of this exit succeeds. (The only condition in the antecedant of the last line that is not manifest is $i_0 \leq i$, but this follows from $i \geq n$ and $i_0 \leq j_0 \leq n$.) The symbolic evaluator now returns to consider the possibility that the test $i \geq n$ failed. All of the facts added to the theorem prover's stack since then are removed, and $i < n$ is added, producing:

$$\begin{array}{l} \mathbf{int}(n) \\ \mathbf{int}(i) \\ i \geq 1 \\ \mathbf{leftordered}(a, i-1, n) \\ i < n \end{array}$$

Now the theorem prover must prove the entry specifications for minloop, given that *mindex*, *minval*, and k are bound to i, $a(i)$ and $i+1$ respectively. The negations of the conjuncts of the entry specification for minloop are added one at a time to the stack; each must be shown inconsistent. After the negation of the first conjünct is added, the stack contains:

$$\begin{array}{l} \mathbf{int}(n) \\ \mathbf{int}(i) \\ i \geq 1 \\ \mathbf{leftordered}(a, i-1, n) \\ i < n \\ \neg\,\mathbf{lowerbound}(a(i), a, i, i) \end{array}$$

(Note that we have used the simplification $i + 1 - 1 = i$. The symbolic evaluator would not acutally perform this simplification, but it makes our displays simpler, and the substitution of $i+1-1$ for i will not slow down Algorithm D by any appreciable amount.) The last formula implies that some k_0 exists contradicting the universal quantifier in the definition of lowerbound, so the stack becomes:

$$\begin{array}{l} \mathbf{int}(n) \\ \mathbf{int}(i) \\ i \geq 1 \\ \mathbf{leftordered}(a, i-1, n) \\ i < n \\ \neg\,\mathbf{lowerbound}(a(i), a, i, i) \\ \mathbf{int}(k_0) \\ i \leq k_0 \leq i \\ a(i) > a(k_0) \end{array}$$

which is trivially inconsistent ($\mathcal{R}$ finds $i = k_0$, $\mathcal{E}$ finds $a(i) = a(k_0)$, and $\mathcal{R}$ finds the inconsistency). Thus the last four lines are popped, and the negation of the next conjunct of the entry specification for minloop is added, namely $a(i) \neq a(i)$. It is trivially inconsistent, as are all the remaining entry specifications; so the verification of the entry to minloop succeeds. It is therefore valid to assume the exit specification for minloop, since minloop will be verified separately. The variable $\mathcal{V}$ in the exit specification may be replaced with the variable j, since the result of the call is assigned to j. Thus we have:

$\mathrm{int}(n)$
$\mathrm{int}(i)$
$i \geq 1$
$\mathrm{leftordered}(a, i-1, n)$
$i < n$
$\mathrm{lowerbound}(a(j), a, i, n)$
$i \leq j \leq n$
$\mathrm{int}(j)$

The assignment $a \leftarrow a^{(i)\ (j)}_{a(j)\ a(i)}$ is then executed symbolically, and the conjuncts of the entry specification for the recursive call to sortloop are proved one by one. They are all trivial except the one that says that the array has the left-ordered property. In order to prove this specification, the theorem prover must deduce a contradiction from the assumptions:

$\mathrm{int}(n)$
$\mathrm{int}(i)$
$i \geq 1$
$\mathrm{leftordered}(a, i-1, n)$
$i < n$
$\mathrm{lowerbound}(a(j), a, i, n)$
$i \leq j \leq n$
$\mathrm{int}(j)$
$\neg\,\mathrm{leftordered}(a^{(i)\ (j)}_{a(j)\ a(i)}, i, n)$

As usual, the definition of the negated predicate is used to produce:

$\mathrm{int}(n)$
$\mathrm{int}(i)$
$i \geq 1$
$\mathrm{leftordered}(a, i-1, n)$
$i < n$
$\mathrm{lowerbound}(a(j), a, i, n)$
$i \leq j \leq n$
$\mathrm{int}(j)$
$\neg\,\mathrm{leftordered}(a^{(i)\ (j)}_{a(j)\ a(i)}, i, n)$
$\mathrm{int}(j_0)$
$\mathrm{int}(k_0)$
$1 \leq j_0 \leq i$
$j_0 \leq k_0 \leq n$
$a^{(i)\ (j)}_{a(j)\ a(i)}(j_0) > a^{(i)\ (j)}_{a(j)\ a(i)}(k_0)$

Whenever the theorem prover is faced with a term of the form $a^{(i)}_{e}(j)$, it considers both the case where $i = j$ and the term is equal to e, and the case where $i \neq j$ and the term is equal to $a(j)$, as described in section 10. Thus the last formula above causes eight cases to be considered; each case is handled easily by the techniques that have already been illustrated. For example, if j_0 and k_0 are distinct from both i and j, then a contradiction is created by an obvious instantiation of the conjunct leftordered$(a, i-1, n)$—if every pair of elements in an array satisfy the left-ordered predicate, and two elements are swapped, the pairs that do not contain either of the swapped elements will still satisfy the left-ordered property. If $j_0 = i$, then $k_0 \geq i$, and an obvious instantiation of **lowerbound** completes the proof. If $k_0 = i$, then $j_0 \leq i$, and again an obvious instantiation of **leftordered** completes the proof.

After verifying the entry specifications to **sortloop**, the exit specification may be assumed. Finally the symbolic evaluator has come to the end of the procedure, so it must prove the exit specification. But the exit specification has just been assumed! So the verification obviously succeeds. (This situation arises whenever a loop is created by "tail-recursion", so to save work the program checks for tail-recursion explicitly.)

The verifications that **minloop** and **selection-sort** meet their specifications are similar.

The part of the above proof whose mechanization offers the greatest challenge is the matching, and therefore the matching problem has received a tremendous amount of attension. Some people even suggest that instead of using satisfiability procedures, a matcher or resolution theorem-prover be turned loose with the axioms for $\mathcal{R}$, $\mathcal{L}$, and $\mathcal{A}$. Unfortunately, no really satisfactory solution to the matching problem has been found, so the approach of this paper has been to find efficient satisfiability procedures. This is not to suggest that matching will not be necessary in almost every proof; it will be. There are a multitude of heuristics for choosing matches and deciding when to give up; any of them can be used in conjunction with Algorithm D, but they must be modified slightly to use the E-graph's data structure instead of list structure. So, in the remainder of this section, we will look at two examples of matching in the E-graph. They are minor variations of examples in Boyer and Moore [7], pages 111 and 107; it is instructive to compare their approach with ours.

We will assume that the quantified formula of which the matcher is choosing an instance has the form $(\forall x_1 \ldots x_n)\ F$, where F is a *clause*, that is, a disjunction of literals. The more general problem where arbitrary quantifier prefixes occur and arbitrary boolean structure is allowed in F can be reduced to this case by introducing "Skolem functions" to replace existentially quantified variables.

Recall that a substitution θ on $x_1, \ldots, x_n$ is a map that assigns to each variable x_i a term $x_i\theta$. If F is a quantifier-free formula and θ is a substitution on $x_1, \ldots, x_n$, then $F\theta$ denotes the formula obtained by simultaneously substituting $x_i\theta$ for each x_i in F.

The matcher looks for instances of universally quantified formulas that are "close" to the set of terms that are represented in the current E-graph. The exact definition of "close" depends on the heuristics being used; the definition that we will assume for the examples in this section is the following: Given the formula $(\forall x_1 \ldots x_n) F$ and a substitution θ on $x_1, \ldots, x_n$, the instance $F\theta$ is *close* to a given E-graph if for each x_i, there is some term t that occurs in F and properly contains x_i, such that $t\theta$ is represented in the E-graph. In our application F will be a clause $L_1 \vee \ldots \vee L_n$; to add the instance $F\theta$ means to assume the unquantified formula $L_1\theta \vee \ldots \vee L_n\theta$. In general, this is a proper disjunction that will cause a case-split, but in the special case that all but one of the $L_i\theta$ are contradicted by the E-graph, the $L_i\theta$ that is not contradicted can be asserted. (If all the $L_i\theta$ are contradicted, the context has been shown inconsistent.) If it happens for some i that $\neg L_i\theta$ is contradicted by the E-graph, then the instance is *subsumed*; such instances are ignored.

To illustrate matching in the E-graph, it suffices to use this crude algorithm: repeatedly add to the graph all non-subsumed close instances of quantified formulas, giving priority to those that do not cause case-splits. In practice we would first add those instances that create the fewest new nodes.

Example 1. Suppose that we have assumed or previously proved:

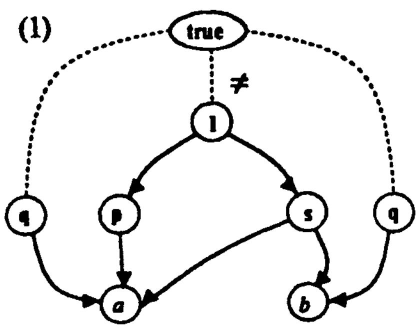
Axiom 1: $(\forall x)\ \mathrm{q}(x) \supset \mathrm{l}(\mathrm{p}(x), x)$.
Axiom 2: $(\forall x\, y)\ \mathrm{q}(y) \supset \mathrm{l}(x, \mathrm{s}(x, y))$.
Axiom 3: $(\forall x\, y\, z)\ \mathrm{l}(x, y) \wedge \mathrm{l}(y, z) \supset \mathrm{l}(x, z)$.

and from these we want to prove

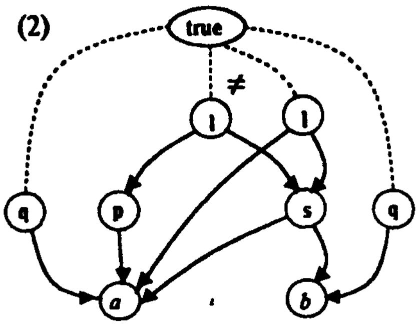
$$\text{Lemma 1: } \mathrm{q}(a) \wedge \mathrm{q}(b) \supset \mathrm{l}(\mathrm{p}(a), \mathrm{s}(a,b)).$$

When Algorithm D is applied to the negation of Lemma 1, it constructs the E-graph (1).

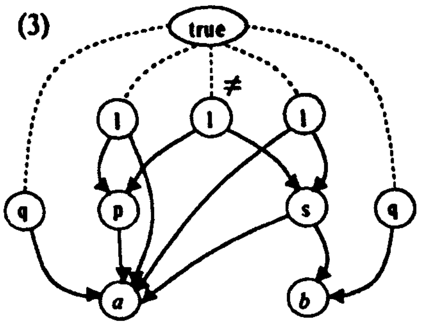
(1)



(2)



(3)



There are no close matches for axiom 3 in the graph (1). If, for example, x and z were instantiated to $\mathrm{p}(a)$ and $\mathrm{s}(a,b)$ respectively, in order to satisfy the definition of "close" for x and z, then, since there are no terms of the form $\mathrm{l}(t, \mathrm{s}(a,b))$ or $\mathrm{l}(\mathrm{p}(a), t)$ represented in the ground graph, there is no way to instantiate y that satisfies the definition.

To find a close match for axiom 2, x must be instantiated to a term that is a first argument of s. Thus there is only one close match for axiom 2, that in which x and y are instantiated to a and b respectively. This match does not cause a split, since $\mathrm{q}(b) = \mathrm{true}$ is explicitly represented in the graph. Thus, after this match is added, the graph becomes as shown in (2).

There are two close matches for axiom 1 in this graph; one is relevant to the proof and one is not. Neither of them cause splits. We will only add the one that is relevant to the proof; the result is shown in (3).

Now there is a unique close match for axiom 3: the match obtained by substituting $\mathrm{p}(a)$ for x, a for y, and $\mathrm{s}(a,b)$ for z. Assuming this instance of axiom 3 causes a contradiction and completes the proof.

Formulas like axiom 3 present a classic difficulty to matchers using "backwards chaining" stategies, because of the variable y that appears in the "hypothesis" of the axiom but not in the "conclusion". The heuristic definition of a close match used here does what is usually the right thing with the transitivity axiom: it uses the axiom as soon as there are compatible instances of any two of the three literals.

Example 2: Suppose that we have assumed or previously proved:

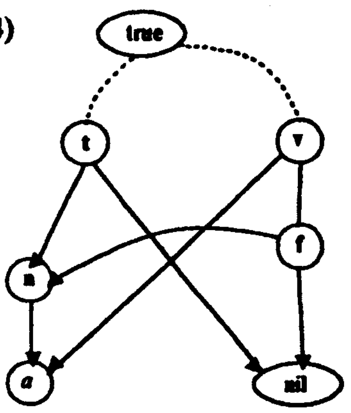
Axiom 1: $(\forall x\, y)\ \mathrm{v}(\mathrm{n}(x), y) = \mathrm{v}(x, y)$.
Axiom 2: $(\forall x)\ \mathrm{p}(\mathrm{n}(x))$.
Axiom 3: $(\forall x\, y)\ \neg \mathrm{t}(x,y) \wedge \mathrm{p}(x) \supset \mathrm{f}(x,y) \neq \mathrm{nil}$.
Axiom 4: $(\forall x\, y)\ \mathrm{p}(x) \wedge \mathrm{f}(x,y) \neq \mathrm{nil} \supset \neg \mathrm{v}(x, \mathrm{f}(x,y))$.

and from these assumptions we try to prove:

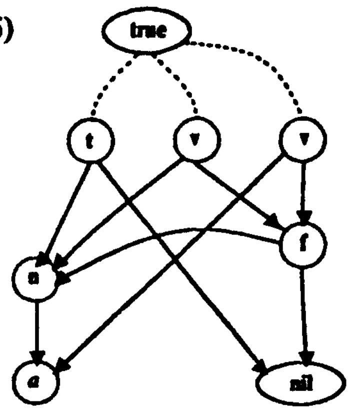
$$\text{Lemma 1: } \mathrm{v}(a, \mathrm{f}(\mathrm{n}(a), \mathrm{nil})) \supset \mathrm{t}(\mathrm{n}(a), \mathrm{nil}).$$

When Algorithm D is applied to the negation of the lemma, it constructs the E-graph (4).

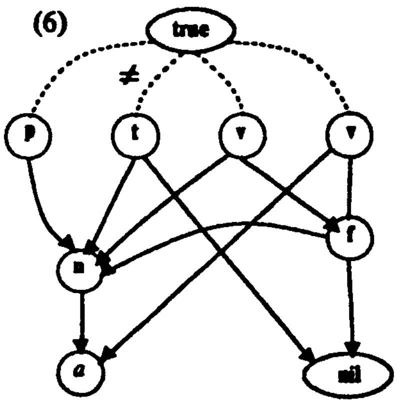
(4)



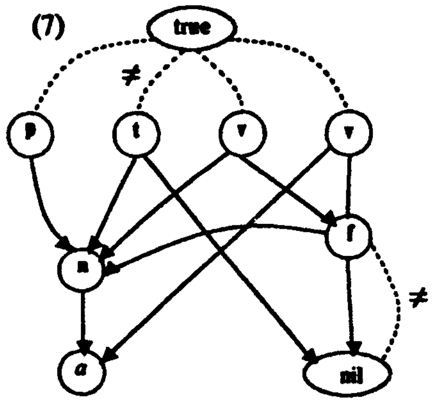
(5)



(6)



(7)



There is only one close match in this graph for axiom 1: that in which a and f(n(a), nil) are substituted for x and y respectively. (This is a daring match, since intuitively it uses axiom 1 "backwards.") The graph after the match is added is shown in (5). Suppose that axiom 2 is considered by the matcher next. The only close match for this axiom is obtained by instantiating x to a; after adding this match we get the graph (6). Now the graph contains a "perfect" match for axiom 3, namely $\neg$ t(n(a), nil) $\wedge$ p(n(a)) $\supset$ f(n(a), nil) $\neq$ nil. This instance is a clause, not a single literal, but the graph has enough information to contradict every literal in the clause except the last; therefore this last literal can be assumed in the graph, as shown in figure (7). Now there is a perfect match for axiom 4, namely p(n(a)) $\wedge$ f(n(a), nil) $\neq$ nil $\supset$ $\neg$ v(n(a), f(n(a), nil)). All of the literals of this instance are directly contradicted in the graph, so finding this instance completes the proof of the lemma.

The conventional top-down matching algorithm can be adapted to the E-graph, but its worst-case behavior becomes exponential. In fact, Kozen [35] has shown that the problem of determining whether a node of an E-graph is an instance of a pattern is NP-complete. This is the price that must be paid for dealing squarely with equality. Experiance with the Stanford Pascal Verifier suggests that the extra cost of matching in the E-graph is justified by the improved results.

3. Recursion and Induction

Many program specifications of practical importance cannot be formulated in classical first-order logic. Therefore, in sections 15 through 20, we discuss the extention of first-order logic by partial recursive function definitions, and the inductive theorem-proving techniques that are required to deal with them.

15. Examples

It is more difficult to verify invariants of programs that manipulate linked data structures than to verify invariants of programs that use only sequential data structures. Some people think the difficulty arises from assignments to dereferenced pointers, since to analyse the effect of such an assignment, one must determine which pointers have identical referents. But this is no more difficult than dealing with array assignments—in fact, the assignment $a(p) \leftarrow x$ may be viewed either as changing the pth field of the array a, or the a field of the record pointed to by p; its predicate transformation semantics are the same in the two cases. A sharp distinction between pointers and other indices is required by Algol-like languages in order to make them compilable without a theorem-prover, but to a verifier pointers and array indices are essentially the same. The satisfiability procedure for the theory of arrays described in section 10 can be viewed as a satisfiability procedure for a theory of pointers, if one chooses.

Why then is it more difficult to verify invariants of programs that manipulate linked structures? The answer is in the nature of the invariants. A typical invariant for a sequential structure is universally quantified over some range of indices; for example, a typical invariant about an array $a(1), \ldots, a(n)$ might be $(\forall i)\ 1 \leq i \leq j \supset a(i) < x$. Such invariants are first-order statements that can be verified by matching together with arithmetic reasoning. But a typical invariant about a linked structure concerns a sequence of values connected by link fields; for example an invariant might be "the sequence a, $l(a)$, $l(l(a))$, $\ldots$, contains nil." As mentioned in section 3, the class of such invariants is not compact, hence first-order methods of definition and inference are inadequate to deal with them. Thus, we cannot expect to verify such invariants without a formal system that supports inductive definitions and proofs. Sections 15 through 20 describe such a system.

Of course, induction is needed for other things besides verifying linked list programs, and several systems supporting induction have been designed and implemented. The system of Boyer and Moore [7] has the most impressive mechanical theorem-prover. Unfortunately, their system prohibits partial recursive function definitions, which, as we will see, are very useful in describing the invariants of programs that manipulate linked data structures. Thus our goal is to extend the work of Boyer and Moore to handle partial recursive functions.

This section contains two examples of the use of partial recursive functions in verifying properties of programs that manipulate linked data structures. The first example is a correctness verification for a program that reverses a linked list. It is perhaps a rather pointless verification, since the specifications are not significantly simpler than the program, but it was chosen to illustrate the most important differences between theorem proving with partial functions and theorem-proving with total functions. The second example is the specification of the trace-and-mark phase of a garbage collector; it illustrates the use of non-determinism.

The list reversal program uses three variables a, b, and l. The variable l is a function of one argument that represents the link fields: $l(n)$ is the record pointed to by the link field of the record n (unless it is the constant nil, which marks the end of the list); assignments to $l(n)$ change the contents of the link field of n. The program destructively reverses the list a and makes b point to the new first element. Here are informal specifications for the program:

> When entering **rev**, the sequence a, $l(a)$, $l(l(a))$, $\ldots$, contains nil. Also, $a = a_0$ and $l = l_0$.
>
> When exiting **rev**, the original sequence a_0, $l_0(a_0)$, $l_0(l_0(a_0))$, $\ldots$, (until nil), is the reverse of the sequence b, $l(b)$, $l(l(b))$, $\ldots$, (until nil).
>
> The only global variables changed by **rev** are a, b, and l.

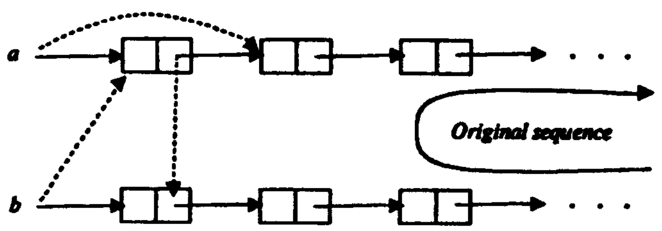
"Until nil" is short for "up to but not including the first nil." The use of the phrase implies the unstated specification that the given list contain nil.

The program is written as two procedures:

Procedure **rev**() $\equiv [\![\, b \leftarrow \text{nil};\ \textbf{revloop}() \,]\!]$;

Procedure **revloop**() $\equiv [\![\, a = \text{nil} \Rightarrow [\![\;]\!];\ (b, a, l(a)) \leftarrow (a, l(a), b);\ \textbf{revloop}() \,]\!]$.

Here is an illustration of the general step of **revloop**:



In this illustration, the right field of a node n contains an arrow pointing to the node $l(n)$; the left field contains data values that do not concern us. Notice that the specifications require that the list be reversed "destructively", since they refer to the sequence of cells and not to the sequence of data values.

Here are the specifications for **revloop**:

> When entering **revloop**, the sequence a_0, $l_0(a_0)$, $\ldots$, (until nil) is the concatenation of the reverse of the sequence b, $l(b)$, $\ldots$ (until nil) with the sequence a, $l(a)$, $\ldots$, (until nil). Furthermore, the latter two sequences have no common elements.
>
> When exiting **revloop**, the sequence a_0, $l_0(a_0)$, $l_0(l_0(a_0))$, $\ldots$, (until nil), is the reverse of the sequence b, $l(b)$, $l(l(b))$, $\ldots$, (until nil).

Finite sequences will be regarded formally as nests of ordered pairs; for example $(a, (b, (c, \text{nil})))$ is regarded as the sequence of a, b, and c. The Lisp functions **car**, **cdr**, and **cons** will be used for operating on ordered pairs. The constant nil represents the empty sequence. Using this convention we define a recursive function **sequence** such that **sequence**(x, f) returns the sequence x, $f(x)$, $f(f(x))$, $\ldots$, (until nil):

Function **sequence**$(p, f) \equiv [\![\, p = \text{nil} \Rightarrow \text{nil};\ \textbf{cons}(p, \textbf{sequence}(f(p), f)) \,]\!]$.

We also define the function **reverse** that returns the concatenation of the reversal of its first argument with its second argument:

$$\text{Function } \mathbf{reverse}(x,y) \equiv [\![x = \mathbf{nil} \Rightarrow y;\ \mathbf{reverse}(\mathbf{cdr}(x), \mathbf{cons}(\mathbf{car}(x), y))]\!].$$

For example, $\mathbf{reverse}((x,(y,\mathbf{nil})),(u,(v,\mathbf{nil}))) = (y,(x,(u,(v,\mathbf{nil}))))$. Note that the reversal of the sequence s is given by $\mathbf{reverse}(s,\mathbf{nil})$.

Finally, we define the function **reach**; $\mathbf{reach}(x,y,f)$ returns **true** or **false** according to whether the sequence x, $f(x)$, $f(f(x))$, ... contains an occurence of y before it contains an occurence of **nil**:

$$\text{Function } \mathbf{reach}(x,y,f) \equiv [\![x = y \Rightarrow \mathbf{true};\ x = \mathbf{nil} \Rightarrow \mathbf{false};\ \mathbf{reach}(f(x),y,f)]\!].$$

Notice that the $x = y$ test is made before the $x = \mathbf{nil}$ test, so $\mathbf{reach}(x,\mathbf{nil},f)$ will evaluate to **true** if and only if the sequence x, $f(x)$, $f(f(x))$, ... contains an occurence of **nil**.

Formally, **reach** is regarded as a function, not a predicate—**true** and **false** are ordinary constants. But the term $\mathbf{reach}(a,b,c)$ will be used to abbreviate the formula $\mathbf{reach}(a,b,c) = \mathbf{true}$ if the context makes it clear that a formula, not a term, is required.

The computations specified by these recursive definitions do not always terminate. If a and b are terms, "a halts" means the evaluation of a terminates; "a loops" means the evaluation of a does not terminate; $a \simeq b$ means that a and b either both loop, or both halt with the same value; and $a = b$ means that a and b both halt with the same value. Thus, if a is a term that loops, $a = a$ is false, and $a \neq a$ is true. Of course, these definitions only make sense with respect to a fixed evaluation rule. The formal definition of the evaluation rule is in section 16; for now it is enough to say that it is conventional call-by-value evaluation. Thus if a loops, so do $[\![a \Rightarrow 1; 1]\!]$ and $0 \times a$.

To formalize the entry specification for **revloop** we define the predicate **disjoint**:

$$\text{Predicate } \mathbf{disjoint}(x,y,f) \equiv (\forall z)\ \mathbf{reach}(x,z,f) \land \mathbf{reach}(y,z,f) \supset z = \mathbf{nil}.$$

These functions and predicate are sufficient to formalize the specifications of **rev** and **revloop**. The specifications for **rev** are:

Entering **rev**, $a = a_0 \land l = l_0 \land \mathbf{reach}(a,\mathbf{nil},l)$.
Exiting **rev**, $\mathbf{reverse}(\mathbf{sequence}(b,l),\mathbf{nil}) = \mathbf{sequence}(a_0,l_0)$.

Because of the convention that $a = b$ is false if either a or b loop, the exit specification implies that the applications of **sequence** halt; notice that this corresponds with the understanding that the use of the phrase "until **nil**" implies an occurence of **nil**.

The entry specification for **revloop** is

Entering **revloop**,
$\mathbf{reverse}(\mathbf{sequence}(b,l),\mathbf{sequence}(a,l)) = \mathbf{sequence}(a_0,l_0) \land \mathbf{disjoint}(a,b,l)$.

When Peter Deutsch saw this formula, he immediately pointed out (to the author's surprise) that the second conjunct of the formula is a consequent of the first. For any common element of $\mathbf{sequence}(a,l)$ and $\mathbf{sequence}(b,l)$ would appear twice in $\mathbf{reverse}(\mathbf{sequence}(b,l),\mathbf{sequence}(a,l))$. But if the sequence a_0, $l_0(a_0)$, ... has a repeated element before it has an occurence of **nil**, then it has no occurence of **nil** at all, in which case $\mathbf{sequence}(a_0,l_0)$ loops and the first conjunct of the formula is false. However, the proof that the procedures meet their specifications is simpler if the redundant conjunct is included.

The exit specification for revloop is the same as the exit specification of rev:

$$\text{Exiting revloop, } \mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{nil}) = \mathbf{sequence}(a_0, l_0).$$

Let us now follow the symbolic evaluator along one interesting path that starts at the top of rev. At entry to rev, we assume:

$$\begin{aligned}&a = a_0\\&l = l_0\\&\mathbf{reach}(a, \mathbf{nil}, l)\end{aligned}$$

Then b is assigned nil, and the entry assertion for revloop must be proved. There are two conjuncts to prove, the most interesting of which is the first. To prove the first conjunct, its negation is assumed:

$$\begin{aligned}&a = a_0\\&l = l_0\\&\mathbf{reach}(a, \mathbf{nil}, l)\\&\mathbf{reverse}(\mathbf{sequence}(\mathbf{nil}, l), \mathbf{sequence}(a, l)) \neq \mathbf{sequence}(a_0, l_0)\end{aligned}$$

The essential reason that these assumptions are inconsistent is shown by the simple derivation:

$$\begin{aligned}&\mathbf{reverse}(\mathbf{sequence}(\mathbf{nil}, l), \mathbf{sequence}(a, l))\\&\quad \simeq \mathbf{reverse}(\mathbf{nil}, \mathbf{sequence}(a, l))\\&\quad \simeq \mathbf{sequence}(a, l)\\&\quad \simeq \mathbf{sequence}(a_0, l_0).\end{aligned}$$

The matcher essentially performs this derivation, using the definitions of sequence and reverse. But the derivation is not yet a proof: notice the use of "$\simeq$" instead of "$=$." If all the terms in the derivation chain loop, then the disequality will be satisfied, since an equality is defined to be false if both sides loop. To complete the proof we need to show that one of the four terms in the derivation halts, from which it will follow that they all halt. That $\mathbf{sequence}(a, l)$ halts is an obvious consequence of $\mathbf{reach}(a, \mathbf{nil}, l)$, but it is not clear how to make the inference mechanically. This fact, although obvious, is not a consequence of any finite number of instances of the definitions of sequence and reach, so it cannot be proved by the matcher; it can only be proved by an induction argument. The system will not attempt any inductive proofs in the middle of symbolic evaluation; any facts that are needed in a verification and that require inductive proofs must be formulated explicitly as lemmas. Thus for the current example, the file given to the verifier must contain the following lemma:

$$\text{Lemma 1: } (\forall x\, f): \quad \mathbf{reach}(x, \mathbf{nil}, f) \Leftrightarrow \mathbf{sequence}(x, f) \text{ halts.}$$

(The $\Leftrightarrow$ could be changed to $\supset$ as far as the current path goes, but the other direction is needed later on.) Since $\mathbf{reach}(a, \mathbf{nil}, l)$ is in the antecedent, the matcher deduces from this lemma that $\mathbf{sequence}(a, l)$ halts, which completes the derivation above.

The system's induction rule is described in section 20. We illustrate it by describing the inductive proof of lemma 1: Consider this half of the lemma:

$$(\forall x\, f): \quad \mathbf{reach}(x, \mathbf{nil}, f) \supset \mathbf{sequence}(x, f) \text{ halts.}$$

It is trivially true if $\mathbf{reach}(x, \mathbf{nil}, f)$ loops, so we may assume that the evaluation of this term halts, and prove the formula by induction on the depth of recursion reached in the evaluation. If there are no recursive calls in the evaluation of $\mathbf{reach}(x, \mathbf{nil}, f)$, then $x = \mathbf{nil}$, so $\mathbf{sequence}(x, f)$ halts. If there is a recursive call, then $x \neq \mathbf{nil}$, the recursive call is $\mathbf{reach}(f(x), \mathbf{nil}, f)$, and the call must return true. Therefore, by induction, we have that $\mathbf{sequence}(f(x), f)$ halts. But if $x \neq \mathbf{nil}$, $\mathbf{sequence}(x, f) \simeq \mathbf{cons}(x, \mathbf{sequence}(f(x), f))$; a call to **cons** will halt if both arguments do, so the proof is complete.

The proof of the other half of lemma 1,

$$\mathbf{sequence}(x, f) \text{ halts} \supset \mathbf{reach}(x, \mathbf{nil}, f),$$

is similar, but the induction is on the depth of recursion of $\mathbf{sequence}(x, f)$, since this is the term that must halt in order for the formula to be true. Of course, it happens that these two functions recurse in essentially the same way, so the two inductions are identical.

Four other lemmas are required in the verification of the list reversal program. Here is the actual input to the experimental verifier on this example:

Function $\mathbf{sequence}(p, f) \equiv \llbracket p = \mathbf{nil} \Rightarrow \mathbf{nil};\ \mathbf{cons}(p, \mathbf{sequence}(f(p), f)) \rrbracket$.
Function $\mathbf{reverse}(x, y) \equiv \llbracket x = \mathbf{nil} \Rightarrow y;\ \mathbf{reverse}(\mathbf{cdr}(x), \mathbf{cons}(\mathbf{car}(x), y)) \rrbracket$.
Function $\mathbf{reach}(x, y, f) \equiv \llbracket x = y \Rightarrow \mathbf{true};\ x = \mathbf{nil} \Rightarrow \mathbf{false};\ \mathbf{reach}(f(x), y, f) \rrbracket$.
Predicate $\mathbf{disjoint}(x, y, f) \equiv (\forall z) : \mathbf{reach}(x, z, f) \wedge \mathbf{reach}(y, z, f) \supset z = \mathbf{nil}$.
Entering **rev**, $a = a_0 \wedge l = l_0 \wedge \mathbf{reach}(a, \mathbf{nil}, l)$.
Exiting **rev**, $\mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{nil}) = \mathbf{sequence}(a_0, l_0)$.
rev changes a, b, l.
Procedure $\mathbf{rev}() \equiv \llbracket b \leftarrow \mathbf{nil};\ \mathbf{revloop}() \rrbracket$.
Entering **revloop**,
  $\mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{sequence}(a, l)) = \mathbf{sequence}(a_0, l_0) \wedge \mathbf{disjoint}(a, b, l)$.
Exiting **revloop**, $\mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{nil}) = \mathbf{sequence}(a_0, l_0)$.
revloop changes a, b, l.
Procedure $\mathbf{revloop}() \equiv \llbracket a = \mathbf{nil} \Rightarrow \llbracket\ \rrbracket;\ (b, a, l(a)) \leftarrow (a, l(a), b);\ \mathbf{revloop}() \rrbracket$.
Lemma 1: $(\forall x\, f) : \mathbf{reach}(x, \mathbf{nil}, f) \Leftrightarrow \mathbf{sequence}(x, f)$ halts.
Lemma 2: $(\forall x\, y\, f\, e) : \neg\,\mathbf{reach}(x, y, f) \supset \mathbf{sequence}(x, f\,^{(y)}_{e}) \simeq \mathbf{sequence}(x, f)$.
Lemma 3: $(\forall x\, y\, f) : \mathbf{reach}(x, y, f) \wedge y \neq \mathbf{nil} \supset \mathbf{reach}(x, f(y), f)$.
Lemma 4: $(\forall x\, f) : \mathbf{reach}(x, \mathbf{nil}. f) \wedge x \neq \mathbf{nil} \supset \neg\,\mathbf{reach}(f(x), x, f)$.
Lemma 5: $(\forall x\, y\, z\, f\, e) : \neg\,\mathbf{reach}(x, y, f) \supset \mathbf{reach}(x, z, f\,^{(y)}_{e}) \simeq \mathbf{reach}(x, z, f)$.

(It is obviously unacceptable to require so much labor just to verify that such a simple program meets such simple specifications. But the methods of this section can be improved.)

For the benefit of readers who are interested in following through the example to see why the other lemmas are required, we now consider the only other nontrivial path through the program; the path from the top of **revloop** to the recursive call. Other readers should skip to the last example in this section.

We will no longer give a stack of literals to represent the theorem prover's data structure, but describe the argument informally. The assignment $l(a) \leftarrow b$ is equivalent to $l \leftarrow l\,^{(a)}_{b}$, so to prove that the recursive entry to **revloop** satisfies the entry specification, the system assumes

$$\mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{sequence}(a, l)) = \mathbf{sequence}(a_0, l_0) \wedge a \neq \mathbf{nil} \tag{1}$$

and

$$\mathbf{disjoint}(a, b, l) \tag{2}$$

and from the assumptions deduces

$$\mathbf{reverse}(\mathbf{sequence}(a, l^{(a)}_{b}), \mathbf{sequence}(l(a), l^{(a)}_{b})) = \mathbf{sequence}(a_0, l_0) \tag{3}$$

and

$$\mathbf{disjoint}(b, l(a), l^{(a)}_{b}). \tag{4}$$

It will be proved below that the following two formulas are consequences of (1) and (2):

$$\mathbf{sequence}(b, l^{(a)}_{b}) = \mathbf{sequence}(b, l) \tag{5}$$

$$\mathbf{sequence}(l(a), l^{(a)}_{b}) = \mathbf{sequence}(l(a), l). \tag{6}$$

(It is illuminating to look at figure 1 to see what these formulas mean, namely, that the assignments to link fields have no effect outside of the immediate locality. Similar facts are required in almost all verifications involving linked data structures.) Given (5) and (6), it is easy to prove (3):

$$\begin{aligned}
&\mathbf{reverse}(\mathbf{sequence}(a, l^{(a)}_{b}), \mathbf{sequence}(l(a), l^{(a)}_{b})\\
&\quad \simeq \mathbf{reverse}(\mathbf{cons}(a, \mathbf{sequence}(l^{(a)}_{b}(a), l^{(a)}_{b})), \mathbf{sequence}(l(a), l^{(a)}_{b}))\\
&\quad \simeq \mathbf{reverse}(\mathbf{sequence}(b, l^{(a)}_{b}), \mathbf{cons}(a, \mathbf{sequence}(l(a), l^{(a)}_{b})))\\
&\quad \simeq \mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{cons}(a, \mathbf{sequence}(l(a), l)))\\
&\quad \simeq \mathbf{reverse}(\mathbf{sequence}(b, l), \mathbf{sequence}(a, l))\\
&\quad = \mathbf{sequence}(a_0, l_0).
\end{aligned}$$

It remains to prove (4), (5), and (6). Three more lemmas are required.

Lemma 2: $(\forall x\, y\, f\, e) : \neg \mathbf{reach}(x, y, f) \supset \mathbf{sequence}(x, f^{(y)}_{e}) \simeq \mathbf{sequence}(x, f)$.

That is, if y cannot be reached from x, changing $f(y)$ does not affect $\mathbf{sequence}(x, f)$. Notice that the lemma would be false if the "$\simeq$" were changed to "$=$", since the antecedent does not imply that the applications of sequence halt. This lemma suffices to prove (5): a is not nil, and obviously $\mathbf{reach}(a, a, l)$ is true; if $\mathbf{reach}(b, a, l)$ were also true, (2) would be contradicted. Since a cannot be reached from b, lemma 2 implies $\mathbf{sequence}(b, l^{(a)}_{b}) \simeq \mathbf{sequence}(b, l)$. The term on the right must halt, since it appears in (1); this proves (5).

Lemma 4: $(\forall x\, f) : \mathbf{reach}(x, \mathbf{nil}, f) \wedge x \neq \mathbf{nil} \supset \neg \mathbf{reach}(f(x), x, f)$.

That is, if nil can be reached from x, then x is non-circular. This lemma suffices to prove (6). By lemma 1 and (1), $\mathbf{reach}(a, \mathbf{nil}, l)$ is true, and $a \neq \mathbf{nil}$, so by lemma 4, it is impossible to reach a from $l(a)$; thus (6) follows by lemma 2.

Lemma 5: $(\forall x\, y\, z\, f\, e) : \neg \mathbf{reach}(x, y, f) \supset \mathbf{reach}(x, z, f^{(y)}_{e}) \simeq \mathbf{reach}(x, z, f)$.

That is, if y cannot be reached from x, changing $f(y)$ does not affect what can be reached from x. This lemma suffices to prove (4). Suppose (4) is false; then a z exists such that

$$z \neq \mathbf{nil} \wedge \mathbf{reach}(b, z, l^{(a)}_{b}) \wedge \mathbf{reach}(l(a), z, l^{(a)}_{b}).$$

It has already been shown that a cannot be reached by following l-links either from $l(a)$ or from b; thus lemma 5 implies

$$z \neq \mathbf{nil} \land \mathbf{reach}(b, z, l) \land \mathbf{reach}(l(a), z, l).$$

Since $a \neq$ nil, $\mathbf{reach}(l(a), z, l)$ implies $\mathbf{reach}(a, z, l)$; this contradicts $\mathbf{disjoint}(a, b, l)$, and completes the verification of the path.

The proofs of lemmas 2 and 5 are almost identical to the proof of lemma 1, and will not be described. The proof of lemma 4 is a bit more interesting; it is in order to prove lemma 4 that lemma 3 is required. To prove

$$\mathbf{reach}(x, \mathbf{nil}, f) \land x \neq \mathbf{nil} \supset \neg\mathbf{reach}(f(x), x, f)$$

the system uses induction on the depth of recursion of $\mathbf{reach}(x, \mathbf{nil}, f)$. The base case has the condition $x = \mathbf{nil}$, in which case the formula is vacuously true. The induction step is

$$\begin{aligned} &x \neq \mathbf{nil} \land \mathbf{reach}(x, \mathbf{nil}, f) \\ &\quad \land (\mathbf{reach}(f(x), \mathbf{nil}, f) \land f(x) \neq \mathbf{nil} \supset \neg\mathbf{reach}(f(f(x)), f(x), f)) \\ &\supset \neg\mathbf{reach}(f(x), x, f). \end{aligned}$$

Since $x \neq$ nil, the first formula on the second line unfolds to $\mathbf{reach}(f(x), \mathbf{nil}, f)$. Since the consequent of the whole formula is true if $f(x) = \mathbf{nil}$, it may be assumed that $f(x) \neq \mathbf{nil}$; thus the antecedent of the implication on the first line is true, and the formula simplifies to:

$$\begin{aligned} &x \neq \mathbf{nil} \land f(x) \neq \mathbf{nil} \land \mathbf{reach}(f(x), \mathbf{nil}, f) \land \neg\mathbf{reach}(f(f(x)), f(x), f) \\ &\quad \supset \neg\mathbf{reach}(f(x), x, f). \end{aligned}$$

This is obvious, since if, contrary to the consequent, x could be reached from $f(x)$, then since neither of these are nil, $f(x)$ could be reached from $f(f(x))$. But to prove this requires another lemma, called lemma 3 above:

$$\text{Lemma 3: } (\forall x\, y\, f): \quad \mathbf{reach}(x, y, f) \land y \neq \mathbf{nil} \supset \mathbf{reach}(x, f(y), f).$$

The proof of lemma 3 is by the obvious induction. It must be placed before lemma 4 in the file that is given to the experimental verifier, since it is used in the proof of lemma 4.

It would, of course, be a great step forward to find mechanical techniques that discover the lemmas required in the proof of the paths through the program, so that a programmer would not have to figure out the five lemmas that are needed. Boyer and Moore [7,8,9] and Aubin [5] describe heuristics for strengthening a conjecture in order to make it provable by induction. These heuristics could probably be used to generate lemma 1 automatically from the path that required it, or lemma 3 from the induction step for lemma 4; but they strengthen a formula into a single lemma, not into a set of lemmas, so they are not able to discover the four lemmas needed to prove the path through **revloop**.

It would be possible to include a satisfiability procedure for the theory of reach. However, **reach** is not as fundamental as equality, numbers or ordered pairs, and it would not be worthwhile to include a satisfiability procedure for **reach** it unless turned out to be useful for many programs.

Another possible approach to this problem is mentioned in section 20.

Although first-order logic extended with partial recursive functions is very expressive, a considerable gain in expressive power is obtained by allowing *nondeterministic* partial recursive functions. The form of nondeterminism we consider is the following: we write $A \veebar B$ to mean the "dovetailed or of A and B," that is, the result of evaluating both terms at once, working on them alternately, and returning **true** if either of the two returns **true**. Used in a logical formula (such as an entry or exit specification), $A \veebar B$ is equivalent to $A \vee B$, but they have different meanings inside a recursive function definition. We have not yet discussed formally what $A \vee B$ means inside a recursive function definition; as we shall see in section 16, it means to evaluate A, returning true if A returns true; and otherwise to evaluate B and return the result. Thus in a recursive function definition, $A \vee B$ loops whenever A does, while $A \veebar B$ halts whenever either A or B halts with **true**.

As an illustration of the use of $\veebar$, we will give formal specifications for a garbage-collection marking algorithm. The algorithm operates on a set of nodes each of which contains four fields: the link fields a and b, which are assumed to point at other nodes; the one-bit *atom* field, which is set if the node represents an "atom", that is, a node whose a and b fields are to be ignored; and the one-bit gc field, which is initially clear for all nodes. The formal specifications state that the procedure **mark** sets the gc field in all nodes that can be reached from **root** (a global constant) by following a and b fields through nodes whose *atom* field is not set. The specifications include a non-deterministic recursive definition of the predicate **reachable**:

Function $\textbf{reachable}(x, y, atom, a, b) \equiv$
$\llbracket x = y \Rightarrow \textbf{true};$
$atom(x) \Rightarrow \textbf{false};$
$\textbf{reachable}(a(x), y, atom, a, b) \veebar \textbf{reachable}(b(x), y, atom, a, b) \rrbracket;$

Predicate $\textbf{gczero}(gc) \equiv (\forall x)\ gc(x) = \textbf{false};$

Predicate $\textbf{gcreach}(gc, atom, a, b) \equiv (\forall x)\ gc(x) \Leftrightarrow \textbf{reachable}(\textbf{root}, x, atom, a, b);$

Entering **mark**, $a = a_0 \wedge b = b_0 \wedge atom = atom_0 \wedge \textbf{gczero}(gc);$

Exiting **mark**, $a = a_0 \wedge b = b_0 \wedge atom = atom_0 \wedge \textbf{gcreach}(gc, atom, a, b);$

The recursive definition for **reachable** might be paraphrased: (a) x is reachable from x, (b) if x is not an atom, and y is reachable from $a(x)$ or from $b(x)$, then y is reachable from x, and (c) nothing is reachable from x unless it is because of (a) and (b).

The procedure **mark** modifies the list structure temporarily. To require that the modifications be temporary, the entry specification labels the original values of a, b, and *atom* as a_0, b_0, and $atom_0$ and the exit specification specifies that their final values equal their original values.

One might try to replace the definition of **reachable** in the example above by the three axioms:

Axiom 1: $(\forall x\ atom\ a\ b)\ \textbf{reachable}(x, x, atom, a, b).$

Axiom 2: $(\forall x\ y\ atom\ a\ b)$
$\neg atom(x) \wedge \textbf{reachable}(a(x), y, atom, a, b) \supset \textbf{reachable}(x, y, atom, a, b).$

Axiom 3: $(\forall x\ y\ atom\ a\ b)$
$\neg atom(x) \wedge \textbf{reachable}(b(x), y, atom, a, b) \supset \textbf{reachable}(x, y, atom, a, b).$

These axioms state obvious properties of the intuitive predicate **reachable**, so to assume them introduces no appreciable danger of error. But they are not sufficient to define the intuitive predicate, since it is consistent with them to assume that anything is reachable from anything.

Using them alone it would not be possible to prove that the marking algorithm marked everything reachable.

16. Formal semantics for the specification language

The last fifteen sections have used recursive procedures, assignment statements, variables over functions, pointers, and arrays, predicates defined by quantification and recursion, inductive proofs, non-determinism, numbers, and logical formulas as specifications for programs. These notions have been treated formally before, most of them in several ways. But, except for numbers and predicates defined by quantification, the notions do not have widely-accepted standard formalizations. Therefore, sections 16 and 17 are necessary to define them precisely.

We will use techniques from many sources. The basic method of defining specifications for programs is due to Floyd [24]. The definition of the semantics of recursive functions is based on the definition in Kleene [31], but is more in the style of McCarthy [39].

The main problem in defining the semantics of the specification language is deciding what to do about the recursive definitions. There are several ways to deal with them.

The lambda calculus of Church, for which models have been constructed by Dana Scott [49], assigns semantics to recursive definitions in such a way that the recursive functions become values; for example a variable can have a recursive function as a value. The system defined here does not have this generality.

Given a first order theory T and a set of recursive function definitions R, Robert Cartwright and John McCarthy [11] extend T to a theory T' by adding the function symbols defined in R and the constant $\perp$, and construct a set S of induction schemes, such that the theorems provable in T' by using induction schemes in S are true in some standard model in which the recursive functions are evaluated by some evaluation rule and $\perp$ is the value of a function application that loops. They describe the construction for several different evaluation rules.

In the "formalism of recursive functions" of Kleene [31], a syntactic characterization of a halting computation is given, and $t = u$ is taken to denote that both terms t and u halt with the same value. Thus the semantics of recursive functions are defined directly instead of modeled in first-order logic, with the result that $=$ behaves differently than conventional equality. Kleene's method will be used here, but his definitions will be generalized to allow recursive functions to compute with values from an arbitrary first-order theory, instead of with the natural numbers only.

The syntax of the specification language is that of the classical first-order calculus of function symbols, with equality as the only predicate.

More precisely, a *variable* (of the formal system) is a sequence of one or more lower-case italic letters, possibly containing hyphens, and possibly primed or subscripted with a numeral, such as x_6 or *tax-rate*. A *function symbol* is a sequence of lower-case bold letters, possibly containing hyphens, such as **nil** or **if-then-else**. Each function symbol has an associated *arity*, which is a non-negative integer. The function symbols are divided into two groups, the *recursive* function symbols and the *non-recursive* function symbols. (No typographical convention distinguishes recursive from non-recursive function symbols, or indicates a function symbol's arity—we will make appropriate assumptions about individual function symbols as we use them.) A *term* is defined as follows: (a) a variable is a term; (b) an expression of the form $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol and the t's are terms, is a term; (c) nothing is a term unless it is because of (a) and (b). The empty parentheses that follow 0-ary function symbols will usually be dropped; thus **nil**( ) will be written **nil** if the context makes it clear that a term, not a function symbol, is intended. An *atomic formula* is an expression of the form $t = u$, where t and u are terms. A *formula* is defined by: (a) an atomic formula is a formula; (b) an expression of one of the forms $(F \wedge G)$, $\neg F$, or $(\forall x) F$, where F and G are formulas and x is a variable, is a formula; (c) nothing is a formula unless it is because of (a) and (b). (We will often omit the parentheses from $(F \wedge G)$, and use other informal notation, such as $a \leq b \leq c$ for $a \leq b \wedge b \leq c$.)

The letters a, b, ..., A, B, ... will be used as "meta-variables" ranging over expressions of the formal system.

Expressions involving $\vee$, $\supset$, $\Leftrightarrow$, and $\exists$ are as usual regarded as abbreviations for equivalent formulas involving only $\wedge$, $\neg$, and $\forall$. For example, $F \vee G$ is an abbreviation for $\neg(\neg F \wedge \neg G)$. Furthermore, if a term t is used where a formula is expected, it is an abbreviation for the formula $t =$ **true**; thus functions can be used as though they were predicates.

A *recursive function definition* is an expression of the form

$$f(x_1, \ldots, x_n) \equiv B$$

where f is an n-ary recursive function symbol, the x's are distinct variables, and B is a term, the *body* of the recursive function. (Free variables are not prohibited in B, although we will have no use for such free variables in this paper.) It may seem odd that the body of the definition is an ordinary term, since in most recursive function definitions the body will be a conditional expression. Conditional expressions involving $[\![$, $\Rightarrow$, and $]\!]$ are regarded as abbreviations for equivalent terms involving the function symbol **if-then-else**, in the obvious way: the conditional expression

$$[\![p_1 \Rightarrow e_1;\ p_2 \Rightarrow e_2;\ \ldots\ ;\ p_n \Rightarrow e_n;\ e_{n+1}]\!]$$

is an abbreviation for the term

$$\textbf{if-then-else}(p_1, e_1, \textbf{if-then-else}(p_2, e_2, \ldots, \textbf{if-then-else}(p_n, e_n, e_{n+1})\cdots)).$$

(A conditional statement like $[\![A;\ B]\!]$ also abbreviates a term, as defined in section 17, but it would be nonsense to include such a term in a recursive function definition.)

The symbols $\wedge$, $\vee$, $\neg$, and $\supset$ may appear in terms, where they are regarded as abbreviations for terms involving **if-then-else**, according to the following rules:

$A \wedge B$	abbreviates	**if-then-else**$(A, B,$ **false**$)$
$A \vee B$	abbreviates	**if-then-else**$(A,$ **true**$, B)$
$\neg A$	abbreviates	**if-then-else**$(A,$ **false**, **true**$)$
$A \supset B$	abbreviates	**if-then-else**$(A, B,$ **true**$)$
$A \Leftrightarrow B$	abbreviates	**if-then-else**$(A, B, \neg B)$.

(Obvious assumptions about the arity of common function symbols will not be mentioned explicitly; thus **if-then-else** is 3-ary and **true** and **false** are 0-ary.)

Finally, an expression of the form $a = b$, if used where a term is expected (such as in the body of a recursive definition), is an abbreviation for the term **equal**(a, b); and the expression $A \,\forall\, B$ is an abbreviation for **dovetail**(A, B). The arithmetic infix operators and predicates are also regarded as abbreviations for function symbols, but we will not list them here.

According to these conventions, the nonsensical function definition

$$\mathrm{p}(x, y) \equiv [\![x = \mathbf{nil} \Rightarrow \mathbf{nil};\ x \neq y \Rightarrow \mathbf{nil};\ x \wedge y]\!]$$

is an abbreviation for

```
p(x, y) ≡
    if-then-else(equal(x, nil),
                 nil,
                 if-then-else(if-then-else(equal(x, y), false, true),
                              nil,
                              if-then-else(x, y, false))).
```

Thus the meaning of many symbols depends on their context. Used as a formula, the expression $a = b \vee c$ is an abbreviation for $a = b \vee c =$ **true**; used as a term, it abbreviates **if-then-else**(**equal**(a, b), **true**, c). This allows conventional notation to be used throughout while preserving the important technical distinction between terms and formulas.

Interpretations are defined in the conventional way: an *interpretation* Ψ with *universe* U is a map that assigns to each variable x an element $\Psi(x)$ of U and to each n-ary non-recursive function symbol f a map $\Psi(f) : U^n \rightarrow U$, such that $\Psi(\mathbf{true})() \neq \Psi(\mathbf{false})()$, and $\Psi(\mathbf{equal})(u, v) = \Psi(\mathbf{true})()$ if and only if $u = v$. (Since **true** is 0-ary, $\Psi(\mathbf{true})$ is a mapping from U^0 to U, so we must write $\Psi(\mathbf{true})()$, not $\Psi(\mathbf{true})$, to denote the element of U representing truth.)

(It is unacceptable to use non-finitary definitions (like the one above) in the foundations of a programming system, but like most mathematics students, the author was addicted to set theory in college. Presumably with more care one could justify the theorems in sections 19 and 20 b·· purely finitary methods.)

In the definition of the semantics of the conventional predicate calculus, the next step is to extend interpretations over terms in the natural way. It is not so simple in this system. When a term t contains recursive functions, the most natural choice for $\Psi(t)$ is the result of evaluating t by some evaluation rule, which must be defined. We will use an ordinary call-by-value evaluation rule, although other rules might be chosen instead.

Instead of defining an evaluation algorithm, it is technically convenient to define a syntactic object (called an *evaluation tree*) that is a record of a complete evaluation, like a sequence of Turing machine instantaneous descriptions. Then the statement "the term t halts when evaluated in the interpretation Ψ using a set of recursive definitions R" can be formalized "there exists an evaluation tree for t, under Ψ and R."

The notation $a\binom{e}{i}$ used previously for arrays and linked structures will be used hereafter for functions in general. In particular, $\Psi\binom{v}{u}$ is the interpretation that differs from Ψ only at the variable v, which it maps to u. In an evaluation tree the interpretation Ψ defines the non-recursive functions and also serves as a "name-value stack" for the evaluator; the stack resulting from Ψ by binding the variable v to the value $u \in U$ is $\Psi\binom{v}{u}$.

If Ψ is an interpretation and R is a set of recursive definitions such that no function symbol is defined twice in R, then (Ψ, R) is a *context*. An *evaluation tree* for a given term *in* a given context with a certain *result* is defined as follows: (a) if (Ψ, R) is a context and t is a term, then the tuple $(t, u, T_1, \ldots, T_n)$ is an evaluation tree for t in (Ψ, R) with result u if any of the following five conditions hold:

- t is a variable, $n = 0$, and $u = \Psi(t)$;

  (I.e., to evaluate a variable, return its value in the current context.)

- t is of the form **if-then-else**(p, a, b), $n = 2$, T_1 is an evaluation tree for p in (Ψ, R), and one of the following two conditions holds: (2a) the result of T_1 is $\Psi(\mathbf{true})()$, T_2 is an evaluation tree for a in (Ψ, R), and u is the result of T_2; or (2b) the result of T_1 is not $\Psi(\mathbf{true})()$, T_2 is an evaluation tree for b in (Ψ, R), and u is the result of T_2;

(I.e., to evaluate **if-then-else**(p, a, b)*, evaluate* p*. If the result is* Ψ(**true**)( )*, return the result of evaluating* a*, else return the result of evaluating* b*.)*

- t is of the form **dovetail**(a, b), $n = 2$, and either (3a) T_1 and T_2 are evaluation trees for a and b respectively, neither have result Ψ(**true**)( ), and $u = \Psi$(**false**)( ); or (b) either T_1 is an evaluation tree for a in (Ψ, R) with result Ψ(**true**)( ), or T_2 is an evaluation tree for b in (Ψ, R) with result Ψ(**true**)( ), and in either case, $u = \Psi$(**true**)( );

  (I.e., to evaluate $a \mathbin{\dot\vee} b$*, evaluate* a *and* b *separately, working on them alternately; return* **true** *if either returns* **true**, *return* **false** *if both return non-true values.)*

- t is of the form $f(t_1, \ldots, t_n)$, f is not **if-then-else** or **dovetail**, f is non-recursive, each T_i is an evaluation tree for t_i in (Ψ, R), and, letting u_i be the result of T_i, $u = \Psi(f)(u_1, \ldots, u_n)$;

  (I.e., to evaluate a non-recursive function application, evaluate the arguments and apply the function.)

- t is of the form $f(t_1, \ldots, t_{n-1})$, f is recursively defined in R by an expression of the form $f(x_1, \ldots, x_{n-1}) \equiv B$, each T_i for $1 \leq i < n$ is an evaluation tree for t_i in (Ψ, R), T_n is an evaluation tree for B in $(\Psi\binom{x_1}{u_1}\cdots\binom{x_{n-1}}{u_{n-1}}, R)$, where each u_i is the result of T_i, and u is the result of T_n;

  (I.e., to evaluate a recursive function application, evaluate the arguments, then return the result of evaluating the body of the function definition in the context obtained from the current context by binding the function's formal parameters to the actual argument values.)

and (b) nothing is an evaluation tree unless it is because of (a).

The effect of (b) is to require that all evaluation trees be finite. If t loops, it has no evaluation tree.

In a context in which R is understood, and it is known that t has an evaluation tree in (Ψ, R), we write $\Psi(t)$ to denote the (necessarily unique) result of an evaluation tree for t in (Ψ, R).

If F is a formula and (Ψ, R) is a context, then F is *true* under (Ψ, R), or equivalently (Ψ, R) *satisfies* F, if and only if one of the following holds:

- F is an atomic formula $t = u$ and there exist evaluation trees for t and u in (Ψ, R) that have identical results.
- F is of the form $\neg G$ and G is not true in (Ψ, R).
- F is of the form $G \wedge H$ and both G and H are true in (Ψ, R).
- F is of the form $(\forall x)\, G$ and for all $u \in U$, G is true in $(\Psi\binom{x}{u}, R)$.

An expression of the form t halts, where t is a term, is an abbreviation for the formula $t = t$. Similarly, t loops is an abbreviation for $t \neq t$, and $t \simeq u$ for (t loops $\wedge$ u loops) $\vee$ $(t = u)$.

Notice that the statement "The formula F is true" is different from the statement "The term t evaluates to true". Perhaps the reader thinks that this is clumsy, and that the formal system could be made more elegant by unifying the notions of formula and term, of being true and of evaluating to true, of the atomic formula $a = b$ and the term **equal**(a, b), and the other mirror-image pairs. But there is a serious obstacle to this plan. Notice that the definition of truth does not need an extremal clause, being justified by "structural induction" on the size of the formula, nor could it have one since the clause for negations is impredicative (see Kleene [31]). On the other hand the definition of an evaluation tree *required* an extremal clause, since the size of the tree is not bounded by the size of the term. Thus one definition will never suffice to define both the semantics of "not" and the semantics of evaluation.

Here are some example formulas to illustrate these definitions.

The formula $\mathrm{p}(\mathrm{f}(x)) \supset \mathrm{f}(x)$ halts is true in all contexts, because an evaluation tree for $\mathrm{p}(\mathrm{f}(x))$ contains one for $\mathrm{f}(x)$.

The formula $(\forall x)\ x$ halts is true in all contexts. Indeed, let Ψ be any interpretation, R any set of recursive functions; then $(x, \Psi(x))$ is an evaluation tree for x in (Ψ, R). This may seem unnatural to those who are used to adding an "undefined" element to the universe to represent the value of looping computations, but from a programmer's point of view it is natural that the evaluator halt on variables. Remember that $(\forall x)$ gives a way of saying something about each value u in the universe, not about each term x.

The last example shows that it is not valid to instantiate a universally quantified variable to an arbitrary term, since although "$(\forall x)\ x$ halts" is valid, there are terms that loop in some contexts. But the following fact is not difficult to prove: if $(\forall x)\ F$ is true in (Ψ, R), and t is any term, then t halts $\supset F^x_t$ is true in (Ψ, R), where F^x_t is the result of substituting t for all free occurences of x in F.

A *theory* is a pair (A, R) where A is a set of formulas, the *axioms* of the theory, and R is a set of recursive function definitions such that no function is defined twice in R. A formula F is *valid* in a theory (A, R) if for any interpretation Ψ and any set R' of recursive definitions such that $R \subseteq R'$, no function is defined twice in R', and (Ψ, R') satisfies every axiom in A, (Ψ, R') satisfies F. A formula is *satisfiable* in a theory if its negation is not valid in the theory. For example, suppose that f is a recursive function symbol. Then $\mathrm{f}(x)$ loops, although true in any context $(\Psi, \{\ \})$, is not valid in the theory $(\{\ \}, \{\ \})$, since for any interpretation Ψ, $\mathrm{f}(x)$ loops is false in the context $(\Psi, \{\mathrm{f}(v) \equiv v\})$. Thus $\mathrm{f}(x)$ halts is satisfiable in the theory $(\{\ \}, \{\ \})$. But $\mathrm{f}(x)$ loops is valid in both of the theories $(\{\ \}, \{\mathrm{f}(x) \equiv \mathrm{f}(x)\})$ and $(\{(\forall x)\, \mathrm{f}(x)\ \text{loops}\}, \{\ \})$.

A *model* for a theory (A, R) is an interpretation Ψ such that every formula $A \in A$ is true in (Ψ, R). If (A, R) is a theory and F is a formula, we write $A, R \vdash F$ if, for every model Ψ of (A, R), F is true in (Ψ, R).

If neither the formula F nor any of the formulas in the set A contain **if-then-else**, $\dot\vee$, or any recursive function symbols, then F is valid in $(A, \{\ \})$ if and only if F is entailed by A in classical first-order logic. In this sense the system is a conservative extension of the first-order predicate calculus.

We now illustrate the definitions above by considering the problem of axiomatizing the theory $\mathcal{Z}$ of the reals and integers under addition. It is staightforward to axiomatize the theory $\mathcal{R}$ of the reals under addition, essentially by writing down the axioms for a transaltion-independent total order on a commutative group. But no formula is equivalent in $\mathcal{R}$ to the infinite disjunction $x = 0 \lor x = 1 \lor x = 2 \lor \ldots$. Thus the natural numbers cannot be given a first-order definition. But the natural numbers can be singled out by the recursive function **nn**, defined by:

$$\mathbf{nn}(x) \equiv [\![\, x = 0 \Rightarrow \mathbf{true};\ \mathbf{nn}(x-1) \,]\!].$$

Notice that if **nn** returns anything at all, it returns **true**, but that it will loop if its argument is not a natural number. Thus the atomic formula $\mathbf{nn}(x)$, (which is an abbreviation for $\mathbf{nn}(x) = \mathbf{true}$), is true if and only if x is a natural number. The formula $\neg\,\mathbf{nn}(x)$ (that is, $\neg\,(\mathbf{nn}(x) = \mathbf{true})$), is true if and only if x is not a natural number. It would be a mistake to write $\mathbf{nn}(x) = \mathbf{false}$—this equality is not true in any context, since if **nn** halts, it does not return **false**.

Similarly, the axiomatization given for $\mathcal{L}$ in section 5 allows "circular" structure. To state that an object is "non-circular" and "finite" under the **car** and **cdr** functions requires a recursive function definition; for example:

$$\mathbf{noncircular}(z) \equiv [\![\mathbf{atom}(z) \Rightarrow \mathbf{true};\ \mathbf{noncircular}(\mathbf{car}(z)) \wedge \mathbf{noncircular}(\mathbf{cdr}(z))]\!].$$

Thus the finite binary trees can be recursively selected out of the more general circular and infinite structures allowed by the first-order axiomatization, just as the natural numbers are selected out of the reals.

This method of defining the non-circular binary trees and the natural numbers comes to the same thing as defining them with some template for inductive definition, as in Boyer and Moore's system [7]. The advantage of definition by recursive selection is that we can easily define the finite circular lists, the directed acylic graphs, and many other useful data structures that are difficult to define with inductive templates.

A point worth noting about the specification language is that an expression like "t halts" makes sense only in a formula, not in a recursive function definition. For example, suppose that **f** has been recursively defined, and the definition

$$\mathbf{g}(x) \equiv [\![\mathbf{f}(x)\ \mathbf{halts}' \Rightarrow x;\ \mathbf{g}(x+1)]\!]$$

is made with the intention of defining **g** so that **g**(0) finds the first natural number for which **f** halts. The definition fails, since it is an abbreviation for

$$\mathbf{g}(x) \equiv \mathbf{if\text{-}then\text{-}else}(\mathbf{equal}(\mathbf{f}(x), \mathbf{f}(x)), x, \mathbf{g}(x+1)),$$

and therefore **g** loops whenever **f** does. To make the required definition, it is first necessary to define a non-recursive function **fhalts** that recognizes the values on which **f** halts:

$$\text{Predicate } \mathbf{fhalts}(x) \equiv \mathbf{f}(x)\ \mathbf{halts}.$$

Now **g** can test **fhalts**.

In particular, if a recursive definition tests **nn**(x), the false branch of the test will never be taken! For this reason we add to the theory Z the definition

$$\text{Predicate } \mathbf{int}(i) \equiv \mathbf{nn}(i) \vee \mathbf{nn}(--i).$$

Recursive definitions should always use **int**, never **nn**.

These considerations make it obvious that the valid formulas of the system are not contained in any finite level of the arithmetic hierarchy (see Kleene [31]). Of course, this also follows from the fact that the system has the natural numbers and quantification.

A final point about the specification language: note that the dovetail operator $\triangledown$ cannot be replaced by an auxilliary predicate definition, although its effect is essentially to introduce a "logician's or" into a term. Consider, for example, the definition of the predicate **r**(x, y) which is to be true if and only if y is equal to $f(g(g(...(x)...)))$, for any sequence of applications of the functions f and g. The obvious definition works fine:

$$\mathbf{r}(x,y) \equiv [\![x = y \Rightarrow \mathbf{true};\ \mathbf{r}(f(x), y) \triangledown \mathbf{r}(g(x), y)]\!].$$

Here is another attempted definition:

$$\text{Function } \mathbf{r}(x,y) \equiv x = y \vee \mathbf{rr}(x,y).$$

$$\text{Predicate } \mathrm{rr}(x,y) \equiv \mathrm{r}(f(x),y) \lor \mathrm{r}(g(x),y).$$

The $\lor$ in the definition of rr is the "logician's or", but the definition fails. It would be consistent with this definition to make $\mathrm{rr}(x,y)$ identically true; then $\mathrm{r}(x,y)$ would as a consequence be identically true also, thus rr so interpreted would satisfy its definition. The problem is that predicate definitions do not have an implicit extremal clause, so "overly liberal" interpretations must always be reckoned with.

17. Formal semantics for the programming language

The programming language is expression-oriented rather than statement-oriented; the execution of a program fragment is similar to the evaluation of an expression, but different in that it may cause side effects. The syntax of program fragments is almost the same as the syntax of terms, with the difference that programs may have embedded assignment statements. More precisely, a *statement* is either a variable, an expression of the form $f(S_1, \ldots, S_n)$ where f is an n-ary function symbol and $S_1, \ldots, S_n$ are statements, or an expression of the form $x \leftarrow S$, where x is a variable and S is a statement, or an expression of the form **with** $v_1 = S_1, \ldots, v_n = S_n$ **do** S, where the v_i are variables and the S_i are statements.

There is no syntactic provision for blocks like $[\![S_1;\ S_2;\ S_3]\!]$; this block is viewed as an abbreviation for the statement $\mathrm{do}(S_1, \mathrm{do}(S_2, S_3))$. (The 2-ary function symbol do will be defined to evaluate its arguments in order and return the value of the second argument. Presumably the first argument causes side effects. This do is not related to the **do** that occurs as syntactic sugar in the **with** statement.) Similarly, $[\![S_1;\ P \Rightarrow E;\ S_2;\ S_3]\!]$ is an abbreviation for

$$\mathrm{do}(S_1, \text{if-then-else}(P, E, \mathrm{do}(S_2, S_3))).$$

Also, the statement $[\![\ ]\!]$, which is used as a no-op, is an abbreviation for the statement nothing; nothing is an ordinary constant symbol, about which the system contains no axioms, so it behaves as a completely arbitrary value.

The general rules for converting informal statements involving $[\![$, $\Rightarrow$, and $]\!]$ into the formal statements they abbreviate are: (a) $[\![\ ]\!]$ is an abbreviation for nothing; (b) $[\![S]\!]$, where S is a statement, is an abbreviation for S; (c) $[\![S \Rightarrow A;\ S_1;\ \ldots;\ S_n]\!]$ is an abbreviation for $\text{if-then-else}(S, A, [\![S_1;\ \ldots;\ S_n]\!])$; and (d) $[\![S;\ S_1;\ \ldots;\ S_n]\!]$, where S is not of the form $P \Rightarrow Q$, is an abbreviation for $\mathrm{do}(S, [\![S_1;\ \ldots;\ S_n]\!])$.

The formal syntax allows only variables on the left of assignments; an assignment like $l(x) \leftarrow y$ is regarded as an abbreviation for $l \leftarrow l^{(x)}_{y}$.

Notice that every term is a statement. The notational conventions for $\land$, $\lor$, $\supset$, $\neg$, $\Leftrightarrow$, equal, store, and select that were defined for terms will also be used for statements.

A *procedure definition* is an expression of the form

$$f(x_1, \ldots, x_n) \equiv B$$

where f is an n-ary function symbol, $x_1, \ldots, x_n$ are distinct variables, and B is a statement. Notice that every recursive function definition is also a procedure definition.

The formal model for statement execution is similar to the formal model for term evaluation. An object, called an *execution tree*, is defined that represents a trace of the execution of a statement. The node of the tree that represents the execution of a statement S is labelled with two interpretations, one for the context at entry to S, the other for the context at exit from S.

The precise definition of an *execution tree* is as follows: (a) If T is an execution tree for the statement S in the *entry context* (Ψ, R) with *result value* u and *result interpretation* Φ, then T is a sequence $(\Psi, S, \Phi, u, T_1, \ldots, T_n)$ such that one of the following seven conditions hold:

- S is a variable, $n = 0$, $u = \Psi(S)$, and $\Phi = \Psi$;

  (I.e., to "execute" a variable, return its value.)

- S is of the form do(S_1, S_2), $n = 2$, T_1 is an execution tree for S_1 in (Ψ, R), T_2 is an execution tree for S_2 in (Ψ', R), where Ψ' is the result interpretation of T_1, and u and Φ are the result value and interpretation of T_2;

  (I.e., to execute do(S_1, S_2)*, execute S_1, then execute S_2, then return the second result.)*

- S is of the form if-then-else(p, a, b), $n = 2$, T_1 is an execution tree for p in (Ψ, R), and, letting u' and Ψ' be the result value and result interpretation of T_1, one of the following two cases holds: (3a) $u' = \Psi(\text{true})(\,)$, T_2 is an execution tree for a in (Ψ', R), and u and Φ are the result value and interpretation of T_2; or (3b) $u' \neq \Psi(\text{true})(\,)$, T_2 is an execution tree for b in (Ψ', R), and u and Φ are the result value and interpretation of T_2;

  (I.e., to execute if-then-else(p, a, b)*, execute p; if the result is true, then return the result of executing a, else return the result of executing b.)*

- S is of the form $x \leftarrow S'$, $n = 1$, T_1 is an execution tree for S' in (Ψ, R), and $\Phi = \Psi'{}^{(x)}_{u'}$, where Ψ' and u' are the result interpretation and result value of T_1.

  (I.e., to execute an assignment statement, execute the expression on the right, rebind the variable on the left to the value of the expression on right, and return anything.)

- S is of the form $f(S_1, \ldots, S_n)$, f is not if-then-else or do, f is non-recursive, and, if $\Psi_0 = \Psi$ and, for $1 \leq i \leq n$, Ψ_i and u_i are the result interpretation and result value of T_i, then each T_i is an execution tree for S_i in (Ψ_{i-1}, R), $\Phi = \Psi_n$, and $u = \Psi(f)(u_1, \ldots, u_n)$;

  (I.e., to execute the application of a non-recursive function, compute the values of the arguments, going from left to right, then return the value of the function applied to the argument values.)

- S is of the form with $x_1 = S_1, \ldots, x_{n-1} = S_{n-1}$ do S', and, if $\Psi_0 = \Psi$ and, for $1 \leq i < n$, Ψ_i and u_i are the result interpretation and result value of T_i, then each T_i, $1 \leq i < n$, is an execution tree for S_i in (Ψ_{i-1}, R), T_n is an execution tree for S' in $(\Psi_{n-1}{}^{(x_1)}_{u_1} \cdots {}^{(x_{n-1})}_{u_{n-1}}, R)$, u is the result value of T_n, and, letting Ψ' be the result interpretation of T_n, $\Phi = \Psi'{}^{(x_1)}_{\Psi_{n-1}(x_1)} \cdots {}^{(x_{n-1})}_{\Psi_{n-1}(x_{n-1})}$;

  (I.e., to execute a with *statement, compute the initial values, working from left to right, bind the local variables, execute the body of the statement, and finally restore the old values of the local variables.)*

- S is of the form $f(S_1, \ldots, S_{n-1})$, f is defined in R by $f(x_1, \ldots, x_{n-1}) \equiv B$, and, if $\Psi_0 = \Psi$ and, for $1 \leq i < n$, Ψ_i and u_i are the result interpretation and result value of T_i, then each T_i, $1 \leq i < n$, is an execution tree for S_i in (Ψ_{i-1}, R), T_n is an execution tree for B in $(\Psi_{n-1}{}^{(x_1)}_{u_1} \cdots {}^{(x_{n-1})}_{u_{n-1}}, R)$, u is the result value of T_n, and, letting Ψ' be the result interpretation of T_n, $\Phi = \Psi'{}^{(x_1)}_{\Psi_{n-1}(x_1)} \cdots {}^{(x_{n-1})}_{\Psi_{n-1}(x_{n-1})}$;

  (I.e., to execute a procedure call, compute the values of the arguments, going from left to right; bind the locals to the argument values; execute the body of the procedure; and rebind the locals before returning the value of the body.)

and (b) anything is an execution tree unless it is barred from being one by (a).

The effect of (b) is to allow infinite execution trees. (At least, that is the intended meaning. If the ordinary form of extremal clause makes sense, then (b) must make sense too.) Infinite execution trees are allowed in order to make specifications about non-terminating computations. For example, suppose the procedure p loops forever, calling q (which halts) each time around the loop. Then the statement that q's entry invariant is true for each call can be expressed by saying that q's entry invariant is true in the context of certain nodes on a certain infinite branch of the execution tree for the call to p.

Notice that no special interpretation is given to **dovetail** in the above definition; the effect of this is to bar procedures from using $\dot{\vee}$, or from calling functions that use it. This is a reasonable restriction from a programmer's point of view.

By regarding finite sequences as nests of ordered pairs and interpretations as arrays, so that $\Psi(x)$ is modeled by **select**(Ψ, x), it is possible to define terms, formulas, and evaluation trees in the specification language itself, using simple recursive function definitions. On the other hand, it is obviously impossible to write a recursive function **xtree** that returns **true** if its argument is an execution tree, because an infinite tree cannot be examined in a finite number of steps. But it is possible to write a recursive function **notxtree** that returns **true** if its argument is *not* an execution tree. One way is to make **notxtree** do a breath-first search of its argument, looking for a violation of one of the above rules. If it ever finds a violation, it returns **true**. If it searches its entire argument without finding a violation, it returns **false**. If its argument is an infinite execution tree, it will loop, but then an atomic formula containing the call will be untrue, as desired. Another way, avoiding the bookkeeping of the breadth-first search, is to use $\dot{\vee}$ to guess which branch of the tree violates the rules. Either way, the predicate definition Predicate **xtree**$(x) \equiv \neg$ **notxtree**(x) can be used to obtain a positive predicate.

Consider the procedure definition and specifications:

Procedure **foo**() $\equiv$ [[**foo**(); **foo**()]].
Entering **foo**, $x = y$.
Exiting **foo**, **false**.

These specifications and definition are consistent, since if $x = y$ is true at some top-level call to **foo**, then as long as the program runs, $x = y$ will be true at every entry to **foo**, and **false** will be true at every exit from **foo**. Now let T be an execution tree for the term **foo**() in the context (Ψ, R), where R contains the definition of **foo** above. Because **foo** calls itself twice, T is essentially an infinite binary tree; the looping computation corresponds to the left-most branch. The entry contexts of the nodes on the left-most branch are all Ψ, but the exit contexts of the nodes on the left-most branch are arbitrary, so long as they are all the same. Thus the entry contexts for the calls to **foo** that are not on the left-most branch need not satisfy $x = y$.

This example shows that it is necessary to distinguish the "accessible" parts of an execution tree, that is, the parts of the tree that correspond to states that would be reached in an actual computation. If T is an execution tree, then (a) T is accessible from T, (b) if $(\Psi, S, \Phi, u, T_1, \ldots, T_n)$ is accessible from T, and T_i is finite for $1 \leq i < k$, then T_k is accessible from T; and (c) nothing is accessible from T unless it is because of (a) and (b). The accessible part of an execution tree is sort of an execution sequence. It is the only part of the tree that is of interest, but it is easier to define the whole tree first and then single out the left-most infinite branch than to do both in a single definition.

We are finally ready to describe the experimental verifer precisely. The experimental verifier reads a file containing a sequence of function, procedure, and predicate definitions, entry and exit specifications, axioms, and lemmas and determines whether the file is consistent or not. Essentially, the file is consistent if every lemma is a consequence of the axioms and definitions, and any call to a procedure in a context that satisfies its entry specification will result in a computation that never violates any specifications. To make this definition precise, we describe the simple algorithm that the experimental verifier executes.

Algorithm V. (*Verify a file*). This algorithm determines whether a program meets its specifications.

V1. [Get file name.] Prompt the user for the name of a file to verify, read the file name, and open the file.

V2. [Initialize.] Initialize the variables L and R' to be empty sets. (These variables represent the set of lemmas and the set of procedure definitions, respectively.) Initialize E and X to be functions such that for all function symbols p, $E(p) = X(p) =$ true. (These variables represent the entry and exit specifications; the default specifications for a procedure are true.) Initialize A to be the union of the sets of axioms for $Z^{\times}$, $\mathcal{L}$, and $\mathcal{A}$; initialize R to be the set containing only the recursive definition of nn. (These variables represent the set of axioms and the set of recursive function definitions.) Read the file opened in step V1, which (like all files) contains a sequence of expressions separated by periods. For each expression, perform one of the following actions:

- [Axiom?] If the expression is of the form Axiom n: F, where n is a numeral and F is a formula, then add F to A.
- [Predicate?] If the expression is of the form Predicate $P(x_1, \ldots, x_n) \equiv B$, where $x_1, \ldots, x_n$ are distinct variables, P is a non-recursive function symbol, and P does not appear in B and has not appeared previously in the file, then add the formula $(\forall x_1, \ldots, x_n)\ P(x_1, \ldots, x_n) \Leftrightarrow B$ to A.
- [Lemma?] If the expression is of the form Lemma n: F, where n is a numeral and F is a formula, then add F to L.
- [Function?] If the expression is of the form Function D, where D is a recursive function definition, then add D to both R and R'. (D is added to both sets because a recursive function may appear in both programs and specifications.)
- [Procedure?] If the expression is of the form Procedure D, where D is a procedure definition, then add D to R'.
- [Entry specification?] If the expression is of the form Entering P, F, where F is a formula and P is a function symbol, then set $E(P) \leftarrow F$.
- [Exit specification?] If the expression is of the form Exiting P, F, where F is a formula and P is a function symbol, then set $X(P) \leftarrow F$.
- [Error?] If none of the other cases apply to the expression, then give an error message and halt.

V3. [Consistent?] (The end of the file has been reached without syntactic error.) If, for every model Ψ of (A, R), the following two conditions hold:

- Each formula $F \in L$ is true in (Ψ, R);
- For each procedure p defined in R' with locals $x_1, \ldots, x_n$, if $E(p)$ is true in (Ψ, R), T is an execution tree for $p(x_1, \ldots, x_n)$ in (Ψ, R'),

$$(\Phi, q(t_1, \ldots, t_m), \Phi', u, T_1, \ldots, T_{m+1})$$

is a subtree of T such that T_{m+1} is accessible from T, and q is defined in R', then $E(q)$ is true in (Φ'', R), where Φ'' is the entry context for T_{m+1}, and if T_{m+1} is finite, then $X(q)$ is true in $(\Phi'{}^{(V)}_{u}, R)$;

then print "`Sorry. Can't find any more errors.`" Otherwise print a message indicating which lemma or program fragment violates these conditions and why. ∎

That is, the verification succeeds if every lemma is a consequence of the axioms and definitions, and any call to a procedure in a context that satisfies its entry specification will result in a computation that never violates any entry or exit specifications. (Recall that $\mathcal{V}$ is the variable denoting a procedure's returned value in its exit specification.)

18. Correctness of Algorithm E

Now that we have the formal definition of the semantics of the specification language, we can prove that Algorithm E of section 5 is valid for theories that contain partial recursive function definitions.

Here is an example of a pair of theories for which Algorithm E fails: let $\mathcal{S}$ be the theory with no recursive function definitions and the single axiom

$$(\exists a\, b)\, (\forall x)\; x = a \vee x = b,$$

and let $\mathcal{T}$ be the theory with no recursive function definitions and the single axiom

$$(\exists a\, b\, c)\; a \neq b \wedge a \neq c \wedge b \neq c.$$

These two theories have no free functions, hence they have no common free functions. Start Algorithm E with these two theories and two empty stacks S_1 and S_2. The empty conjunction is satisfiable in both $\mathcal{S}$ and $\mathcal{T}$, there are no equalities to propagate, no splits to do, so the algorithm incorrectly returns "`satisfiable`". This example motivates the following definitions.

A theory is *satisfiable* if it has a model. A theory (A, R) is *stably infinite* if for all quantifier-free formulas F, $(A \cup \{F\}, R)$ has an infinite model if it has any model at all. Note that any theory that has no finite models is stably infinite; therefore $\mathcal{R}$ is stably infinite. It is not difficult to prove that $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{A}$ are stably infinite.

This explains the first restriction on $\mathcal{S}$ and $\mathcal{T}$ required by Algorithm E. The second restriction is that the theories contain only finitely many "free variables". A *free variable* of a theory is a free variable of some axiom or recursive function definition of the theory. A *free variable* of a recursive function definition is a variable that occurs in the body of the definition but not in the list of locals. None of the theories $\mathcal{Z}$, $\mathcal{L}$, and $\mathcal{A}$ have free variables. $\mathcal{E}$ has one free variable, x.

To see that this restriction is necessary, consider the theory $\mathcal{S}$ with all of the axioms and recursive function definitions of $\mathcal{Z}$, as well as the axioms $x_0 = 0$, $x_1 = 1, \ldots$. That is, $\mathcal{S}$ labels every natural number with a variable. Let $\mathcal{T}$ be the theory with no recursive function definitions, and the axioms $x \neq x_0$, $x \neq x_1, \ldots$. Using these two theories, start Algorithm E with S_2 empty and S_1 containing the single literal $\mathrm{nn}(x)$; it will incorrectly return "`satisfiable`".

The correctness of Algorithm E is intuitively a consequence of Craig's interpolation lemma, but Craig's proof of this lemma (as well as all the proofs in the literature that the author has found) are proof-theoretic, and therefore invalid for theories that contain recursive function definitions, where not all true statements are provable. Therefore, we must either give a model-theoretic proof of Craig's lemma or do without it. It turns out to be just as easy to give a direct model-theoretic proof of the correctness of Algorithm E. The proof is a long but straightforward "cut-and-paste" argument on interpretations.

If A is a term or formula and R is a set of recursive definitions, the function symbols that are *relevant to A via R* are defined as follows: (a) if f occurs in A, f is relevant to A via R; (b) if

g is a function symbol defined in R that is relevant to A via R, and f occurs in the definition of g, then f is relevant to A via R, and (c) nothing is relevant to A via R unless it is because of (a) and (b). A variable v is *relevant to A via R* if v occurs free either in A or in the definition of some recursive function f that is relevant to A via R.

The contexts (Ψ_1, R_1) and (Ψ_2, R_2) are *compatible* for A, where A is a term or a formula, if (a) Ψ_1 and Ψ_2 have the same universe; and (b) if f is a recursive function symbol relevant to A via R_i, then f is relevant to A via R_{3-i}, and the definitions for f in R_i and R_{3-i} are identical; and (c) Ψ_1 and Ψ_2 agree at every variable or non-recursive function symbol that is relevant to A via R_1 or R_2.

We now check formally that if (Ψ, R) and (Ψ', R') are compatible for a formula F, then F is true in (Ψ, R) if and only if F is true in (Ψ', R'). The proofs are straightforward inductions.

Lemma 1. *Let t be a term, (Ψ, R) and (Ψ', R') compatible contexts for t, n an integer, and u an element of the common universe of Ψ and Ψ'. Then there exists an evaluation tree for t in (Ψ, R) with result u and height n if and only if there exists an evaluation tree for t in (Ψ', R') with result u and height n.*

Proof. The proof is a trivial induction on n. Only one case of the argument will be given, that in which t is the application $f(t_1, \ldots, t_m)$ of a recursive function f. This function must be defined by some expression $f(x_1, \ldots, x_m) \equiv B$ in both R and R' (or else it is defined in neither and t has no evaluation tree in either (Ψ, R) or (Ψ', R')). Suppose there is an evaluation tree for t in (Ψ, R) with result u and height n. The tree is an expression of the form $(\Psi, t, u, T_1, \ldots, T_{m+1})$ where, for $1 \leq i \leq m$, T_i is an evaluation tree for t_i in (Ψ, R), with result, say, u_i, and T_{m+1} is an evaluation tree for B in $(\Psi\binom{x_1}{u_1}\cdots\binom{x_m}{u_m}, R)$ with result u. The maximum of the heights of T_1, $\ldots$, T_{m+1} is $n-1$, so it follows by induction that there exist evaluation trees $T'_1, \ldots, T'_{m+1}$, such that T'_i is an evaluation tree for t_i in (Ψ', R') with result u_i, for $1 \leq i \leq m$, and T'_{m+1} is an evaluation tree for B in $(\Psi'\binom{x_1}{u_1}\cdots\binom{x_m}{u_m}, R)$ with result u, where the maximum of the heights of $T'_1, \ldots, T'_{m+1}$ is $n-1$. (For $(\Psi\binom{x_1}{u_1}\cdots\binom{x_m}{u_m}, R)$ and $(\Psi'\binom{x_1}{u_1}\cdots\binom{x_m}{u_m}, R)$ certainly agree at $x_1, \ldots, x_m$, and by assumption at all other variables appearing in B.) Thus $(\Psi', t, u, T'_1, \ldots, T'_{m+1})$ is an evaluation tree for t in (Ψ', R') with height n and result u. ∎

Lemma 2. *Let F be a formula and let (Ψ, R) and (Ψ', R') be contexts compatible for F, with common universe U. Then F is true in (Ψ, R) if and only if F is true in (Ψ', R').*

Proof. The proof is an obvious induction argument on the size of F. If F is an atomic formula $t = u$, the previous lemma says that there exist evaluation trees for t and u with the same result in (Ψ, R) if and only if there exist such trees in (Ψ', R'). If F is of the form $\neg G$ or of the form $G \wedge H$, the induction argument is trivial. If F is of the form $(\forall x)G$, F is true in (Ψ, R) if and only if for all $u \in U$, G is true in $(\Psi\binom{x}{u}, R)$; that is if and only if for all $u \in U$, G is true in $(\Psi'\binom{x}{u}, R')$, (since $\Psi\binom{x}{u}$ and $\Psi'\binom{x}{u}$ must agree at x and by assumption they agree at every free variable of G); that is, if and only if F is true in (Ψ', R'). ∎

The reference in Algorithm E to the satisfiability or non-convexity of "the conjunction of literals in S_1" (or S_2) should of course be taken to mean the satisfiability or convexity of the conjunction in the theory S (or T). To be precise, we define a theory (A, R) to be *convex* if it entails no disjunction $u_1 = v_1 \vee \ldots \vee u_k = v_k$ for $k > 1$ unless it entails one of the equalities $u_i = v_i$. Thus a formula F is convex in a theory (A, R) if and only if the theory $(A \cup \{F\}, R)$ is convex.

If a theory has an infinite model, it is easy to prove that it has a denumerably infinite model, since the interpretation can be pulled back to the set of terms in the obvious way.

We are now ready to prove the correctness of Algorithm E.

Theorem 1. (Algorithm E is correct.) *Let (A, R) and (B, S) be satisfiable theories such that neither (A, R) nor (B, S) contain infinitely many free variables, both (A, R) and (B, S) are stably infinite and convex, no free function of (A, R) is a free function of (B, S), and finally, for any two variables x and y, either both (A, R) and (B, S) entail $x = y$ or neither do. Then $(A \cup B, R \cup S)$ is satisfiable.*

Before proving the theorem let us see why it justifies the algorithm. Let F_1 and F_2 be the conjunctions of literals in the stacks S_1 and S_2, respectively, at the moment step E5 returns "satisfiable". Let the theories S and T be (A_1, R_1) and (A_2, R_2). Then the conditions of theorem 1 are satisfied for the theories $(A_1 \cup \{F_1\}, R_1)$ and $(A_2 \cup \{F_2\}, R_2)$, so the theorem implies that $(A_1 \cup A_2 \cup \{F_1, F_2\}, R_1 \cup R_2\}$ has a model; that is, that $F_1 \wedge F_2$ is satisfiable in the combination of S and T.

Proof of Theorem 1. Let $v_1, \ldots, v_n$ be the free variables of (A, R) and (B, S). Partition the set $\{v_1, \ldots, v_n\}$ into $\{Q_1, \ldots, Q_k\}$ by putting v_i and v_j into the same equivalence class if and only if $A, R \vdash v_i = v_j$. (Equivalently, if $B, S \vdash v_i = v_j$.) Let w_i be some variable in Q_i, for i between 1 and k. Let F be the conjunction

$$\bigwedge_{1 \le i < j \le k} (w_i \ne w_j).$$

(If $k < 2$, F is the constant true.) Now suppose that $(A \cup \{F\}, R)$ has no model. Then $k \ge 2$ and

$$A, R \vdash \bigvee_{1 \le i < j \le k} (w_i = w_j).$$

If $k > 2$, this contradicts the assumption that (A, R) is convex. If $k = 2$, this contradicts the fact that w_1 and w_2 are in different equivalence classes. Thus $(A \cup \{F\}, R)$ does have a model; let it be Ψ with universe U. Since (A, R) is stably infinite, we may assume that the cardinality of U is the cardinality of the set of integers, ω. A similar argument shows that $(B \cup \{F\}, S)$ has a model Ψ' with universe U' with cardinality ω.

Let $\alpha : U \to U'$ be a bijection such that (a) $\alpha(\Psi(\text{true}()) = \Psi'(\text{true}())$; (b) $\alpha(\Psi(\text{false}()) = \Psi'(\text{false}())$; and (c) $\alpha(\Psi(v_i)) = \Psi'(v_j)$ if and only if v_i and v_j are in the same Q_p, for some $1 \le p \le k$. That an α exists satisfying (c) follows from the fact that the contexts satisfy F—the interpretations make two variables equal only if they are forced to.

We now define an interpretation Φ and prove it is a model for $(A \cup B, R \cup S)$. The universe of Φ is U. For a variable v, $\Phi(v) = \Psi(v)$. For a free function f of (A, R), $\Phi(f) = \Psi(f)$. For a free function f of (B, S), $\Phi(f)(u_1, \ldots, u_l)$ is $\alpha^{-1}(\Psi'(f)(\alpha(u_1), \ldots, \alpha(u_l)))$. On function symbols that are not free functions of either theory, Ψ is arbitrary.

It is now easy to show that for each formula A in A, (Ψ, R) and $(\Phi, R \cup S)$ are compatible for A: (A, R) and (B, S) have no common free functions, therefore any function f relevant to A via $R \cup S$ is relevant to A via R, hence any such f is a free function of (A, R), and therefore Φ and Ψ agree on f. It follows that $(\Phi, R \cup S)$ satisfies every formula in A.

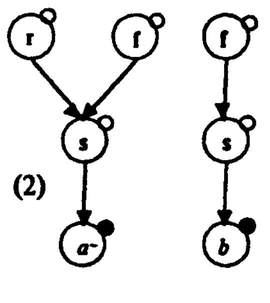
To show that $(\Phi, R \cup S)$ satisfies every formula in B, consider the interpretation Ψ'—which by assumption satisfies (B, S)—as it is mapped by α^{-1} onto the universe U. To be precise, let the interpretation Ψ'' with universe U be defined by $\Psi''(v) = \alpha^{-1}(\Psi'(v))$, for any variable v; and $\Psi''(f)(u_1, \ldots, u_n) = \alpha^{-1}(\Psi'(f)(\alpha(u_1), \ldots, \alpha(u_n))$, for any function symbol f. This interpretation Ψ'' differs from Ψ' only by a renaming of the elements of the universe, so Ψ'' must also be a model of (B, S). But the definitions of Ψ'' and Φ agree for free functions of (B, S), so (Ψ'', S) and $(\Phi, R \cup S)$ are compatible for every formula B in B. This proves that $(\Phi, R \cup S)$ satisfies every formula in $A \cup B$, which proves the theorem. ▮

19. Recursive functions in the E-graph

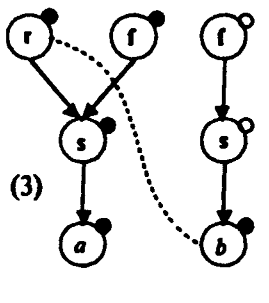
The satisfiability procedure for $\mathcal{E}$ described in section 6 must be generalized to allow recursive functions, since if f is a recursive function symbol, $x = y \land \mathsf{f}(x) \not\simeq \mathsf{f}(y)$ is satisfied by a context in which $\mathsf{f}(x)$ loops. Here is an example illustrating how recursive functions can be handled by a simple generalization of the method used for non-recursive functions. Suppose that r and s are recursive function symbols and f is a non-recursive function symbol. Figures (2), (3), and (4) illustrate the mechanical proof of the formula

$$\mathsf{r}(\mathsf{s}(a)) \simeq b \land \mathsf{s}(a) \simeq \mathsf{s}(b) \supset \mathsf{f}(\mathsf{s}(a)) = \mathsf{f}(\mathsf{s}(b)). \tag{1}$$

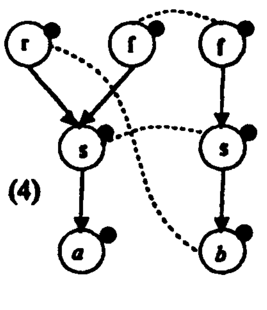
First a graph is constructed representing the set of terms appearing in (1). Each vertex of the graph contains a one-bit field that is set if the term represented by the vertex is known to halt. Initially, in (2), the halt-bit is set only in the vertices representing a and b. Equivalence classes in the graph represent equivalence classes of $\simeq$, not of $=$. To "assume" the literal $t \simeq u$, the equivalence class of the representatives of t and u are merged. To assume the literal $t = u$, the equivalence classes are merged and the halt bits of all vertices in the resulting class are set. The other cases in which halt bits are set will be illustrated by this example. When the vertex for $\mathsf{r}(\mathsf{s}(a))$ is merged with that for b, the halt bit of the former vertex is set, since it has become equivalent to a vertex whose halt bit is set. (The formula $\mathsf{r}(\mathsf{s}(a)) \simeq b$ is a silly one chosen only for this example; it would be more reasonable to write $\mathsf{r}(\mathsf{s}(a)) = b$, since variables always halt.) Since a term loops if any of its subterms loop, we can set the halt bits of all successors of any vertex whose halt bit is set; thus we may now set the halt bit of the vertex representing $\mathsf{s}(a)$. Since an application of a non-recursive function halts if all its arguments halt, we can set the halt bit of a vertex labelled with a non-recursive function if all its successors have their halt bits set, thus after setting the halt bit of $\mathsf{s}(a)$ we can set the halt bit of $\mathsf{f}(\mathsf{s}(a))$. Figure (3) shows the state of the graph after $\mathsf{r}(\mathsf{s}(a)) \simeq b$ is assumed. When $\mathsf{s}(a) \simeq \mathsf{s}(b)$ is assumed, the halt bit of $\mathsf{s}(b)$ is set, since this vertex becomes equivalent to another whose halt bit is set. Figure (4) shows the state of the graph after $\mathsf{s}(a) \simeq \mathsf{s}(b)$ is assumed; notice that the graph explicitly represents the fact that $\mathsf{f}(\mathsf{s}(a))$ and $\mathsf{f}(\mathsf{s}(b))$ halt with the same value.



(2)



(3)



(4)

Let $G = (V, E, \lambda, R)$ be an E-graph and H a subset of V. (H represents the vertices whose halt bits are set.) Then H is *closed* with respect to G if (a) $u \in H$ and $(u, v) \in R$ imply $v \in H$, (b) $u \in H$ and v a successor of u imply $v \in H$, (c) if $\lambda(v)$ is a non-recursive function symbol and all successors of v are in H, then v is in H, (d) if $\lambda(v)$ is a variable, then $v \in H$. If $H \subset V$, the *closure* of H is the smallest closed super-set of H.

Theorem H. *Let S be a finite set of positive literals (that is, of equalities between terms), T a finite set of negative literals (that is, of disequalities between terms) that contains* **true** $\not\simeq$ **false**, *and suppose that no literal in S or T contains any occurence of* **if-then-else** *or* **dovetail**. *Let $G =$*

(V, E, λ, R) be the E-graph corresponding to the set of terms appearing in S and T, where R is the congruence closure of $\{(t,u) \mid t = u \in S\}$ and H the closure with respect to G of $\{t, u \mid t = u \in S\}$. Then there exists a context satisfying all the literals in S and T if and only if for every $t \neq u$ in T, either $(t,u) \notin R$, or $t \notin H$.

Proof. It is straightforward to prove the "only if" part. To prove the "if" part, we construct a context satisfying the literals in S and T. Let the recursive function symbols appearing in literals in S and T be $f_1, \ldots f_k$. Choose distinct non-recursive function symbols $L_1, \ldots, L_k, F_1, \ldots, F_k$ that do not appear in any literal in S or T, such that for each i from 1 to k, the arity of L_i and F_i is equal to the arity of f_i. We define an interpretation Ψ with universe U, where U is the partition of the vertices of G corresponding to R, by the rules: (a) For a variable v, $\Psi(v)$ is the equivalence class of any vertex of G labelled with v, if one exists; otherwise $\Psi(v)$ is arbitrary. (b) For an n-ary non-recursive function symbol g that is not one of the L_i or F_i, $\Psi(g)(Q_1, \ldots, Q_n)$ is the equivalence class of any vertex v labelled g such that $v[i] \in Q_i$, for $1 \leq i \leq n$, if such a vertex exists; otherwise $\Psi(g)(Q_1, \ldots, Q_n)$ is arbitrary. (c) $\Psi(L_i)(Q_1, \ldots, Q_k)$ is $\Psi(\text{true})()$ if G contains a vertex v labelled f_i such that $v[i] \in Q_i$ for $1 \leq i \leq k$ and $v \notin H$; $\Psi(L_i)(Q_1, \ldots, Q_k)$ is $\Psi(\text{false})()$ otherwise. (d) $\Psi(F_i)(Q_1, \ldots, Q_k)$ is the equivalence class of any vertex v labeled f_i such that $v[i] \in Q_i$, $1 \leq i \leq k$, if such a vertex exists; otherwise $\Psi(F_i)(Q_1, \ldots, Q_k)$ is arbitrary.

The effect of these definitions is that L_i recognizes those argument tuples on which f_i is supposed to loop, and F_i is a non-recursive function that agrees with f_i wherever f_i halts. Now let D be the set containing, for each f_i, the recursive definition

$$f_i(x_1, \ldots, x_n) \equiv [\![L_i(x_1, \ldots, x_n) \Rightarrow f_i(x_1, \ldots, x_n); F_i(x_1, \ldots, x_n)]\!]$$

where n is the arity of f_i and $x_1, \ldots, x_n$ are any n distinct variables. Thus f_i loops wherever it is supposed to, and halts with the right value wherever it is supposed to halt. It is now straightforward to show that for every term t appearing in S or T, there is an evaluation tree for t in (Ψ, D) if and only if $t \in H$; and if there is an evaluation tree, then its result is the equivalence class of t. Thus every formula in either S or T is satisfied by the context (Ψ, S). ∎

The theorem would be true for infinite sets S and T as well, so long as there were enough non-recursive function symbols left over to serve as the L's and F's.

The theorem gives a procedure for determining the satisfiability of conjunctions of literals of the form $t = u$ or $t \neq u$. It is trivial to modify it to handle literals of the form $t \simeq u$, but if literals of the form $\neg t \simeq u$ are allowed, the problem becomes NP-complete. The theorem prover must do a three-way case split when it is is given such a literal; the cases are t loops, u loops, and t halts $\wedge$ u halts $\wedge$ $t \neq u$.

It is straightforward to modify Algorithm D of section 11 to maintain halt bits. The halt bits are set in every atomic E-node except the label nodes for recursive functions. Whenever the halt bit is set in a vertex node N, it must be set in *ecar*(*ecdr*(N)), *ecar*(*ecdr*(*ecdr*(N))), and so on. (But not in *ecar*(N), which may be a label node for a recursive function.) Whenever the halt bits are set in *ecar*(N) and *ecdr*(N), the halt bit must be set in N. When merge finds that its arguments have been asserted unequal, it does not signal a contradiction unless the halt bit of one of the two equivalence classes has been set. Therefore, it is possible that setting a halt bit produces an inconsistency, and this must be checked for.

20. An induction rule

The purpose of this section is to show that the heuristic methods of Boyer and Moore [7] can be used to find inductive proofs of the validity of theorems in the system defined in Section 16. This is not obvious, since Boyer and Moore consider only total functions, while we must deal with partial functions. Also, the dovetail operator $\overset{\circ}{\vee}$ must be reckoned with.

The first step in proving a formula by induction mechanically is to choose an "induction scheme"; this is essentially a list that contains, for each case of the induction argument, the condition and list of induction hypotheses for the case.

An *induction scheme* (or simply a *scheme*) on the distinct variables $x_1, \ldots, x_k$ is a set of pairs $\{(P_1, S_1), \ldots, (P_m, S_m)\}$, where each P_i is a formula and each S_i is a set of substitutions on $x_1, \ldots, x_k$. We denote a substitution θ on $x_1, \ldots, x_k$ by the expression $(x_1, \ldots, x_k) \rightarrow (x_1\theta, \ldots, x_k\theta)$, sometimes omitting the parentheses if $k = 1$. Each pair in an induction scheme represents a case of an induction argument; P_i represents the assumptions for the case, and each substitution in S_i represents an "inductive hypothesis". The cases for which S_i is empty are the "base cases" of the argument. For example, the scheme

$$\{(x = 0, \{\ \}), (x \neq 0, x \rightarrow x - 1)\}$$

should be viewed intuitively as notation for a proof rule of the form: prove some formula F by first proving F under the assumption $x = 0$, with no inductive hypotheses, and then proving F under the assumption $x \neq 0$ and the inductive hypothesis F^x_{x-1}. (Technically the second component of the second pair in the above scheme should be written $\{x \rightarrow x - 1\}$, but we will drop the braces when the set is a singleton.)

Given a formula F to be proved by induction, we must come up with a scheme Σ such that to prove F by the induction scheme Σ is both sound and possible. The first step is to define the "height" of a scheme in a context.

We define the notion of a *height-bound* for a scheme Σ on $x_1, \ldots, x_k$ in a context (Ψ, R), as follows. (a) Let $\Sigma = \{(P_1, S_1), \ldots, (P_m, S_m)\}$, and suppose that P_i is true in (Ψ, R). Let S_i be

$$\{(x_1, \ldots, x_k) \rightarrow (a_{11} \ldots, a_{1k}), \ldots, (x_1, \ldots, x_k) \rightarrow (a_{l1}, \ldots, a_{lk})\}.$$

Suppose that there is an evaluation tree for each a_{ij} in (Ψ, R), with result, say, u_{ij}. Suppose that, for $1 \leq i \leq l$, n_i is a height-bound for Σ in

$$(\Psi^{(x_1)}_{u_{i1}} \cdots {}^{(x_k)}_{u_{ik}}, R).$$

Then $1 + \max_i n_i$ is a height-bound for Σ in (Ψ, R). (If $l = 0$, this condition is vacuously satisfied, and the empty maximum is taken to be zero.) (b) Nothing is a height-bound unless it is because of (a).

If a scheme has no height-bounds in a context, its *height* in the context is ∞; otherwise its height is the minimum of all its height-bounds.

For example, consider the following scheme Σ:

$$\{(x = 0, \{\ \}), (\text{true}, x \rightarrow x + 3), (\text{true}, x \rightarrow x - 5)\}.$$

Let (Ψ, R) be a standard model for Z such that $\Psi(x) = 1$. Then the height of Σ in (Ψ, R) is 6, because it takes five steps to get to 0 from 1 by adding 3's and subtracting 5's (e.g. 1, 4, 7, 2, 5, 0). One way of viewing the height of a scheme is to imagine the scheme as a Dijkstra-style do construct executed in such a way as to reach an empty set of substitutions as fast as possible. However, this intuition must be modified in case some of the S_i contain more than 1 substitution—in this case, the maximum heights of the "branches" must be computed.

To prove that a formula F is valid in a theory (A, R), it suffices to prove that $(A \cup \{\neg F\}, R)$ has no models. To prove by induction that the given theory has no models, we proceed in two steps. First, we construct a scheme Σ such that if $(A \cup \{\neg F\}, R)$ has any models at all, it has a model Ψ such that Σ has finite height in (Ψ, R). Then we prove by induction on the supposed height of Σ that no such model exists.

We illustrate this technique with an example. Let A and R be the axioms and recursive function definitions of Z; that is, let Z be (A, R). To prove that $\mathrm{nn}(x) \supset x \geq 0$ is valid in Z, we show that $(A \cup \{\mathrm{nn}(x) \wedge x < 0\}, R)$ has no models. Informally, it is clear that if Ψ is a model for this theory, then the scheme $\Sigma = \{(x = 0, \{\,\}), (x \neq 0, x \rightarrow x - 1)\}$ has finite height in (Ψ, R), because $\mathrm{nn}(x)$ must halt in (Ψ, R). We show by induction on the supposed height n of Σ in (Ψ, R) that no such model exists. If $n = 1$, then $x = 0$ must be true in (Ψ, R), but this contradicts the fact that $x < 0$ is true in (Ψ, R). If $n > 0$, then $x \neq 0$ must be true in (Ψ, R), and Σ has height $n-1$ in $(\Psi\,{}^{(x)}_{\Psi(x-1)}, R)$. By induction, $\Psi\,{}^{(x)}_{\Psi(x-1)}$ is not a model for $(A \cup \{\mathrm{nn}(x) \wedge x < 0\}, R)$. Thus one of the formulas $\mathrm{nn}(x)$ or $x < 0$ is false in $(\Psi\,{}^{(x)}_{\Psi(x-1)}, R)$; hence one of the formulas $\mathrm{nn}(x-1)$ or $x - 1 < 0$ is false in (Ψ, R). Thus we conclude that Ψ is a model for

$$(A \cup \{\mathrm{nn}(x) \wedge x < 0 \wedge x \neq 0 \wedge \neg(\mathrm{nn}(x-1) \wedge x - 1 < 0)\}, R).$$

Both the formulas $\mathrm{nn}(x) \wedge x \neq 0 \supset \mathrm{nn}(x-1)$ and $x < 0 \supset x - 1 < 0$ can be proved without induction, so the proof is complete.

The technique illustrated by the example generalises in an obvious way to give a proof of the following lemma:

Lemma A. *Let F be a formula, Σ the induction scheme $\{(P_1, S_1), \ldots, (P_m, S_m)\}$ on $x_1, \ldots, x_k$, and (Ψ, R) a context in which Σ has finite height and in which the following formula is true:*

$$(\forall x_1, \ldots x_k) \bigwedge_{1 \leq i \leq m} \Big((P_i \wedge \bigwedge_{\theta \in S_i} F\theta) \supset F \Big). \tag{1}$$

Then F is true in (Ψ, R).

Proof. The proof is a straightforward induction on the height of Σ in (Ψ, R). If this height is one, then for some i, P_i is true in (Ψ, R) and S_i is empty. Since the empty conjunction is true,

$$P_i \wedge \bigwedge_{\theta \in S_i} F\theta$$

is true in (Ψ, R), hence, since (1) is true, so is F.

Suppose the lemma is true whenever the height of Σ is less than n, and suppose the height of Σ in (Ψ, R) is exactly n and that (1) is true in (Ψ, R). Then there is an i such that P_i is true, and furthermore, for each $\theta \in S_i$, $\Psi(x_j\,\theta)$ is defined (and equal to, say, u_j) for each $1 \leq j \leq k$, and $n - 1$ is a height-bound for Σ in $(\Psi\,{}^{(x_1)}_{u_1} \cdots {}^{(x_k)}_{u_k}, R)$. Obviously (1) is true in $(\Psi\,{}^{(x_1)}_{u_1} \cdots {}^{(x_k)}_{u_k}, R)$, so it follows from the induction hypothesis that F is true in $(\Psi\,{}^{(x_1)}_{u_1} \cdots {}^{(x_k)}_{u_k}, R)$. It is easy to deduce from this, using induction on k and structural induction on F, that $F\theta$ is true in (Ψ, R). Hence

$$P_i \wedge \bigwedge_{\theta \in S_i} F\theta$$

is true in (Ψ, R), hence, since (1) is true, so is F. ∎

The lemma shows how to "use" a scheme on a formula, but how is a sound scheme discovered? The example above suggested the answer: the formula F will contain recursive function applications that must halt in order for F to be non-vacuous, and from these a scheme Σ can be constructed that must have finite height in any context in which F is non-vacuous.

Let R be a set of recursive function definitions. A scheme Σ *fits* a term t *for* R if, for any interpretation Ψ such that there exists an evaluation tree T for t in (Ψ, R), there exists a case $(P, \{\theta_1, \dots, \theta_n\})$ of Σ such that P is true in (Ψ, R) and for each θ_i, there exists a proper subtree of T that is an evaluation tree for t with entry interpretation Ψ' that satisfies $\Psi'(x_j) = \Psi(x_j\,\theta_i)$. (In particular, $\Psi(x_j\,\theta_i)$ must be defined for each $1 \leq j \leq k$, $1 \leq i \leq n$.)

In other words, Σ fits t if any evaluation of t will "follow" one of the cases of Σ in the sense that it will lead to recursive evaluations of t in contexts that reflect the substitutions of the case.

Lemma B. *If Σ fits t for R, then Σ has finite height in any context (Ψ, R) in which t halts.*

Proof. The proof is a straightforward induction on the height of an evaluation tree for t. Let the lemma be true whenever the evaluation tree for t has height less than n, and let T be an evaluation tree for t in (Ψ, R) with height n. Since Σ fits t, there is a case $(P, \{\theta_1, \dots, \theta_m\})$ of Σ such that P is true in (Ψ, R) and for each θ_i, there exists a proper subtree of T that is an execution tree for t with entry interpretation Ψ' that satisfies $\Psi'(x_j) = \Psi(x_j\,\theta_i)$. If $m = 0$, then the height of Σ in (Ψ, R) is zero; otherwise the height is at most one greater than the maximum of the heights of Σ in the various contexts (Ψ', R); by the induction hypothesis, these are all finite. ∎

Given a set R of recursive function definitions that contains the definition $f(x_1, \dots, x_k) \equiv B$, it is not difficult to construct an induction scheme Σ on $x_1, \dots, x_k$ that fits B for R, essentially by symbolically evaluating B, constructing one case for every branch of the symbolic evaluation; the case has one substitution for every recursive call on the branch. For example, if R contains the definition

$$\mathbf{f}(x) \equiv [\![\mathbf{p}(x) \Rightarrow \mathbf{true};\ \mathbf{f}(\mathbf{a}(x)) \Rightarrow \mathbf{false};\ \mathbf{f}(\mathbf{b}(x))]\!]$$

then the following scheme is easily seen to fit the body of $\mathbf{f}$:

$$\{\,(\mathbf{p}(x), \{\,\}), (\mathbf{p}(x) \mathrel{\not\doteq} \mathbf{true}, x \rightarrow \mathbf{a}(x)), (\mathbf{p}(x) \mathrel{\not\doteq} \mathbf{true} \wedge \mathbf{f}(\mathbf{a}(x)) \mathrel{\not\doteq} \mathbf{true}, x \rightarrow \mathbf{b}(x))\,\}.$$

(We write $a \mathrel{\not\doteq} b$ as an abbreviation for a halts $\wedge$ b halts $\wedge$ $a \neq b$.) Notice that in contexts in which $\mathbf{p}(x)$ loops, none of the conditions of this scheme are true, but it has never been assumed that the conditions of a scheme were such that one was true in every context. If none of the conditions of the scheme are true in a context, then there is no evaluation tree for the body of $\mathbf{f}$ in the context.

As another example, the scheme associated with

$$\mathbf{f}(x, y) \equiv [\![x = y \Rightarrow \mathbf{true};\ \mathbf{f}(\mathbf{a}(x), y) \mathbin{\dot\vee} \mathbf{f}(\mathbf{b}(x), y)]\!]$$

is

$$\{\,(x = y, \{\,\}), (x \neq y, (x, y) \rightarrow (\mathbf{a}(x), y)), (x \neq y, (x, y) \rightarrow (\mathbf{b}(x), y))\,\}.$$

Notice that the $\dot\vee$ operator leads to multiple cases with the same condition; it would be very wrong to combine these into one case!

Finally, the scheme associated with

$$\mathbf{f}(x) \equiv [\![x > 100 \Rightarrow x - 10;\ \mathbf{f}(\mathbf{f}(x + 11))]\!]$$

is

$$\{\,(x \leq 100, \{\,\}), (x > 100, \{\,x \rightarrow x + 11, x \rightarrow \mathbf{f}(x + 11)\,\})\,\}.$$

Lemmas A and B justify a simple induction rule: to prove a formula F in the theory (A, R) by induction, look for an application of a recursive function in F such that (a) the arguments of the application are distinct variables, and (b) if the application loops, then F is vacuously true. Lemma B then shows how to construct a scheme Σ such that F is vacuously true in those contexts in which Σ has infinite height. Lemma A then shows how to prove that the formula is true in those contexts in which Σ has finite height. (If the arguments of the recursive function application are not distinct variables, it may still be possible to derive an induction scheme from the term, by using the notions of "changeables" and "unchangeables" from [6].)

This simple induction rule is sufficient to justify the inductive proofs in the examples in section 15. On the other hand, the rule is not satisfactory, since, as Boyer and Moore have shown, the induction scheme required to prove a conjecture is often a "combination" of the schemes associated with the various recursive functions appearing in the conjecture. The remainder of this section assumes the reader is familiar with the methods described in [7] for manipulating induction schemes and proving the totality of functions, and sketches how these methods might be adapted to the current system.

Suppose the theory Z is extended with the function definition

$$\mathbf{half}(x) \equiv [\![x = 0 \Rightarrow 0;\ x = 1 \Rightarrow 0;\ 1 + \mathbf{half}(x - 2)]\!],$$

and that we then try to prove the theorem:

$$\mathbf{nn}(x) \supset \mathbf{half}(x) \leq x. \tag{1}$$

By Lemma B, we are justified in using the recursion scheme for **nn**, but this scheme is no help in proving (1). To prove (1) it is essential to use the recursion scheme suggested by the term $\mathbf{half}(x)$. But Lemma B does not justify using this scheme, since it is not obvious that (1) is true when $\mathbf{half}(x)$ loops. To get around this difficulty, it is necessary to show that the scheme for **nn** "justifies" the scheme for **half**, in the sense that the latter has finite height in any context in which the former does. Presumably, this can be done by using the same "transitivity" techniques that Boyer and Moore use to prove at definition time that **half** is total over their system's domain. Similar examples can be given in which the correct scheme is justified as a "lexicographic combination" of two schemes that are justified by Lemma B—for example consider proving:

$$\mathbf{nn}(x) \wedge \mathbf{nn}(y) \supset \mathbf{ack}(x, y)\ \mathbf{halts}$$

where **ack** is Ackerman's function as Boyer and Moore define it.

Thus, the main difference in allowing partial functions appears to be that the definition-time analysis of the Boyer and Moore system must be performed when induction schemes are formulated. Out of all the schemes "suggested" by the terms in the conjecture, some will be known to be sound because of Lemma B and some will not. After justifying as many of the latter schemes as possible, using transitive extensions and lexicographic combinations, a set of valid schemes is obtained. These can then be heuristically manipulated to produce the final scheme.

There are other possible approaches to automating induction. For example, Harry Lewis [37] proves that the equivalence problem for program schemata with non-intersecting loops is decidable. If this result could be generalized to allow the schemata to make equality tests, it would essentially give a satisfiability procedure for the theory of all *linear* recursive function definitions, where a recursive function definition $f(x_1, \ldots, x_n) \equiv B$ is linear if there is only one occurrence of a recursive function symbol in B. Such a satisfiability procedure would be enormously useful; for example, a verifier equipped with such a procedure would be able to verify the list reversal program of section 15 without the aid of the five lemmas, since the functions **reach** and **sequence** are both linear. But it appears to be difficult to generalize Lewis' result.

References

1. W. Ackermann: *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

2. A. V. Aho, J. E. Hopcroft and J. D. Ullmann: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

3. B. Aspvall, M. Plass, and R. Tarjan: "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *Info. Proc. Letters*, vol. 8 no. 3, March 1979.

4. B. Aspvall and Y. Shiloach, "An efficient algorithm for solving systems of linear inequalities with two variables per constraint." *SIAM Journal on Computing*, vol. 9 no. 4, Nov. 1980, pp. 827-45.

5. R. Aubin: *Mechanizing structural induction*, Ph.D. thesis, University of Edinburgh, Edinburgh 1976.

6. W. W. Bledsoe: "The Sup-Inf method in presburger arithmetic." Univ. of Texas Math Dept. Memo ATP-18, December 1974.

7. Robert S. Boyer and J Strother Moore: *A Computational Logic*, Acedemic Press, New York, 1979.

8. Robert S. Boyer and J Strother Moore: "A lemma driven automatic theorem prover for recursive function theory," *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, Department of Computer Science, Carnegie-Mellon University, pp. 511–520 1977.

9. Robert S. Boyer and J Strother Moore: "Proving theorems about LISP functions," JACM, Vol. 22, no. 1, pp. 129-144, 1975.

10. D. Brotz: *Proving theorems by mathematical induction*, Ph.D. thesis, Computer Science Department, Stanford University, 1974.

11. Robert Cartwright and John McCarthy: "Recursive programs as functions in a first order theory", Report STAN-CS-79-717, Stanford University, March 1979.

12. George E. Collins: "Quantifier elimination for real closed fields by cylindrical algebraic decomposition," *Automata Theory and Formal Languages 2nd GI conference*, Ed. G. Goos and J. Hartmanis, Springer-Verlag, Berlin, 1975.

13. Paul Cohen: "Decision procedures for real and *p*-adic fields," *Communications on Pure and Applied Mathematics*, March, 1969, vol. 22 no. 2, pp 131–151.

14. Patrick Cousot and Nicholas Halbwachs: "Automatic discovery of linear restraints among variables of a program," *Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 84–96.

15. W. Craig: "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory", *Journal of Symbolic Logic*, vol. 22.

16. G. Dantzig: *Linear Programming and Extensions.* Princeton University Press, Princeton, New Jersey, 1963.

17. L. P. Deutsch: *An Interactive Program Verifier.* PhD Thesis, Univ. of California, Berkeley, 1973.

18. Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis: "Social processes and proofs of theorems and programs", CACM, vol. 22 no. 5, May 1979, pp. 271–280.

19. E. W. Dijkstra: *Dicipline of Programming*, Englewood Cliffs, N.J. 1976.

20. P. J. Downey, R. Sethi and R. E. Tarjan: "Variations on the common subexpression problem", *JACM* vol. 27 no. 4, October 1980, pp. 758–71.

21. P. Downey and R. Sethi: "Assignment commands with array references", JACM, vol. 25, no. 4.

22. Jon Doyle and Ronald L. Rivest: "Linear expected time of a simple union-find algorithm", *Information Processing Letters*, vol. 5, 1976, pp 146–148.

23. Michael Fischer and Michael Rabin: "Super-exponential complexity of presburger arithmetic," *SIAM-AMS Proceedings*, vol. 7, pp. 27–41, AMS, Providence, RI 1974.

24. Robert Floyd: "Assigning meanings to programs," *Mathematical Aspects of Computer Science, Vol. XIX, Proceedings of Symposia in Applied Mathematics*, pp. 19–32, AMS, Providence, RI, 1967.

25. A. Frankel, M. Garey, D. Johnson, T. Schaefer, and Y. Yesha: "The complexity of checkers on an $n \times n$ board," 19th FOCS conference proceedings, IEEE, 1978 pp. 55-64.

26. David Hilbert: *Grundlagen Der Geometrie*, 1909.

27. David Hilbert: "Mathematical problems", trans. by Mary W. Newson, *Bulletin of the American Mathematical Society*, vol. 8 pp. 437–479.

28. D. M. Kaplan: "Some completeness results in the mathematical theory of computation", JACM, vol. 15 no. 1 pp. 124–34, January, 1968.

29. L. G. Khachian: "Polynomial algorithm for linear programming," Computing Center, Academy Sciences USSR, Moscow, 4 October 1978, Trans. by E.L. Lawler and M. Vlach.

30. J. King: *A Program Verifier.* PhD Thesis, Carnegie-Mellon Univ., 1969.

31. Stephen Kleene: *Introduction to Meta-mathematics*, North-Holland, Amsterdam, 1952.

32. Donald E. Knuth and Peter B. Bendix: "Simple word problems in universal algebras," *Computational Problems in Abstract Algebra*, J. Leech, Ed., pp. 263–297, Pergamon Press, 1969.

33. Donald E. Knuth and Arnold Schönhage: "The expected linearity of a simple equivalence algorithm", *Theoretical Computer Science*, vol. 6 no. 3, pp. 281–315, June 1978.

34. Donald E. Knuth: *The Art of Computer Programming*, Volume 1, Addison-Wesley 2nd ed. 1973.

35. Dexter Kozen: "Complexity of Finitely Represented Algebras", *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, 1977.

36. B. W. Lampson, J. J. Horning, R. London, J. G. Mitchell, and J. Popek: "Revised report on the programming language Euclid," to appear as a Xerox PARC technical report.

37. Harry Lewis: "The equivalence problems for program schemata with nonintersecting loops," Center for Research in Computing Technology, TR-18-76, Harvard University, 1976.

38. David Luckham and Norihisa Suzuki: "Automatic program verification V: verification-oriented proof rules for arrays, records, and pointers," Stanford Artificial Intelligence Laboratory Memo AIM-278, March, 1976.

39. John McCarthy: "A Basis for a Mathematical Theory of Computation", in *Computing Programming and Formal Systems*, edited by P. Braffort and D. Hirshberg, North-Holland.

40. John McCarthy: "Recursive functions of symbolic expressions and their computation by machine," CACM, Vol. 3, no. 4, pp. 184–195 (1960).

41. Y. V. Matiyasevich: "Diophantine Representation of Recursively Enumerable Predicates", *Proceedings of the Second Scandinavian Logic Symposium (Oslo, 1970), Studies in Logic and the Foundations of Mathematics*, Vol 63, pp. 171–177, North-Holland, Amsterdam, 1971.

42. J Moore: "Introducing iteration into the pure LISP theorem prover," *IEEE Transactions on Software Engineering*, Vol. 1, No. 3, pp. 328–338 (1975).

43. C. G. Nelson and D. C. Oppen: "Fast Decision Algorithms based on Congruence Closure", *JACM* vol. 27 no. 2, April 1980, pp. 356–64.

44. C. G. Nelson and D. C. Oppen: "Simplification by cooperating decision procedures", *TOPLAS* vol. 1 no. 2, October 1979, pp. 245–57.

45. C. G. Nelson: "An $n^{\log n}$ algorithm for the two-variable-per-constrant linear programming satisfiability problem", Report STAN-CS-78-689, Stanford, 1978.

46. D. C. Oppen: "Convexity, complexity, and combinations of theories", To appear, *Theoretical Computer Science*, 1980.

47. D. C. Oppen: "Reasoning about recursively defined data structures", *JACM* vol. 27 no. 3 July 1980, pp. 403–11.

48. V. Pratt: "Two easy theories whose combination is hard." Manuscript, September 1977.

49. Dana Scott: "Data types as lattices", *Logic Conference, (Kiel 1974), Lecture notes in mathematics 499*, Springer-Verlag.

50. A. Seidenburg: "A new decision method for elementary algebra", *Ann. of Math.*, Ser. 2, vol. 60, pp. 365–374, 1954.

51. Joseph R. Shoenfield: *Mathematical Logic*, Addison-Wesley, Reading, MA 1967.

52. R. Shostak: "An algorithm for reasoning about equality", CACM, pp. 583–585, July 1978.

53. R. Shostak: "An efficient decision procedure for arithmetic with function symbols", *JACM* vol. 26 no. 2 April 1979, pp. 351–60.

54. R. Shostak: "Deciding linear inequalities by computing loop residues," Manuscript, March 1978.

55. R. Shostak: "On the SUP-INF method for proving Presburger formulas." *JACM*, vol. 24, no. 4, pp. 529–543, October 1977.

56. Stanford Verification Group: "Stanford Pascal Verifier user manual," Report STAN-CS-79-731, Stanford University, March 1979.

57. N. Suzuki and D. Jefferson: "Verification decidability of presburger array programs," *JACM* vol. 27 no. 1 January 1980, pp. 191–205.

58. R. E. Tarjan: "Efficiency of a good but not linear set union algorithm", JACM, pp. 215-225, 1975.

59. A. Tarski: *A Decision Method for Elementary Algebra and Geometry*, (with the assistance of J. C. C. McKinsey), Univ. of Cal. Press, Berkeley and Los Angelas, 1951.

60. Ben Wegbreit: "Property extraction in well-founded property sets," *IEEE Transactions on Software Engineering* vol. 1, no. 3 pp. 270-85.

61. B. Wegbreit, B. Brosgol, G. Holloway, C. Prenner, and J. Spitzen: *ECL Programmer's Manual*, Center for Research in Computing Technology Report 21-72, Harvard University, September 1972.

62. Wu Wen-tsun: "On the decision problem and the mechanization of theorem-proving in elementary geometry," *Scientica Sinica*, vol. 21 no. 2, March–April 1978, pp 159–172.

63. Andrew Yao, "On the average behavior of set merging algorithms" *Proc. ACM Symp. Theory of Computation*, vol. 8, 1976 pp. 192–195.