# Reasoning About Recursively Defined Data Structures

DEREK C. OPPEN

*Stanford University, Stanford, California*

ABSTRACT. A decision procedure is given for the quantifier-free theory of recursively defined data structures which, for a conjunction of length *n*, decides its satisfiability in time linear in *n*. The first-order theory of recursively defined data structures, in particular the first-order theory of LISP list structure (the theory of **cons**, **car**, and **cdr**), is shown to be decidable but not elementary recursive. (This answers an open question posed by John McCarthy.)

KEY WORDS AND PHRASES: recursive data structures, program verification, theorem proving, decidability, logical complexity, simplification

CR CATEGORIES: 4.12, 5.21, 5.24, 5.25, 5.7

## 1. *Introduction*

A wealth of advanced programming tools is being proposed currently—high-level program optimizers, symbolic program testers and debuggers, program verifiers, program transformation systems, program manipulation systems, programming "environments," etc. Such systems depend on an underlying theorem proving capability for reasoning about pieces of program text, for instance, to determine if a transformation preserves correctness or if a program is consistent with its documentation. But until mechanical theorem proving of the sort required by these systems becomes practical (and it is certainly not yet clear that that will ever be the case), such systems will remain experimental toys. As a step in the appropriate direction, we describe in this paper a linear decision procedure for a class of formulas over a very common data structure; the procedure is easily implemented.

We are interested in the decidability and complexity of particular classes of data structures for another reason. Today language designers devote considerable effort to evaluating new and old language features—deciding which should be banned, which tolerated, and which encouraged. The arguments given are generally subjective rather than objective and involve deciding whether programs are "easier" or "harder" to understand with or without particular constructs. One way to make such arguments more precise is to know the decidability and complexity of reasoning about these constructs. It is hoped that this will allow us to be more objective in deciding their fate.

We explore in detail the question of reasoning about a particular class of data structures, the *recursively defined data structures*. These are essentially the *recursive data structures* proposed by Hoare [4] as a structured and constructive alternative to pointers. Most programming languages support such data structures either explicitly or implicitly (they can be mimicked by arrays), but the best known example of them is LISP list structure, with constructor **cons** and selectors **car** and **cdr**.

More precisely, *recursively defined data structures* are data structures which have asso-

This research was supported by the National Science Foundation under Contract MCS 76-000327.

Author's address: Computer Science Department, Stanford University, Stanford, CA 94305.

ciated with them one constructor function $c$, and $k$ selector functions $s_1, \ldots, s_k$ with the following abstract structural properties:

I. (Construction)

$$c(s_1(x), s_2(x), \ldots, s_k(x)) = x$$

II. (Selection)

$$\begin{gathered} s_1(c(x_1, \ldots, x_k)) = x_1 \\ s_2(c(x_1, \ldots, x_k)) = x_2 \\ \vdots \\ s_k(c(x_1, \ldots, x_k)) = x_k \end{gathered}$$

III. (Acyclicity)

$$\begin{gathered} s_1(x) \neq x \\ s_2(x) \neq x \\ \vdots \\ s_k(x) \neq x \\ s_1(s_1(x)) \neq x \\ s_1(s_2(x)) \neq x \\ \vdots \end{gathered}$$

We consider first-order theories (with equality) axiomatized by schemata of the above form.

We give a decision procedure for the quantifier-free theory of recursively defined data structures, which, for a conjunction of equalities and disequalities, determines its satisfiability in linear time. The procedure has possible applications in any theorem prover which handles such data structures, for instance, Boyer and Moore's prover for recursively defined functions [1], Guttag and Musser's prover for abstract data types [3], or the Stanford simplifier [8, 9]. The latter two papers show how a decision procedure for a particular class of data structures such as the one described here can be combined efficiently with decision procedures for other theories to form a decision procedure for their union. The Stanford simplifier contains a decision procedure for the quantifier-free theory of reals, list structure, arrays, Pascal records, and uninterpreted function symbols under $+$, $\leq$, **cons**, **car**, **cdr**, **store**, and **select**.

We also show that it follows that the quantifier-free DNF theory of recursively defined data structures (that is, the quantifier-free theory in which every formula is in disjunctive normal form) is decidable in linear time and that the (full) quantifier-free theory of recursively defined data structures is NP-complete.

Thus far we have restricted our attention to formulas without quantifiers. Consider now theories of recursively defined data structures in which quantification is allowed, in particular the first-order theory of LISP list structure. We use results in the logical literature on "pairing functions" to show that the first-order theory of list structure (recursively defined data structures) is decidable. This answers an open question posed by John McCarthy.

Unfortunately, although decidable, the theory does not admit a practical decision procedure; we show that the theory is not elementary recursive. That is, there cannot exist a decision procedure for it which always halts in time $2^{2^{\cdot^{\cdot^{2^n}}}}$ for any fixed number of 2's ($n$ is the length of the formula). Thus, in the worst case any decision procedure must be egregiously slow.

If one drops the acyclicity axiom schema III, different results obtain. Nelson and Oppen [7] give a decision procedure for the quantifier-free theory of possibly cyclic list structure which, for a conjunction of equalities and disequalities of length $n$, decides its satisfiability in time $O(n^2)$. Downey et al. [2] have improved the underlying graph algorithm to run in time $O(n \log^2 n)$.

## 2. *Decision Procedure for the Quantifier-Free Theory*

2.1 INTRODUCTION. The language of the theory consists of variables, function symbols $c$, $s_1, \ldots, s_k$, and the predicate $=$. The decision procedure described in this section determines the satisfiability of a conjunction of atomic literals in time linear in the length of the conjunction.

Assume we are given a conjunction. The basic strategy of the procedure is to construct a directed graph whose vertices represent the terms of the conjunction and an equivalence relation on the vertices of the graph representing all the equalities that are entailed by the conjunction. The procedure then checks if any asserted disequality conflicts with any of these equalities or if any of the acyclicity axioms are violated. If so, the conjunction is unsatisfiable; otherwise, it is satisfiable.

The algorithm represents terms in the conjunction by (the equivalence classes of) vertices in a directed, acyclic graph, possibly with multiple edges. A vertex in the graph may have outdegree zero or outdegree $k$ (corresponding to the $k$ selector functions). The edges leaving a vertex are ordered. If $u$ is a vertex, then, for $1 \le i \le outdegree(u)$, let $u[i]$ denote the $i$th *successor* of $u$, that is, the vertex to which the $i$th edge of $u$ points. Since multiple edges are allowed, possibly $u[i] = u[j]$ for $i \neq j$.

Every term in the conjunction is either an atomic symbol or an expression of one of the forms $s_i(t)$ or $c(t_1, t_2, \ldots, t_k)$, where $t, t_1, t_2, \ldots, t_k$ are terms. An atomic term $x$ will be represented by a vertex labeled $x$. A term of the form $s_i(t)$ will be represented by a vertex $v$ such that $v = u[i]$ for some vertex $u$ representing $t$. (If necessary, "dummy" successors of $u$ are added to represent the $s_j(t)$, $j \neq i$, if these do not appear in the formula.) A term of the form $c(t_1, t_2, \ldots, t_k)$ will be represented by a node with $k$ successors representing $t_1, t_2, \ldots, t_k$, respectively. To represent the fact that two terms are equal, we will *merge*, that is, make equivalent, the vertices that represent them.

The first step taken by the decision procedure is to construct the graph representing the terms in the conjunction. Vertices representing terms asserted equal in the conjunction are then merged. Vertices representing the same atomic symbol (that is, vertices having the same label) are also merged.

The main work of the algorithm is to *close* the graph under all entailed equivalences of vertices, checking as it does so that no cycles are being introduced into the graph (since such cycles would violate the acyclicity condition). First, if two vertices $u$ and $v$ are equivalent and both have nonzero outdegree, then the equivalence classes of their corresponding successors must be merged (since $x = y \supset s_1(x) = s_1(y) \wedge \cdots \wedge s_k(x) = s_k(y)$). Second, if all the corresponding successors of two vertices $u$ and $v$ with nonzero outdegree are equivalent, then the equivalence classes of $u$ and $v$ must be merged (since $s_1(x) = s_1(y) \wedge \cdots \wedge s_k(x) = s_k(y) \supset x = y$).

The following fragment of a procedure carries out the above step, but does not check for cycles.

```
1. For all pairs of vertices u, v with nonzero outdegree
     if u and v are equivalent
         then (if any corresponding successors of u and v are not equivalent
             then merge the corresponding successors
             also restart step 1)
         else if all the corresponding successors of u and v are equivalent
             then merge u and v
             also restart step 1.
2. Return.
```

This algorithm is nonlinear. The remaining Sections 2.2–2.5 prove the following theorem.

THEOREM 1. *There exists a decision procedure for the quantifier-free theory of recursively defined data structures which, for a conjunction of length n, decides its satisfiability in time linear in n.*

It follows immediately from Theorem 1 that the quantifier-free DNF theory of recursively defined data structures (that is, the quantifier-free theory in which every formula is in disjunctive normal form) is decidable in linear time.

THEOREM 2. *The quantifier-free theory of recursively defined data structures is NP-complete.*

PROOF. Consider an arbitrary formula $F$ of length $n$ in the quantifier-free theory. Consider the disjunctive normal form of $F$. Each disjunct is a conjunction of literals; $F$ is satisfiable if and only if one of these disjuncts is satisfiable. We can guess which atomic formulas in $F$ make up a satisfiable disjunct in nondeterministic polynomial time; the length of the disjunct is $O(n)$. Using our linear decision procedure, we can confirm the satisfiability of this disjunct in time $O(n)$. It follows that the quantifier-free theory is in NP. However, the theory admits arbitrary Boolean structure and so is also NP-hard. Hence the (full) quantifier-free theory of recursively defined data structures is NP-complete. □

2.2 BIDIRECTIONAL CLOSURE. Let $G = (V, E)$ be a directed graph, possibly with multiple edges, such that the edges leaving each vertex are ordered. If $R$ is an equivalence relation on the vertices of $G$, then $G$ is *acyclic under* $R$ if there is no sequence of vertices $u_0$, $u'_0$, $u_1$, $u'_1$, $\ldots$, $u_p = u_0$ of $G$, $p > 0$, such that $\langle u_i, u'_i \rangle \in R$ and $\langle u'_i, u_{i+1}\rangle \in E$ for $0 \leq i < p$.

Let $R$ be an equivalence relation on the vertices of $G$. Define the *congruence closure* $R\uparrow$ of $R$ on $G$ to be the unique minimal extension of $R$ such that (1) $R\uparrow$ is an equivalence relation and (2) any two vertices $u$ and $v$ with equal, nonzero outdegree are equivalent under $R\uparrow$ if all their corresponding successors are equivalent under $R\uparrow$. If $G$ under $R\uparrow$ is acyclic, there are linear algorithms for constructing $R\uparrow$ [2]; the algorithm aborts if $G$ under $R\uparrow$ is not acyclic.

Let $R$ be an equivalence relation on the vertices of $G$. Define the *unification closure* $R\downarrow$ of $R$ on $G$ to be the unique minimal extension of $R$ such that (1) $R\downarrow$ is an equivalence relation and (2) if any two vertices $u$ and $v$ with equal, nonzero outdegree are equivalent under $R\downarrow$, then all their corresponding successors are equivalent under $R\downarrow$. If $G$ under $R\downarrow$ is acyclic, there are linear algorithms for constructing $R\downarrow$ (for instance, the linear unification algorithm of [10]); the algorithm aborts if $G$ under $R\downarrow$ is not acyclic.

We use the notation $R\uparrow$ and $R\downarrow$ to suggest the directional duality of the two notions of closure.

Let $R$ be an equivalence relation on the vertices of $G$. Define the *bidirectional closure* $R\updownarrow$ of $R$ on $G$ to be the unique minimal extension of $R$ such that (1) $R\updownarrow$ is an equivalence relation and (2) for any two vertices $u$ and $v$ with equal, nonzero outdegree, $u$ and $v$ are equivalent under $R\updownarrow$ if and only if all their corresponding successors are equivalent under $R\updownarrow$.

Consider now the problem of constructing the bidirectional closure. First, it is apparent that if a congruence closure algorithm and a unification closure algorithm are run alternately enough times over $G$, eventually $G$ will be bidirectionally closed. That is, $R\downarrow\uparrow\downarrow\uparrow \cdots = R\updownarrow$. However, if $G$ is such that the outdegree of each vertex is either 0 or $k$ for some fixed $k$, then one pass of each algorithm is sufficient by the following lemma.

LEMMA. *Let $G = (V, E)$ be a directed graph with multiple edges such that the edges leaving each vertex are ordered. Assume that the outdegree of each vertex in G is either* 0 *or k for some fixed k. Let R be an equivalence relation on the vertices of G. Then $R\updownarrow = R\downarrow\uparrow$.*

PROOF. It suffices to prove that $R\downarrow\uparrow$ is unification closed.

We first need a property of unification closed relations. Let $R_1$ be a unification closed relation on $G$. Let $u$ and $v$ be a pair of vertices in $G$ with outdegree $k$ such that $\langle u[i]$,

$v[i]\rangle \in R_1$ for all $1 \le i \le k$. Then we claim that the minimal equivalence relation $R_2$ containing $R_1$ and $\langle u, v\rangle$ is also unification closed. Note first that $R_2$ is $R_1$ except that the equivalence classes of $u$ and $v$ have been merged. Consider any pair of vertices $x$ and $y$ with outdegree $k$ such that $\langle x, y\rangle \in R_2$. If $\langle x, y\rangle \in R_1$ then certainly $\langle x[i], y[i]\rangle \in R_2$ for all $1 \le i \le k$. So suppose $\langle x, y\rangle$ is not in $R_1$. Then $\langle x, u\rangle \in R_1$ and $\langle y, v\rangle \in R_1$ (or $\langle x, v\rangle$ and $\langle y, u\rangle$ are in $R_1$). It follows that, for all $1 \le i \le k$, $\langle x[i], u[i]\rangle \in R_1$ and $\langle y[i], v[i]\rangle \in R_1$ (since $R_1$ is unification closed and the outdegree of all the vertices $x, y, u, v$ is $k$), and thus that $\langle x[i], y[i]\rangle \in R_1$, since $\langle u[i], v[i]\rangle \in R_1$ by assumption. Thus, merging $u$ and $v$ did not affect the unification closure property.

Therefore, starting out with $R\downarrow$ and making equivalent any two vertices with outdegree $k$, all of whose corresponding sons are equivalent, leaves the resulting minimal equivalence relation unification closed. By induction, it follows that $R\downarrow\uparrow$ is unification closed. □

It is important for this proof that the vertices have the same outdegree if they have nonzero outdegree. Otherwise, in the above proof it is not necessarily the case that if $\langle x, u\rangle \in R_1$, then all their corresponding successors are equivalent. The order of the passes is also important; $R\uparrow\downarrow$ is not necessarily equal to $R\updownarrow$.

If $G$ under $R\updownarrow$ is acyclic, there is therefore a linear algorithm for constructing $R\updownarrow$. One first constructs $R\downarrow$ using a linear unification closure algorithm and then closes $R\downarrow$ under congruences (that is, constructs $R\downarrow\uparrow$) using a linear congruence closure algorithm. If $G$ under $R\updownarrow$ is not acyclic, one of these algorithms will abort.

2.3 The Decision Procedure. We will now state more precisely the decision procedure described informally in Section 2.1. We start by describing the data structures manipulated by the procedure.

First, corresponding to every term $t$ in a formula, there is a directed, acyclic graph $G(t)$. $G(t)$ will contain a vertex $V_{G(t)}(t)$ "representing" $t$.

(1) If $t$ is an atomic symbol, $G(t)$ has a single vertex with zero outdegree labeled with $t$. $V_{G(t)}(t)$ will be this vertex.

(2) If $t$ is of the form $s_i(\alpha)$, then $G(t)$ will be $G(\alpha)$ and $V_{G(t)}(\beta)$ will be $V_{G(\alpha)}(\beta)$ for all subexpressions $\beta$ of $\alpha$. However, if $V_{G(\alpha)}(\alpha)$ has outdegree 0, we will add $k$ successors to $V_{G(t)}(\alpha)$ (each successor will be a new unlabeled vertex with outdegree 0). In either case, $V_{G(t)}(t)$ will be the $i$th successor of $V_{G(t)}(\alpha)$.

(3) If $t$ is of the form $c(\alpha_1, \ldots, \alpha_k)$, then $G(t)$ is the disjoint union of $G(\alpha_1), \ldots, G(\alpha_k)$ together with a new vertex $u$ with $k$ successors. For all $1 \le i \le k$, $u[i]$ is $V_{G(\alpha_i)}(\alpha_i)$. $V_{G(t)}(t)$ is $u$. (In taking the disjoint union, we will always assume that the label of any vertex in the union is its old label in the graphs whose union we are taking. Similarly, for any term $\beta$, if $V_{G(\alpha_i)}(\beta)$ exists in $G(\alpha_i)$, then $V_{G(t)}(\beta)$ will be the same vertex.)

Notice that the only labeled vertices are those representing atomic terms, and that all vertices either have outdegree 0 or outdegree $k$.

In what follows, we may refer to $V(t)$ instead of $V_{G(t)}(t)$.

*Decision Procedure.* This algorithm determines the satisfiability of a conjunction $F$ of the form

$$v_1 = w_1 \wedge \cdots \wedge v_r = w_r \wedge x_1 \neq y_1 \wedge \cdots \wedge x_s \neq y_s.$$

1. Construct $G$, the disjoint union of $G(v_1), \ldots, G(v_r), G(w_1), \ldots, G(w_r), G(x_1), \ldots, G(x_s), G(y_1), \ldots, G(y_s)$. Let $R$ be $\{(V(v_i), V(w_i)) \mid 1 \le i \le r\} \cup \{(\alpha, \beta) \mid \alpha$ and $\beta$ are vertices in $G$ with the same label$\}$. That is, the initial equivalence relation $R$ makes equivalent vertices representing terms asserted equal in $F$ and vertices representing the same atomic term in $F$.

2. Construct $R\updownarrow$, the bidirectional closure of $R$ on $G$. Let $\llbracket u \rrbracket$ denote the equivalence class of vertex $u$ in $G$ under $R\updownarrow$. If $G$ under $R\updownarrow$ is not acyclic, return **unsatisfiable**.

3. For $i$ from 1 to $s$, if $\llbracket V(x_i) \rrbracket = \llbracket V(y_i) \rrbracket$, return **unsatisfiable.** Otherwise, return **satisfiable**.

2.4 CORRECTNESS OF THE DECISION PROCEDURE. It is straightforward to verify that the algorithm is correct if it returns **unsatisfiable**. Suppose that it returns **satisfiable**; we will construct an interpretation satisfying $F$.

Let $R_0$ be the partition of the vertices of $G$ corresponding to the final equivalence relation $R\updownarrow$. We define $k$ functions $s_{10}, \ldots, s_{k0}$ from a subset of $R_0$ to $R_0$, and a function $c_0$ from a subset of $R_0^k$ to $R_0$. For $1 \leq i \leq k$, an equivalence class $Q$ is in the domain of $s_{i0}$ if $Q$ contains a vertex $u$ with outdegree $k$; in this case, $s_{i0}(Q) = \llbracket u[i] \rrbracket$. (Since every vertex in $G$ has outdegree either 0 or $k$, $Q$ is in the domain of a particular $s_{i0}$ if and only if it is in the domain of $s_{i0}$ for all $1 \leq i \leq k$.) A $k$-tuple $(Q_1, \ldots, Q_k)$ of equivalence classes is in the domain of $c_0$ if there exists a vertex $u$ with outdegree $k$ such that $u[i] \in Q_i$ for $1 \leq i \leq k$; in this case, $c_0(Q_1, \ldots, Q_k) = \llbracket u \rrbracket$. Note that $c_0, s_{10}, \ldots, s_{k0}$ are well defined, since $G$ is bidirectionally closed and every vertex in $G$ has outdegree either 0 or $k$. However, these functions are not necessarily defined over the whole of $R_0^k$ and $R_0$. To construct an interpretation we must extend these functions; in the process we will construct an infinite domain for the interpretation. We now describe this construction.

Let $G_0 = G$. Construct as above the tuple $(G_0, R_0, c_0, s_{10}, \ldots, s_{k0})$. Suppose we have constructed the first $j + 1$ tuples $(G_0, R_0, c_0, s_{10}, \ldots, s_{k0}), \ldots, (G_j, R_j, c_j, s_{1j}, \ldots, s_{kj}), \ldots$. Construct $(G_{j+1}, R_{j+1}, c_{j+1}, s_{1j+1}, \ldots, s_{kj+1})$ to be the following extension of $(G_j, R_j, c_j, s_{1j}, \ldots, s_{kj})$:

(1) For each equivalence class $Q$ of $R_j$ which is not in the domain of any $s_{ij}$, choose any vertex $u$ in $Q$ ($u$ therefore has outdegree 0 in $G_j$). In $G_{j+1}$ add $k$ new vertices as successors to $u$, each in an equivalence class of its own in $R_{j+1}$. Let $c_{j+1}(\llbracket u[1] \rrbracket, \ldots, \llbracket u[k] \rrbracket) = Q$ and $s_{ij+1}(Q) = \llbracket u[i] \rrbracket$, for $1 \leq i \leq k$. By this construction the domain of $s_{ij+1}$ is $R_j$.

(2) For each tuple $(Q_1, \ldots, Q_k)$ of equivalence classes of $R_j$ not in the domain of $c_j$, add a new vertex $u$ to $G_{j+1}$ in an equivalence class of its own in $R_{j+1}$. Let $u$ have outdegree $k$, and for $1 \leq i \leq k$, let $u[i] = v$ for some $v$ in $Q_i$. (Since $(Q_1, \ldots, Q_k)$ is not in the domain of $c_j$, there is no other vertex $w$ in $G_{j+1}$ with outdegree $k$ such that $w[i] \in Q_i$ for $1 \leq i \leq k$.) Let $c_{j+1}(Q_1, \ldots, Q_k) = \llbracket u \rrbracket$, and, for $1 \leq i \leq k$, let $s_{ij+1}(\llbracket u \rrbracket) = Q_i$. By this construction the domain of $c_{j+1}$ is $R_j^k$.

$G_{j+1}$ is thus $G_j$ except for the new vertices $u_1, \ldots, u_p$ added in steps 1 and 2 above. $R_{j+1}$ is $R_j$ together with the additional singleton equivalence classes $\llbracket u_1 \rrbracket, \ldots, \llbracket u_p \rrbracket$. $c_{j+1}, s_{1j+1}, \ldots, s_{kj+1}$ are $c_j, s_{1j}, \ldots, s_{kj}$ extended as described in steps 1 and 2 above. The extensions are well defined. Notice in particular that if any $s_{ij+1}(Q)$ is defined, then all the $s_{ij+1}(Q)$ are defined.

LEMMA. *Suppose $Q, Q_1, \ldots, Q_k$ are equivalence classes of $R_j$. Then the following hold:*

(1) *If $Q$ is in the domain of $s_{ij}$, for $1 \leq i \leq k$, then $(s_{1j}(Q), \ldots, s_{kj}(Q))$ is in the domain of $c_j$ and $c_j(s_{1j}(Q), \ldots, s_{kj}(Q)) = Q$.*

(2) *If $(Q_1, \ldots, Q_k)$ is in the domain of $c_j$, then $c_j(Q_1, \ldots, Q_k)$ is in the domain of $s_{ij}$, and $s_{ij}(c_j(Q_1, \ldots, Q_k)) = Q_i$ for $1 \leq i \leq k$.*

(3) *$G_j$ under $R_j$ is acyclic.*

PROOF. *Base step*: $j = 0$. If $Q$ is in the domain of the $s_{i0}$, then there exists a vertex $u$ in $Q$ with outdegree $k$. Therefore $(s_{10}(Q), \ldots, s_{k0}(Q))$ is in the domain of $c_0$ and $c_0(s_{10}(Q), \ldots, s_{k0}(Q)) = Q$. So the first clause of the lemma holds. If $(Q_1, \ldots, Q_k)$ is in the domain of $c_0$, then there is a vertex $u$ with outdegree $k$ such that $c_0(Q_1, \ldots, Q_k) = \llbracket u \rrbracket$, and, for $1 \leq i \leq k$, $u[i] \in Q_i$. Therefore, for $1 \leq i \leq k$, $c_0(Q_1, \ldots, Q_k)$ is in the domain of $s_{i0}$, and $s_{i0}(c_0(Q_1, \ldots, Q_k)) = \llbracket u[i] \rrbracket = Q_i$. So the second clause of the lemma holds. Since $G_0$ under $R_0$ is acyclic, the third clause holds.

Suppose the lemma holds for $j$; we show it also holds for $j + 1$.

*Proof of Clause* 1. If $Q$ is in the domain of one (and hence all) of $s_{1j}, \ldots, s_{kj}$, then the result follows from the induction hypothesis and the fact that $(G_{j+1}, R_{j+1}, c_{j+1}, s_{1j+1}, \ldots, s_{kj+1})$ extends $(G_j, R_j, c_j, s_{1j}, \ldots, s_{kj})$. If $Q$ is not in the domain of the $s_{ij}$, then in constructing $(G_{j+1}, R_{j+1}, c_{j+1}, s_{1j+1}, \ldots, s_{kj+1})$ we added $k$ vertices as successors to some vertex $u$ in $Q$ and defined $c_{j+1}(s_{1j+1}(Q), \ldots, s_{kj+1}(Q)) = c_{j+1}(\llbracket u[1] \rrbracket, \ldots, \llbracket u[k] \rrbracket) = Q$.

*Proof of Clause* 2. If $(Q_1, \ldots, Q_k)$ is in the domain of $c_j$, then the result follows from the induction hypothesis and the fact that $(G_{j+1}, R_{j+1}, c_{j+1}, s_{1j+1}, \ldots, s_{kj+1})$ extends $(G_j, R_j, c_j, s_{1j}, \ldots, s_{kj})$. Otherwise, in constructing $(G_{j+1}, R_{j+1}, c_{j+1}, s_{1j+1}, \ldots, s_{kj+1})$ we added a vertex $u$ such that $[\![u]\!] = c_{j+1}(Q_1, \ldots, Q_k)$, and $u[i] \in Q_i$, for $1 \leq i \leq k$. $c_{j+1}(Q_1, \ldots, Q_k)$ is thus in the domain of $s_{ij+1}$ and $s_{ij+1}(c_{j+1}(Q_1, \ldots, Q_k)) = Q_i$ for $1 \leq i \leq k$. The second clause therefore holds.

The third clause holds from the construction. □

Let $R'$ be the union of the $R_i$. Let $s_i'(Q)$ be $s_{ij}(Q)$ for the first $j$ such that $s_{ij}(Q)$ is defined. Let $c'$ be defined similarly. It follows that $c', s_1', \ldots, s_k'$ satisfy the axioms and are defined on all of $R_k'$.

We will now define an interpretation $\psi$ which satisfies $F$. $\psi$ interprets $c, s_1, \ldots, s_k$ as $c'$, $s_1', \ldots, s_k'$. It follows that this interpretation satisfies the axioms. It remains to show that $\psi$ satisfies $F$. It is straightforward to show that for every term $t$ in the formula, $\psi(t) = [\![V(t)]\!]$. But $V(t_i)$ and $V(w_i)$ have been merged for $1 \leq i \leq r$, so $\psi$ satisfies the equalities in $F$. $V(x_i)$ and $V(y_i)$ are in different equivalence classes since step 3 returned **satisfiable**, so $\psi$ satisfies the disequalities in $F$.

2.5 Linearity of the Decision Procedure. $G$ can be constructed in several ways, but some care must be taken if it is to be constructed in linear time, that is, in time $O(n)$ where $n$ is the length of the formula $F$. We describe one way of doing so.

Step 1. For each term $t$ in the formula, we construct $G(t)$. We do not bother to identify common subexpressions; distinct occurrences of similar subterms of $t$ will be represented by distinct vertices in $G(t)$. However, we keep a list of pairs $\langle t, V(t)\rangle$ for each term $v_i$, $w_i$, $x_i$, and $y_i$ in the formula. We also keep a list of pairs $\langle a, V(a)\rangle$ for each occurrence of each atomic symbol $a$ in the formula. We then form $G$, the disjoint union of these graphs. The number of vertices and edges in $G$ is $O(n)$, and the time required to construct $G$ is also $O(n)$.

Step 2. We next add to the graph the equalities asserted in the formula by merging vertices $V(v_i)$ and $V(w_i)$ for each equality $v_i = w_i$ in the formula. Since in step 1 we kept track of each $V(v_i)$ and $V(w_i)$, we can do step 2 in time $O(n)$.

Step 3. We now make equivalent all vertices with the same label. Each such vertex represents an atomic symbol in the original formula and so appears in the list of pairs $\langle a, V(a)\rangle$ constructed in step 1. Under a reasonable model, we can sort this list on the first argument of each pair $\langle a, V(a)\rangle$ in time $O(n)$ using lexicographic sorting. We then scan this list; for each pair of adjacent elements $\langle a_1, V(a_1)\rangle$ and $\langle a_2, V(a_2)\rangle$ in this list, if $a_1 = a_2$, then we make equivalent $V(a_1)$ and $V(a_2)$. This step again takes time $O(n)$.

(In practice, this elaborate method would not be used. Instead, we would use a hash table to store $V(a)$ for each $a$ and would never create two vertices with the same label. Languages such as LISP support this very efficiently.)

Step 4. Next we construct $G$, the bidirectional closure of the relation on $G_0$ constructed in the previous steps. Again we can do this in linear time, as shown in Section 2.2. Notice that in constructing the bidirectional closure, we automatically identify (make equivalent) all common subexpressions. The algorithm will abort if cycles appear in the graph, in which case we return **unsatisfiable**.

Step 5. Finally, we check for any $i$ from 1 to $s$ if $[\![V(x_i)]\!] = [\![V(y_i)]\!]$. Since in step 1 we kept track of each $V(x_i)$ and $V(y_i)$, we can do step 5 in time $O(n)$.

## 3. *The First-Order Theory*

For concreteness, we will consider the first-order theory of list structure (with function symbols **cons**, **car**, and **cdr**, and predicate symbols = and **atom**).

First, the decision procedure given in the previous section for quantifier-free conjunctions can be modified to be the basis for a quantifier-elimination method for this theory. However, as we will see, any decision procedure for the theory must be (asymptotically) very inefficient, and so we will content ourselves with an indirect proof.

A *pairing function* on a set $S$ is a one-to-one map $J: S \times S \rightarrow S$. An example of a pairing function over the natural numbers is the function $J(x, y) = 2^x 3^y$.

Associated with each pairing function $J$ are its projection functions $K$ and $L$. These are partial functions $S \rightarrow S$ satisfying $K(J(x, y)) = x$ and $L(J(x, y)) = y$. Since $K$ and $L$ are partial, we will formally consider all functions as relations but will continue to write, for instance, $K(z) = x$ instead of $K(z, x)$. (An alternative would be to make all functions total by introducing $\perp$, the undefined element, into the logic.)

$K$ and $L$ satisfy the axioms

(1) $\forall x \forall y \exists ! z [K(z) = x \wedge L(z) = y]$.

(2) $\forall z [\exists x (K(z) = x \vee L(z) = x) \supset \exists ! x \exists ! y (K(z) = x \wedge L(z) = y)]$.

The pairing function $J$ is defined in terms of $K$ and $L$ by $J(x, y) = z \equiv K(z) = x \wedge L(z) = y$.

The first-order theory of pairing functions (the first-order theory with these axioms) is undecidable (unpublished results by Hanf, Scott, and Morley). However, with appropriate additional axioms the theory is decidable. These additional restrictions on $K$ and $L$ correspond to the acyclicity condition we put on our recursively defined data structures together with the decidability of the theory of atoms.

First we partition the set $S$ into two disjoint parts, the set $A$ of *atoms* and the set $S - A$ of nonatoms; **atom**$(x)$ holds if and only if $x$ is an atom.

The following infinite axiom schema requires that the pairing function be acyclic on all nonatoms.

(3) (Acyclicity)
$\forall z [\neg \mathbf{atom}(z) \wedge \exists x (K(z) = x) \supset K(z) \neq z]$.
$\forall z [\neg \mathbf{atom}(z) \wedge \exists x (L(z) = x) \supset L(z) \neq z]$.
$\forall z [\neg \mathbf{atom}(z) \wedge \exists x (K(L(z)) = x) \supset K(L(z)) \neq z]$.
$\vdots$

Next, if $z$ is not an atom, it must have projections.

(4) $\forall z [\neg \mathbf{atom}(z) \supset \exists x (K(z) = x)]$.
$\forall z [\neg \mathbf{atom}(z) \supset \exists x (L(z) = x)]$.

Finally, once an element $z$ lies in $A$, all iterations of projection functions from $z$ (as long as they are defined) must lie in $A$.

(5) $\forall z [\mathbf{atom}(z) \wedge \exists x (K(z) = x) \supset \mathbf{atom}(K(z)) \wedge \mathbf{atom}(L(z))]$.

A pairing function satisfying these axioms is defined to be *acyclic except for* $A$.

If $A$ is empty, the first-order theory with the above as axioms is decidable [5, 6]. If $A$ is nonempty, the theory may or may not be decidable: Tenney [12] reduces the question of decidability to the decidability of the theory restricted to the atoms; if the latter is decidable then so is the former.

Consider the first-order theory of list structure. **cons** is the pairing function $J$, **car** is the left projection $K$, **cdr** is the right projection $L$, $S$ is the set of $s$-expressions, and $A$ is the set of atoms. It follows easily from [12] that the first-order theory of list structure is decidable if the theory of atoms under **car**, **cdr**, and $=$ is decidable.

There are many possible choices for $A$ and its associated theory. First, $A$ might be infinite (as in LISP) or consist of the single atom **nil** (as in Boyer and Moore's original prover). Second, **car** and **cdr** may or may not be defined on all or some of the atoms. If defined, **car** and **cdr** may be cyclic or acyclic (for instance, we might choose **car(nil)** and **cdr(nil)** to be **nil** as in MACLISP). Regardless of the choice, as long as the theory of atoms is decidable so is the overlying theory of list structure. For a reasonable choice of the theory of atoms its decidability is apparent.

Therefore, for any "reasonable" axiomatization of the theory of LISP list structure, its first-order theory is decidable. Unfortunately, an efficient decision procedure for the theory cannot exist.

Rackoff [11] has shown that no theory of pairing functions admits an elementary recursive decision procedure, that is, one which always halts in time $2^{2^{\cdot^{\cdot^{2^n}}}}$ for any fixed number of 2's ($n$ is the length of the formula). It follows that any decision procedure for the theory of list structure must be very inefficient in the worst case.

Although Tenney [12] proved his result for pairing functions $S \times S \rightarrow S$, his argument holds as well for $k$-ary pairing functions, that is, pairing functions $S^k \rightarrow S$ which satisfy the obvious generalization of the above axioms. Similarly, Rackoff [11] proves that his lower bound also applies to any $k$-ary pairing function. It follows that given a recursive data structure with constructor $c$ and selectors $s_1, \ldots, s_k$ satisfying the obvious generalization of the above axioms, the associated first-order theory is decidable but not elementary recursive.

The contents of this section can be summarized as follows.

THEOREM 3. *The first-order theory of recursively defined data structures is decidable but not elementary recursive.*

ACKNOWLEDGMENTS. I am indebted to Greg Nelson, Dave Stevenson, and Bob Tarjan for numerous helpful discussions.

REFERENCES

1. BOYER, R., AND MOORE, J. A lemma driven automatic theorem prover for recursive function theory. Proc. of the Fifth Int. Joint Conf. on Artificial Intelligence, Tdilisa, USSR, 1977.
2. DOWNEY, P., SETHI, R., AND TARJAN, R. Variations on the common subexpression problem. To appear *J. ACM.*
3. GUTTAG, J., HOROWITZ, E., AND MUSSER, D. Abstract data types and software validation. *Comm. ACM 21*, 12 (Dec. 1978), 1048-1064.
4. HOARE, C.A.R. Recursive data structures. *Int. J. Comptr. Inform. Sci.* (June 1975).
5. MAL'CEV, A. On the elementary theories of locally free universal algebras. *Soviet Math.* [Doklady] (1961).
6. MAL'CEV, A. Axiomatizable classes of certain types of locally free algebras. *Sibirskii Matematicheskii Zhurnal* (1962).
7. NELSON, C.G., AND OPPEN, D.C. Fast decision procedures based on congruence closure. *J. ACM 27*, 2 (April 1980), 356-364.
8. NELSON, C.G., AND OPPEN, D.C. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Syst. 1*, 2 (Oct. 1979), 245-257.
9. OPPEN, D.C. Convexity, complexity, and combinations of theories. To appear *Theoret. Comptr. Sci.*
10. PATERSON, M., AND WEGMAN, M. Linear unification. To appear *J. Comptr. Syst. Sci.*
11. RACKOFF, C. The computational complexity of some logical theories. Ph.D. Th., M.I.T., Cambridge, Mass. 1975.
12. TENNEY, R. Decidable pairing functions. To appear *J. Symbolic Logic.*

RECEIVED JUNE 1978; REVISED AUGUST 1979; ACCEPTED AUGUST 1979