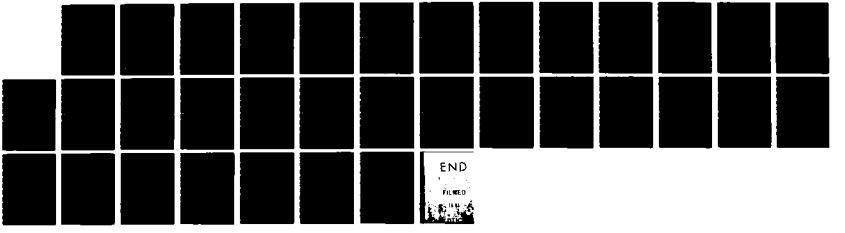
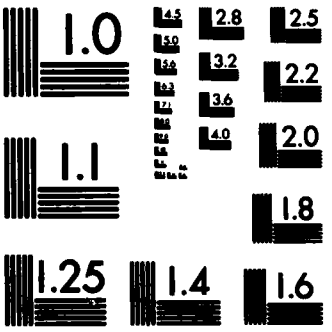
AD-A133 613 SOLVING UNINTERPRETED EQUATIONS WITH CONTEXT FREE EXPRESSION GRAMMARS(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB D A MCALLESTER

1/1

UNCLASSIFIED MAY 83 AI-M-708 N00014-80-C-0505 F/G 5/7 NL





UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AIM 708	2. GOVT ACCESSION NO.: AD-A133 613
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Solving Uninterpreted Equations with Context Free Expression Grammars	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): David Allen McAllester	8. CONTRACT OR GRANT NUMBER(s): N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Massachusetts 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, Virginia 22209	12. REPORT DATE: May 1983
	13. NUMBER OF PAGES: 31
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution is unlimited

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

theorem proving
congruence closure
simplification
automated deduction
context free grammar

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

It is shown here that the equivalence cass of an expression under the congruence closure of any finite set of equations between ground terms is a context free expression language. An expression is either a symbol or an n-tuple of expressions the difference between expressions and strings is that expressions have inherent phrase structure. The Downey, Sethi, and Tarjan algorithm for computing congruence closures can be used to convert a finite set of equations $\mathcal{E}$ to a context free expression grammar G such that for any expression u the equivalence class of of u under $\mathcal{E}$ is precisely the language generated by an expression form $\Gamma(u)$ under the grammar G. The fact that context tree expression languages are closed under

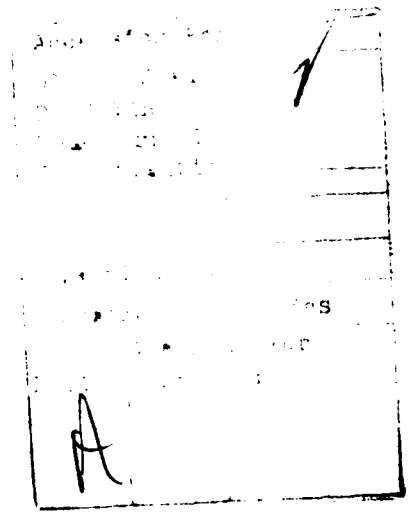
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

OVER

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

intersection is used to derive an algorithm for computing a grammar for the equivalence class of a given expression under any finite disjunction of finite sets of equations between ground expressions. This algorithm can also be used to derive a grammar representing the equivalence class of conditional expressions of the form if P then u else v. The description of an equivalence class by a context free expression grammar can also be used to simplify expressions under "well behaved" simplicity orders. Specifically if G is a context free expression grammar which generates an equivalence class of expressions then for any well behaved simplicity order there is a subset G' of the productions of G such that the expressions generated by G' are exactly those expressions of the equivalence class which are simplicity bounds and whose subterms are also simplicity bounds. Furthermore G' can be computed from G in order nlog(n) time plus the time required to do order nlog(n) comparisons between expressions where n is the size G.



Massachusetts Institute of Technology
Artificial Intelligence Laboratory

AI Memo No. 708 May 1983

Solving Uninterpreted Equations with Context Free Expression Grammars

by

David Allen McAllester

Abstract:

It is shown here that the equivalence class of an expression under the congruence closure of any finite set of equations between ground terms is a context free expression language. An expression is either a symbol or an n-tuple of expressions; the difference between expressions and strings is that expressions have inherent phrase structure. The Downey, Sethi, and Tarjan algorithm for computing congruence closures can be used to convert a finite set of equations Σ to a context free expression grammar G such that for any expression u the equivalence class of u under Σ is precisely the language generated by an expression form $\Gamma(u)$ under the grammar G. The fact that context free *expression* languages are closed under intersection is used to derive an algorithm for computing a grammar for the equivalence class of a given expression under any finite disjunction of finite sets of equations between ground expressions. This algorithm can also be used to derive a grammar representing the equivalence class of conditional expressions of the form **if** P **then** u **else** v. The description of an equivalence class by a context free expression grammar can also be used to simplify expressions under "well behaved" simplicity orders. Specifically if G is a context free expression grammar which generates an equivalence class of expressions then for any well behaved simplicity order there is a subset G' of the productions of G such that the expressions generated by G' are exactly those expressions of the equivalence class which are simplicity bounds and whose subterms are also simplicity bounds. Furthermore G' can be computed from G in order $n\log(n)$ time plus the the time required to do order $n\log(n)$ comparisons between expressions where n is the size G.

Keywords: Theorem Proving, Automated Deduction, Congruence Closure, Context Free Grammar, Simplification

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

Acknowledgments: There are many people who supported and encouraged this work. Ed Barton encouraged me to consider the problem of simplification under an arbitrary Boolean formula and suggested that an equivalence class might representable as a grammar. Gerald Sussman, Guy Steele, Vaughn Pratt, and Derek Oppen were all instrumental in both motivating this work and in furthering my understanding of the problems involved. Chuck Rich, Jerry Roylance, Howie Shrobe, and Ken Forbus provided many stimulating discussions.

CONTENTS

1. INTRODUCTION

There are several areas in which it is important to be able to "solve" for certain expressions. The most familiar example is in applied mathematics where one is interested in "solving" a system of simultaneous equations. A related problem is that of answering data base queries of the form "what is x?". In general one is presented with a collection of terms, some methods for showing equivalences between terms, and the problem of "solving for" or "simplifying" some particular term.

One motivation for algorithms which solve for expressions arises from procedural attachment. Consider a data base consisting of a collection of first order sentences and suppose that the binary function symbol + is intended to denote ordinary addition over the integers. Clearly it is possible to directly compute the sum of two numerals and thus one can benefit from associating the function symbol + with such an addition procedure (such procedural attachment in FOL is described by Weyhrauch [1]). However to apply the addition procedure to a term such as **+[f[x], f[y]]** one must solve for its subterms, in this case **f[x]** and **f[y]**, in terms of numeric constants. Thus the usefulness of procedures attached to function or predicate symbols can depend upon the ability to solve for expressions.

Another motivation for algorithms that solve for expressions arises from Moore's notion of a rigid designator [2]. Moore addresses the philosophical question of when an individual (or a data base) "knows" the meaning of some term. For example when does an individual "know" John's phone number. Moore addresses this question by first defining a modal logic with a Kripke style semantics and then defining the notion of a "rigid designator" as a term, such as the numeral 2, which has the same meaning in all possible worlds. Moore then defines "knowing the meaning of a term" such that a person knows the meaning of the term **phone#[John]** just in case there is a rigid designator *d* such that the person can deduce that **phone#[John]** equals *d*. The details of Moore's constructions are not important here other than to note that he makes a distinction between terms which may denote different things in different worlds and rigid designators which can not. Now consider a data base consisting of sentences in Moore's logic and a question of the form "what is **phone#[John]**?". Intuitively one should expect the system to either answer "I don't know", or to return a rigid designator such as the numeral 2537884. Thus the problem of answering a query of the form "what is x?" can be reduced to the problem of "solving" for x in terms of a rigid designator.

In general suppose that some set of symbols has been identified as "independent" (or rigid in Moore's sense) and all other symbols are considered "dependent". The problem addressed here is that of taking a finite set of equalities Σ between ground terms and a particular term u and "solving" for u in terms of the independent symbols by performing substitutions of equals for equals. As an example consider the three equations **a = f[f[f[a]]]**, **b = f[a]** and **c = f[b]** and suppose that the symbols f and c are taken to be independent. It is possible to solve for b in terms of f and c, specifically **b = f[f[c]]**. As another example consider the equations **a = f[b c]**, **b = g[a c]**, and **c = f[g[a c] c]** where g and c are taken to be independent. It follows from these equalities that **b = g[c c]**.

The only technique for showing equivalences used by the procedures described here is the

substitution of a ground expression for an equivalent ground expression. The problem of solving for an expression using substitution as the only means of showing equivalences may seem like a very special case of the problem of solving for an expression using arbitrary techniques for showing equivalences. However the substitution of equals for equals can play an important role in cases where other techniques for showing equivalences are also sound. For example consider a set of simultaneous linear equations which can be "solved" using some standard matrix inversion procedure. Steele and Sussman [3] propose an alternative to the standard matrix inversion techniques which they call "constraint propagation". Constraint propagation can be more efficient than matrix inversion when the matrix of coefficients is sparse. Constraint propagation also seems to model the way people often solve sets of equations. In constraint propagation a set of equations is expanded to a larger set by solving each equation locally for each variable appearing in that equation. There is then a "propagation" phase based purely on the substitution of equals for equals. This "propagation" phase of the process could be handled by the algorithms presented here. A more complete discussion of the relationship between constraint propagation and substitution of equals for equals is given in [4].

The problem of solving for an expression given a set of equalities between ground terms can be usefully generalized in two ways. First instead of considering a set of equations one can consider an arbitrary Boolean formula built up from equalities between ground terms. For example the following Boolean formula implies the equality $f(x)=1$.

$$(x=a \vee x=b) \wedge f[a]=1 \wedge f[b]=1$$

A second way the problem can be generalized is to consider simplification under some simplicity order on terms. Solving for an expression is just a special case of simplification. To see this consider the partial order on terms defined by making a term u "simpler" than a term v just in case v contains dependent symbols while u does not. Under such a simplicity order solving for a given term is equivalent to "simplifying" that term to an expression containing only independent symbols.

The importance of the more general problem of simplifying terms under a predefined simplicity order can be seen by considering a set of simultaneous equations in applied mathematics. Numerals and terms constructed purely from numerals and "known" function symbols such as + are in some sense simpler than other terms. Often symbolic "constants" are used in applied mathematics and terms constructed purely from numeric and symbolic constants are in some sense simpler than terms which contain "variables". Variables are sometimes divided into dependent and independent variables and terms which contain no dependent variables are in some sense simpler than terms which do not. Also the size (or some other measure of complexity) can be an important factor in determining a terms simplicity.

The above problems are approached here by establishing a relationship between equivalence classes and context free expression grammars. A context free expression grammar is just like a context free string grammar except that it describes a set of expressions (terms) rather than a set of strings (it is important to note that an expression is a tree and therefore has an inherent phrase structure while a string has no such

structure). A finite set Σ of equalities between ground expressions induces a congruence relation on expressions where two expressions are congruent just in case they can be proven equal from the equalities in Σ. Such a congruence relation will be called a finitely equational congruence relation. A finitely equational congruence relation can be represented by a context free expression grammar. Given a finite set Σ of equalities between expressions the congruence closure algorithm of Downey, Sethi, and Tarjan [5] can be used to construct a context free expression grammar G such that for any expression u the equivalence class of u is the language generated by an expression form Γ(u) under G. Furthermore G can be constructed from Σ in order $n\log(n)$ average time where n is the size of Σ.

Let u be an arbitrary expression and let Q be an arbitrary consistent Boolean expression built up from equalities and the standard Boolean connectives ¬, ∧, and ∨. Using the fact that context free *expression* languages are closed under intersection it is possible to construct a context free expression grammar G which generates the set of expressions which can proven equivalent to u given the formula Q. This procedure can be used to compute a grammar for the class of simple expressions which are equivalent to a conditional expression of the form **if** Q **then** u **else** v.

Finally a procedure is developed for simplifying expressions under an arbitrary well behaved simplicity order. Let G be a grammar such that the equivalence class of an expression u is the language generated by an expression form Γ(u) under G. For any well behaved simplicity order there is a subset G' of the productions of G such that the language generated by Γ(u) under G' is precisely the set of simplifications of u. The subset G' can be computed from G in order $n\log(n)$ time plus the time required to do order $n\log(n)$ comparisons between expressions where n is the size of G.

It is hoped that the ability to represent finitely equational congruence relations with context free expressions grammars will provide both a deeper understanding of such congruence relations and a more flexible computational framework in which to perform deduction.

2. EXPRESSIONS, EQUATIONS, AND GRAMMARS

The algorithms described here work just as well on second order terms as on first order terms. Thus expressions which are intended to denote functions or predicates are not handled any differently from expressions denoting domain elements. In fact the algorithms described here do not depend on any "typing" at all. This observation motivates the following definition of the set of all expressions over an alphabet *A*.

Definition: An *expression* over an alphabet *A* is either a symbol in *A* or an n-tuple $\langle u_1\ u_2 \ldots u_n \rangle$ of expressions over *A*.

This definition of an expression is similar to the definition of an s-expression in LISP. While all of the algorithms described here operate on expressions they work just as well when restricted to typed expressions or first order terms.

Consider a set of equalities Σ between expressions. If Σ was a set of equalities between first order terms then it would be clear what equalities follow from Σ. However since expressions are in some sense more general than first order terms an explicit definition of the set of equalities which are deducible from Σ is given below (it is based on the standard deductive properties of equality). There is a simple semantics (which will not be presented here) for expressions under which the following notion of "deducible" is both sound and complete.

Definition: The set of equalities *deducible* from a set of equalities Σ, is the smallest set which contains Σ and which satisfies is the following deductive principles:

Reflexivity: For any expression u, $u = u$ is deducible.

Symmetry: If $u = v$ is deducible from Σ then $v = u$ is deducible from Σ.

Transitivity: If both $v = w$ and $u = w$ are deducible from Σ then $u = v$ is deducible from Σ.

Substitutivity: If the equalities $u_1 = v_1, u_2 = v_2, \ldots u_n = v_n$ are all deducible from Σ then the equality $\langle u_1\ u_2 \ldots u_n \rangle = \langle v_1\ v_2 \ldots v_n \rangle$ is deducible from Σ.

The following definitions and lemma 2.1 provide some basic concepts relating sets of equalities to equivalence relations on expressions. While these notions may seem obvious and redundant they facilitate precision in later sections.

Definitions: Let $\approx$ be an equivalence relation on expressions. The relation $\approx$ will be said to *subsume* an equality $u = v$ just in case u is equivalent to v under $\approx$, i.e. just in case $u \approx v$. The relation $\approx$ will be said to subsume a set of equalities Σ just in case it subsumes every

equality in Σ. The relation $\approx$ is called a *congruence relation* just in case it is substitutive, i.e. whenever the equivalences $u_1 \approx v_1$, $u_2 \approx v_2$, ... $u_n \approx v_n$ hold, the equivalence $\langle u_1\ u_2 \ldots u_n\rangle \approx \langle v_1\ v_2 \ldots v_n\rangle$ also holds.

lemma 2.1: Congruence relations are deductively closed, i.e. if $\approx$ is a congruence relation and Σ is a set of equalities subsumed by $\approx$ then any equality deducible from Σ is also subsumed by $\approx$.

The following definition provides another characterization of the set of equalities deducible from a given set Σ.

Definition: The *congruence closure* of a set of equalities Σ is the equivalence relation $\approx_\Sigma$ on expressions which subsumes exactly those equalities which are deducible from Σ, i.e. $u \approx_\Sigma v$ just in case the equality $u = v$ is deducible from Σ. The equivalence class of an expression u under the congruence closure of Σ is denoted $|u|_\Sigma$. The congruence closure of a *finite* set Σ will be called a *finitely equational* congruence relation.

It is interesting to note that $|u|_\Sigma$ can be an infinite set even when Σ is finite. For example if Σ is $\{\langle f\ a\rangle = a\}$ then $|a|_\Sigma$ includes all expressions of the form $\langle f \langle f \ldots \langle f\ a\rangle \ldots\rangle\rangle$. The main result of this section will be that $|u|_\Sigma$ can be described by a context free expression grammar as defined below.

Definition: Let A be an alphabet of terminal symbols and let N be a collection of non-terminal symbols such that N is disjoint from A. A *context free expression grammar* over A and N is a set of productions of the form $X \Rightarrow \alpha$ where X is a non-terminal symbol and α is an expression over A union N.

Expression grammars are very much like string grammars. Each non-terminal symbol of an expression grammar generates a set of expressions over the terminal alphabet in the same way that each non-terminal of a context free string grammar generates a set of strings. For example the grammar consisting of the productions $A \Rightarrow a$ and $A \Rightarrow \langle f\ A\rangle$ describes the set of expressions of the form $\langle f \langle f \ldots \langle f\ a\rangle \ldots\rangle\rangle$.

Let A be a set of terminal symbols and N be a set of non-terminal symbols. Expressions over A (expressions containing only terminal symbols) will be called *terminal expressions* (or simply expressions) and will be denoted by the letters u, v, and w. Expressions over A union N will be called *expression forms* and will be denoted by the greek letters α, β, and γ. For a given grammar G the relation $\Rightarrow^*$ is defined on expression forms as the smallest relation satisfying the following three properties:

1) For any expression form α, $\alpha \Rightarrow^* \alpha$.

2) If $X \Rightarrow \alpha$ is a production of G then $X \Rightarrow^* \alpha$.

3) If $\alpha \Rightarrow^* \beta$ then for any expression form $\langle \gamma_1 \ldots \alpha \ldots \gamma_n \rangle$ containing α, $\langle \gamma_1 \ldots \alpha \ldots \gamma_n \rangle \Rightarrow^* \langle \gamma_1 \ldots \beta \ldots \gamma_n \rangle$

Intuitively $\alpha \Rightarrow^* \beta$ just in case β can be derived from α by replacing some number of non-terminal symbols in α by expressions which they generate. The language generated by an expression form α is defined to be the set of terminal expressions u such that $\alpha \Rightarrow^* u$.

The investigation of context free expression grammars is motivated by the existence of a relationship between a certain class of such grammars and finitely equational congruence relations on expressions. The following definition identifies the relevant class of grammars.

Definition: A context free expression grammar will be called *normal* if the following two conditions hold. First all productions are either of the form $X \Rightarrow a$ where a is a terminal symbol, or of the form $X \Rightarrow \langle Y_1\ Y_2 \ldots Y_n \rangle$ where each Y_i is a non-terminal symbol. Second there are no two productions $X \Rightarrow \alpha$ and $Y \Rightarrow \beta$ such that X and Y are distinct non-terminals but such that α and β are the same expression form.

It turns out that any normal grammar G represents a congruence relation on expressions. In general all expressions generated by a given expression form α will be equivalent under this relation. However not all expression forms generate entire equivalence classes. In particular if $\alpha \Rightarrow^* \beta$ then the language generated by β may be only a proper subset of the language generated by α. It turns out that the expression forms which generate entire equivalence classes are precisely the *maximal* expression forms where maximal is defined as below:

Definition: An expression form α will be called *maximal* under an expression grammar G just in case there is no expression form β other than α such that $\beta \Rightarrow^* \alpha$ under G.

Consider the equivalence class of the expression $\langle g\ a \rangle$ under the congruence closure of the equation $\langle f\ a \rangle = a$. The following normal grammar "represents" the congruence relation on expressions imposed by this equality (the first "production" below represents in the standard way the pair of productions $\Lambda \Rightarrow a$ and $\Lambda \Rightarrow \langle F\ \Lambda \rangle$).

$$\Lambda \Rightarrow a \mid \langle F\ \Lambda \rangle$$
$$F \Rightarrow f$$

The equivalence class of $\langle g\ a \rangle$ under the equality $\langle f\ a \rangle = a$ is exactly the language generated by the expression form $\langle g\ \Lambda \rangle$ under the above grammar. Note that the expression forms Λ and $\langle g\ \Lambda \rangle$ are maximal

under this grammar while the expression form $\langle f\ A \rangle$ is not.

For any function Γ on expressions let $\approx_\Gamma$ denote the equivalence relation such that $u \approx_\Gamma v$ just in case $\Gamma(u)$ equals $\Gamma(v)$. Furthermore for any equivalence relation $\approx$ on expressions and any expression v let $|v|_\approx$ denote the equivalence class of v under $\approx$. The following theorem establishes the first half of the relationship between normal expression grammars and finitely equational congruence relations.

Lemma 2.2: Let G be any normal context free expression grammar. For each expression u there exists a unique expression form $\Gamma(u)$ which is maximal under G and which generates u. Thus there exists a unique function Γ, called the maximal generator function of G, which maps each expression u to the maximal expression $\Gamma(u)$ which generates u. Furthermore $\approx_\Gamma$ is a finitely equational congruence relation and for any expression u the language generated by $\Gamma(u)$ is precisely $|u|_{\approx_\Gamma}$.

proof: The function Γ is defined recursively via the following conditions:

1) For any terminal symbol **a** if there is a production of the form $X \Rightarrow \mathbf{a}$ then $\Gamma(\mathbf{a})$ is X otherwise $\Gamma(\mathbf{a})$ is **a**.

2) If u is of the form $\langle u_1\ u_2\ \ldots\ u_n \rangle$ then if there is a production of the form $X \Rightarrow \langle \Gamma(u_1)\ \Gamma(u_2) \ldots \Gamma(u_n) \rangle$ then $\Gamma(u)$ equals X otherwise $\Gamma(u)$ equals $\langle \Gamma(u_1)\ \Gamma(u_2) \ldots \Gamma(u_n) \rangle$.

Since the normality of a grammar ensures that no two distinct productions have the same right hand side, the above definition is well formed. Note that $\Gamma(u)$ can be computed in linear time in the size of u (in general membership in a context free expression language can be determined in linear time although this will not be proven here).

A simple induction on expressions can be used to show that for any expression u, $\Gamma(u) \Rightarrow^* u$ (the details are left to the reader). It will now be shown by induction that for any expression u, $\Gamma(u)$ is maximal. To see this first note that in a normal grammar each non-terminal symbol is a maximal expression form (the only way a non-terminal symbol X could fail to be maximal is if there was some other non-terminal symbol Y such that $Y \Rightarrow X$ was a production of G, but this is not allowed in normal grammars). Now if **a** is a terminal symbol then either $\Gamma(\mathbf{a})$ is a non-terminal symbol (which is a maximal expression form) or $\Gamma(\mathbf{a})$ is **a**. But if $\Gamma(\mathbf{a})$ is **a** then by the definition of Γ there can be no production of G whose right hand side is **a** in which case **a** is a maximal expression form. Now let u be of the form $\langle u_1\ u_2 \ldots u_n \rangle$ and assume that for each u_i, $\Gamma(u_i)$ is maximal. If $\Gamma(u)$ is a non-terminal symbol then it is maximal, on the other hand if $\Gamma(u)$ is not a non-terminal symbol then it is an n-tuple of maximal expression forms which is not the right hand side of any production and is therefore maximal.

The proof that for any expression v there is at most one maximal expression form which generates v is done by induction on v. For a terminal symbol **a** if there is no production of the form

$X \Rightarrow a$ in the grammar then a is the unique maximal expression form which generates a. On the other hand if there is a production of the form $X \Rightarrow a$ then since the grammar is normal there is only one such production and the non-terminal symbol X is the unique maximal expression form which generates a. Now let v be an expression of the form $\langle v_1\ v_2 \ldots v_n \rangle$ such that for each v_i there is at most one maximal expression form which generates v_i. Since $\Gamma(v_i)$ is maximal and generates v_i it the *unique* such maximal expression form. Now there are two cases. The first case occurs when there is a non-terminal symbol X such that $X \Rightarrow^* v$. Since the grammar is normal it must contain a production of the form $X \Rightarrow \langle X_1\ X_2 \ldots X_n \rangle$ where each X_i is a non-terminal symbol such that $X_i \Rightarrow^* v_i$. However since each non-terminal symbol is maximal, X_i must equal $\Gamma(v_i)$ and therefore the production $X \Rightarrow \langle \Gamma(v_i)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ is in the grammar. This must hold for every non-terminal symbol X which generates v, i.e. for each X which generates v the above production must be in the grammar. However since there can be at most one production whose right hand side is $\langle \Gamma(v_i)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ there can be at most one non-terminal symbol which generates v. Note that in this case the expression form $\langle \Gamma(v_i)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ is not maximal. The second case occurs when there exists a maximal expression form α which generates v but which is not a non-terminal symbol, i.e. there exists an expression form α which generates v and is of the form $\langle \alpha_1\ \alpha_2 \ldots \alpha_n \rangle$. Since α is maximal each α_i must also be maximal and since α generates v, α_i must generate v_i for each i. However since there is at most one maximal expression generating v_i, α_i must be $\Gamma(v_i)$ so α must be $\langle \Gamma(v_i)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$. Note that in this case $\langle \Gamma(v_i)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ is maximal and therefore the first and second case can never occur at the same time. So either there is a unique non-terminal symbol which generates v or there is no such non-terminal and $\langle \Gamma(v_i)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ is the unique maximal expression which generates v.

The relation $\approx_\Gamma$ is substitutive, i.e. is a congruence relation. To see this let u be an expression of the form $\langle u_1\ u_2 \ldots u_n \rangle$ and let v be any expression of the form $\langle v_1\ v_2 \ldots v_n \rangle$. It suffices to show that if $\Gamma(u_i)$ equals $\Gamma(v_i)$ for each i, then $\Gamma(\langle u_1\ u_2 \ldots u_n \rangle)$ equals $\Gamma(\langle v_1\ v_2 \ldots v_n \rangle)$. However this follows directly from the definition of Γ.

To establish that the language generated by $\Gamma(u)$ is precisely $|u|_{\approx_\Gamma}$ first note that if v is in $|u|_{\approx_\Gamma}$, i.e. if $\Gamma(v)$ equals $\Gamma(u)$, then since $\Gamma(v) \Rightarrow^* v$ it must be the case that $\Gamma(u) \Rightarrow^* v$. On the other hand if $\Gamma(u) \Rightarrow^* v$ then since there is at most one maximal expression form which generates v, $\Gamma(u)$ equals $\Gamma(v)$ and therefore v is in $|u|_{\approx_\Gamma}$.

Finally it must be shown that the congruence relation defined by Γ is finitely equational, i.e. that there is a finite set of equalities Σ such that $\approx_\Gamma$ is the same as $\approx_\Sigma$. A non-terminal symbol X will be called *coherent* if the language generated by X is not empty. A production or expression form will be called coherent if it contains only coherent non-terminal symbols. Each coherent non-terminal symbol X can be associated with some expression u(X) such that $X \Rightarrow^* u(X)$. Now for each coherent expression form α, $u(\alpha)$ is defined as the result of replacing each non-terminal symbol X by u(X); clearly $\alpha \Rightarrow^* u(\alpha)$. Now let Σ be the set of equalities of the form $u(X) = u(\alpha)$

where $X \Rightarrow \alpha$ is a coherent production of G. Now if $u(X) = u(\alpha)$ is an equality in Σ then Λ generates both $u(X)$ and $u(\alpha)$ under G. Thus if $u(X) = u(\alpha)$ is an equality in Σ then $\Gamma(u(X))$ equals $\Gamma(u(\alpha))$ and therefore $u(X) \approx_\Gamma u(\alpha)$. Thus every equality in Σ is subsumed by $\approx_\Gamma$ and since $\approx_\Gamma$ is a congruence relation every equality deducible from Σ is therefore also subsumed by $\approx_\Gamma$. It remains only to show that every equality subsumed by $\approx_\Gamma$ is deducible from Σ. This is done by showing via a standard induction on expressions that for any expression v the equality $v = u(\Gamma(v))$ is deducible from Σ (the details are left to the reader). Thus if the equality $w = v$ is subsumed by $\approx_\Gamma$ then since $\Gamma(w)$ equals $\Gamma(v)$ and the equalities $v = u(\Gamma(v))$ and $w = u(\Gamma(w))$ are both deducible from Σ, the equality $w = v$ is deducible from Σ.

The following lemma completes the relationship between normal grammars and finitely equational congruence relations by showing that every finitely equational congruence relation can be represented by a normal grammar.

Lemma 2.3: For any finite set of equalities Σ there is a normal context free expression grammar G with maximal generator function Γ such that $\approx_\Gamma$ is the same as $\approx_\Sigma$. Furthermore G can be computed from Σ in order $n\log(n)$ average time where n is the total size of Σ.

proof: Let D be the finite set of expressions contained in Σ either directly or as a subexpression of some expression contained in Σ. Downey, Tarjan, and Sethi [5] show how to construct a function F on D such that for any two expressions u and v in D, F(u) equals F(v) just in case the equality $u = v$ follows from Σ. Furthermore a tabular representation of the function F can be computed in $n\log(n)$ average time where n is the total size of Σ. Such a function F can be translated into a normal context free expression grammar G in the following way: Each expression v in D is associated with a non-terminal symbol X(v) such that for any two expression u and v in D, X(u) equals X(v) just in case F(u) equals F(v). Now for each terminal symbol a in D we include the production $X(a) \Rightarrow a$ and for each expression v in D of the form $\langle v_1\ v_2 \ldots v_n \rangle$ we include the the production $X(v) \Rightarrow \langle X(v_1)\, X(v_2) \ldots X(v_n) \rangle$. The grammar G can be computed from F in time proportional to the total size of Σ.

To show that G is normal it is sufficient to show that no two distinct productions have the same right hand side. Let u and v be two expressions in D such that the production corresponding to u has the same right hand side as the production corresponding to v. It is sufficient to prove that X(u) must equal X(v) since then the production corresponding u would be the same as the production corresponding to v. Since the right hand sides of the productions corresponding to u and v are the same either u and v are both the same terminal symbol (in which case the result is trivial) or u is the form $\langle u_1\ u_2 \ldots u_n \rangle$, v was of the form $\langle v_1\ v_2 \ldots v_n \rangle$, and $X(u_i)$ equals $X(v_i)$ for each i. However in the latter case each equality $u_i = v_i$ would be deducible from Σ and therefore by substitutivity the

equality u=v would also be deducible from Σ and thus X(u) would equal X(v).

It remains to show that $\approx_\Gamma$ is the same as $\approx_\Sigma$ where Γ is the maximal generator function of G. First note that if u=v is an equality in Σ then u and v are in D and X(u) must equal X(v). Furthermore it is easily shown by induction that for any expression u, if u is in D then Γ(u) equals X(u). Thus if u=v is an equality in Σ then Γ(u) equals Γ(v) and thus $\approx_\Gamma$ subsumes Σ. Furthermore lemma 2.2 ensures that $\approx_\Gamma$ is a congruence relation and therefore subsumes any equality deducible from Σ. It remains only to show that any equality subsumed by $\approx_\Gamma$ is deducible from Σ. To show this first note that for each non-terminal symbol Y in G there is at least one expression w in D such that Y equals X(w). Thus each non-terminal symbol Y can be associated with an expression u(Y) in D such that X(u(Y)) equals Y. Now for any expression form α let u(α) be the result of replacing each non-terminal symbol Y in α by u(Y). It can now be shown by a standard induction that for any expression w the equality w=u(Γ(w)) is deducible from Σ (the details are left to the reader). Finally if an equality w=v is subsumed by $\approx_\Gamma$ then since Γ(w) equals Γ(v), and the two equalities w=u(Γ(w)) and v=u(Γ(v)) are both deducible from Σ the equality w=v can also be deduced from Σ.

A set of expressions L will be called a *context free expression language* if there is a context free expression grammar G and a non-terminal symbol X of G such that L is the language generated X under G. Now let G be any normal grammar and let Γ be the maximal generator function of G. Note that for any expression u, if Γ(u) is a non-terminal symbol of G then $|u|_{\approx_\Gamma}$ is a context free expression language. Furthermore if Γ(u) is not a non-terminal symbol then one can simply construct a new grammar G' by adding a new non-terminal symbol X and the production X⇒Γ(u) so that $|u|_{\approx_\Gamma}$ is the language generated by X under G'. In either case $|u|_{\approx_\Gamma}$ is a context free expression language. These observations and the above lemmas now yield the main result of this section:

Theorem 2.4 For any finite set of equations Σ and any expression u, $|u|_\Sigma$ is a context free expression language.

At this point it is useful to introduce another simple example. Consider the following three equations:

$$\langle f\langle f\langle f\, a\rangle\rangle\rangle = a$$
$$\langle f\, a\rangle = b$$
$$\langle f\, b\rangle = c$$

These equations correspond to the following normal grammar:

$$A \Rightarrow a \mid \langle F\ C\rangle$$
$$B \Rightarrow b \mid \langle F\ A\rangle$$
$$C \Rightarrow c \mid \langle F\ B\rangle$$
$$F \Rightarrow f$$

Under this grammar $\Gamma(\langle g\ \langle f\ b\rangle\rangle)$ equals $\langle g\ C\rangle$ and the equivalence class of $\langle g\ \langle f\ b\rangle\rangle$ equals the set of expressions generated by $\langle g\ C\rangle$. What expressions are there, if any, which are both equivalent to $\langle g\ \langle f\ b\rangle\rangle$ and contain only the symbols **g**, **f** and **a**? The answer is the set of expressions generated by the expression form $\langle g\ C\rangle$ under the following subset of productions:

$$A \Rightarrow a \mid \langle F\ C\rangle$$
$$B \Rightarrow \langle F\ A\rangle$$
$$C \Rightarrow \langle F\ B\rangle$$
$$F \Rightarrow f$$

In general consider a division of the terminal symbols into "dependent" and "independent" symbols. The following theorem provides a technique for solving for expressions in terms of independent symbols.

Theorem 2.5: Let G a normal grammar with maximal generator function Γ, and let G' be that subset of the productions of G which do not contain any dependent symbols. For any expression u the language generated by $\Gamma(u)$ under G' is precisely the set of expressions which are equivalent to u under $\approx_\Gamma$ and which do not contain any dependent symbols.

Proof: Any expression generated by $\Gamma(u)$ under G' contains no dependent symbols and is equivalent to u under $\approx_\Gamma$, i.e. is a solution for u. On the other hand any solution v for u must be equivalent to u and therefore must be generated by $\Gamma(u)$ under G. But in this case since v does not contain any dependent symbols the derivation of v from $\Gamma(u)$ must not involve any productions which contain dependent symbols. Thus v must be generated by $\Gamma(u)$ under G' as well.

3. BOOLEAN CONGRUENCE RELATIONS

Now that a good representation has been developed for equivalence classes under finitely equational congruence relations we turn our attention to Boolean congruence relations. An *equational Boolean formula* over an alphabet A is a formula which is built up from equalities between expressions over A using negations, disjunctions, conjunctions, and implications in the standard way. The *congruence closure* of an equational Boolean formula Q is defined to be the congruence relation $\approx_Q$ on expressions such that $u \approx_Q v$ just in case $u = v$ is deducible from Q using the standard deduction rules for Boolean connectives and the deduction rules for equalities described earlier. For an arbitrary expression u the equivalence class of u under the congruence closure of Q will be denoted by $|u|_Q$. It will be shown in this section that $|u|_Q$ is either the universal relation or a context free expression language.

Let P be an equational Boolean formula which is a conjunction $u_1 = v_1 \wedge u_2 \neq v_2 \wedge u_3 = v_3 \ldots$ of equalities and negations of equalities and let Σ be the set of equalities which appear in positive form in P. Using the techniques described in the previous section one can compute a normal expression grammar G with maximal generator function Γ such that $\approx_\Gamma$ is the same as $\approx_\Sigma$. Now the conjunction P is satisfiable just in case there is no negated equality $u_i \neq v_i$ appearing in P such that the equality $u_i = v_i$ is deducible from Σ, or equivalently such that $\Gamma(u_i)$ equals $\Gamma(v_i)$. If the conjunction is unsatisfiable then anything follows from P and thus the congruence closure of P is the universal relation. If the conjunction P is satisfiable then the congruence closure of P is just the congruence closure of Σ and for any expression u, $|u|_P$ equals $|u|_\Sigma$ which equals the language generated by $\Gamma(u)$ under G.

Any equational Boolean formula can be converted to disjunctive normal form, i.e. is equivalent to a disjunction of conjunctions of equalities and negations of equalities. Let Q any equational Boolean formula and let $P_1 \vee P_2 \vee \ldots \vee P_n$ be a disjunctive normal form of Q where each P_i is a conjunction of equalities and negations of equalities. An equality $u = v$ follows from Q just in case it follows from each P_i. In other words $\approx_Q$ is the intersection of the $\approx_{P_i}$'s. But by the above remarks the congruence closure of a given P_i is either the universal relation or the congruence closure of the positive equalities in that conjunction. These remarks lead to the following lemma:

> *Lemma 3.1:* The congruence closure of any equational Boolean formula is either the universal relation or a finite intersection of finitely equational congruence relations.

Given the above observations it is clear that one can decide whether an equality $u = v$ follows from a formula Q by first converting Q to disjunctive normal form and then seeing if $u = v$ follows from each disjunct. However it is not clear that a mechanism which can solve for u under a set of equalities can be extended to a mechanism for solving for u under an arbitrary equational Boolean formula. What is needed is a good characterization of the equivalence class of u under Q. As a simple example consider the following formula in which $\langle f^n a \rangle$ is an abbreviation for an expression of the form $\langle f \langle f \ldots \langle f a \rangle \ldots \rangle \rangle$ where the symbol f

appears n times.

$$\langle f^5\, a\rangle = a \vee \langle f^7\, a\rangle = a$$

The equivalence class of a under this formula is the set of expressions of the form $\langle f^n\, a\rangle$ where n is a multiple of 35. One approach to the problem of solving for expressions under a Boolean formula Q would be to try to find a finite set of equalities Σ such that $\approx_Q$ was the same as $\approx_\Sigma$. The following theorem demonstrates that this can not be done in general.

Theorem 3.2: The intersection of two finitely equational congruence relations need not be finitely equational.

Proof sketch: Consider the following pair of sets of equations:

Σ_1: $\{a = b\}$

Σ_2: $\{\langle f\, a\rangle = a, \langle f\, b\rangle = b, \langle g\, a\rangle = \langle g\, b\rangle\}$

The intersection of $\approx_{\Sigma_1}$ and $\approx_{\Sigma_2}$ is the same as $\approx_Q$ where Q is the Boolean formula:

$$a = b \vee (\langle f\, a\rangle = a \wedge \langle f\, b\rangle = b \wedge \langle g\, a\rangle = \langle g\, b\rangle)$$

The relation $\approx_Q$ is the same as the relation $\approx_\Pi$ where Π is the infinite set of equalities of the form:

$$\langle g\, \langle f^n\, a\rangle\rangle = \langle g\, \langle f^n\, b\rangle\rangle$$

Now $\approx_\Pi$ is finitely equational just in case there is a finite set of equalities Σ such that $\approx_\Pi$ is the same as $\approx_\Sigma$. If such a set Σ exists then if $u = v$ is an equality in Σ then $u \approx_\Pi v$ and thus the equality $u = v$ must be deducible from Π. But an equality is deducible from Π just in case it is deducible from some finite subset of Π and thus all of the equalities in Σ must be deducible from a single finite subset of Π. Thus $\approx_\Pi$ is finitely equational just in case it is the congruence closure of some finite subset of Π. However given the above definition of Π no finite subset of Π can imply all of the equalities in Π so $\approx_\Pi$ is not finitely equational.

The above theorem says that there is a Boolean formula Q such that $\approx_Q$ is not finitely equational. However it is still possible that for each expression u there is a finite set of equalities Σ such that $|u|_Q$ equals $|u|_\Sigma$ (remember that there are infinitely many such expressions u).

Let Q be an arbitrary equational Boolean formula. By Lemma 3.1 it is possible either to determine that the congruence closure of Q is the universal relation or to find a finite collection $\{\Sigma_1\ \Sigma_2 \ldots \Sigma_n\}$ of finite

sets of equations such that the congruence closure of Q is the intersection of the congruence closures of the Σ_i's. Thus for any expression u we have the following relation:

$$|u|_Q = |u|_{\Sigma_1} \cap |u|_{\Sigma_2} \cap ... |u|_{\Sigma_n}$$

Furthermore for each Σ_i there is a normal grammar G_i with an associated maximal generator function Γ_i such that $|u|_{\Sigma_i}$ is the language generated by $\Gamma_i(u)$ under G_i. This implies that $|u|_Q$ is the intersection of a finite collection of context free expression languages. While it is well known that the intersection of two context free string languages need not be a context free language there are fundamental differences between string languages and expression languages. Specifically expressions have a-priori phrase structure while strings do not. This difference is responsible for the fact that membership in a context free expression language can be determined in linear time (as was mentioned in the previous section) and it is also responsible for the following theorem:

Theorem 3.3: The intersection of two context free expression languages is a context free expression language.

Proof: An expression grammar will be called *one level* if every production is either of the form $X \Rightarrow a$ where a is a terminal symbol or of the form $X \Rightarrow \langle X_1 X_2 ... X_n \rangle$ where each X_i is a non-terminal symbol. (A normal grammar is a one level grammar in which no two productions have the same right hand side.) It is easy to show for any context free expression language L there is a one level grammar G such that L is the language generated by a non-terminal symbol of G. Given these remarks Theorem 3.3 follows directly from the following Lemma.

Lemma 3.4: For any two one level expression grammars G_1 and G_2 there exists a grammar denoted as $G_1 \cap G_2$ such that for each non-terminal X of G_1 and non-terminal Y of G_2 there exists a non-terminal of $G_1 \cap G_2$ denoted by $X \cap Y$ such that the language generated by $X \cap Y$ under $G_1 \cap G_2$ is the intersection of the languages generated by X and Y under G_1 and G_2 respectively.

Proof: Given G_1 and G_2 define $G_1 \cap G_2$ to be the one level grammar meeting the following conditions:

1) The set of non-terminal symbols of $G_1 \cap G_2$ is the cross product of the non-terminal symbols of G_1 and the non-terminal symbols of G_2. In other words for each non-terminal symbol X of G_1 and non-terminal Y of G_2 there exists a non-terminal symbol of $G_1 \cap G_2$ which will be denoted here by $X \cap Y$.

2) For each terminal symbol a a production $X \cap Y \Rightarrow a$ is in $G_1 \cap G_2$ just in case the production

$X \Rightarrow a$ is in G_1 and the production $Y \Rightarrow a$ is in G_2.

3) A production of the form $X \cap Y \Rightarrow \langle X_1 \cap Y_1\ X_2 \cap Y_2 \ldots X_n \cap Y_n \rangle$ is in $G_1 \cap G_2$ just in case the production $X \Rightarrow \langle X_1\ X_2 \ldots X_n \rangle$ is in G_1 and the production $Y \Rightarrow \langle Y_1\ Y_2 \ldots Y_n \rangle$ is in G_2.

Let u be any expression over terminal symbols. It will first be shown by induction on u that if $X \cap Y \Rightarrow^* u$ then $X \Rightarrow^* u$ and $Y \Rightarrow^* u$. If u is a terminal symbol then this result follows directly from the definition of $G_1 \cap G_2$. If u is of the form $\langle u_1\ u_2 \ldots u_n \rangle$ then if $X \cap Y \Rightarrow^* u$ then there must be a production of $G_1 \cap G_2$ of the form $X \cap Y \Rightarrow \langle X_1 \cap Y_1\ X_2 \cap Y_2 \ldots X_n \cap Y_n \rangle$ such that $X_i \cap Y_i \Rightarrow^* u_i$ for each i. By the definition of $G_1 \cap G_2$ this implies that the production $X \Rightarrow \langle X_1\ X_2 \ldots X_n \rangle$ is in G_1 and the production $Y \Rightarrow \langle Y_1\ Y_2 \ldots Y_n \rangle$ is in G_2. Furthermore by the induction hypothesis it must be the case that $X_i \Rightarrow^* u_i$ and $Y_i \Rightarrow^* u_i$ for each i. Therefore it must be the case that $X \Rightarrow^* u$ and $Y \Rightarrow^* u$ and the induction is complete.

Now it will be shown by induction on expressions that if $X \Rightarrow^* u$ and $Y \Rightarrow^* u$ then $X \cap Y \Rightarrow^* u$. Again if u is a terminal symbol then the result follows directly from the definition of $G_1 \cap G_2$. If u is of the form $\langle u_1\ u_2 \ldots u_n \rangle$ then if $X \Rightarrow^* u$ and $Y \Rightarrow^* u$ there must be a production in G_1 of the form $X \Rightarrow \langle X_1\ X_2 \ldots X_n \rangle$ and a production in G_2 of the form $Y \Rightarrow \langle Y_1\ Y_2 \ldots Y_n \rangle$ such that $X_i \Rightarrow^* u_i$ and $Y_i \Rightarrow^* u_i$ for each i. But by the definition of $G_1 \cap G_2$ this implies that the production $X \cap Y \Rightarrow \langle X_1 \cap Y_1\ X_2 \cap Y_2 \ldots X_n \cap Y_n \rangle$ is in $G_1 \cap G_2$. Furthermore by the induction hypothesis $X_i \cap Y_i \Rightarrow^* u_i$ for each i. Therefore $X \cap Y \Rightarrow^* u$ and the induction argument is complete.

Finally since $X \cap Y \Rightarrow^* u$ just in case $X \Rightarrow^* u$ and $Y \Rightarrow^* u$ the language generated by $X \cap Y$ under $G_1 \cap G_2$ is exactly the intersection of the language generated by X under G_1 with the language generated by Y under G_2.

Theorem 2.4, Lemma 3.1, and Theorem 3.3 now immediately imply the following corollary:

Corollary 3.5: For any consistent equational Boolean formula Q and any expression u, $|u|_Q$ is a context free expression language.

For any two grammars G_1 and G_2 the size of the grammar $G_1 \cap G_2$ is proportional to the product of the sizes of the grammars G_1 and G_2. However it seems reasonable to expect that for most pairs X, Y of non-terminal symbols of G_1 and G_2 respectively the language generated by X and the language generated by Y will be disjoint and therefore the language generated by $X \cap Y$ will be empty. A non-terminal symbol of a grammar will be called *coherent* if the language generated by that symbol is not empty. The *coherent fragment* of a grammar G is defined to be the grammar resulting from removing all productions which contain non-coherent non-terminal symbols. The coherent fragment of a grammar can be computed in time proportional to the size of that coherent fragment.

The fragment of $G_1 \cap G_2$ which is relevant to describing the language generated by $X \cap Y$ can be restricted even further. A non-terminal symbol Z will be said to be *accessible* from the non-terminal W if

there is a production whose left hand side is W and whose right hand side contains Z, or if there is a non-terminal V such that V is accessible from W and Z is accessible from V. For a given grammar G the fragment of G accessible from W is the subset of productions of G which contain only symbols accessible from W. The fragment of G accessible from W can be computed in time proportional to the size of that fragment. If one is interested in the language generated by $X \cap Y$ under $G_1 \cap G_2$ then one need only consider the fragment of $G_1 \cap G_2$ which is accessible from $X \cap Y$.

Of course the intersection of a large set of context free expression languages can be done by iteratively applying the intersection algorithm implicit in the proof of Lemma 3.4. In the worst case the size of the grammar can grow exponentially in the number of languages which are intersected. However it is expected that such exponential growth will not usually arise since at each step one can restrict the grammar to the coherent subset which is accessible from the non-terminal of interest.

The mechanisms that have been described so far can be extended to deal with conditional expressions. For a given alphabet *A* of terminal symbols one can define a *conditional expression* to be either a simple expression (a non-conditional expression) or an n-tuple $\langle \text{if } P\ u_1\ u_2 \rangle$ where P is an equational Boolean formula and u_1 and u_2 are conditional expressions. As was mentioned earlier it is straightforward to give a semantics for expressions and Boolean formulas. Such a semantics can be extended to a semantics of conditional expressions by defining the denotation of $\langle \text{if } P\ u_1\ u_2 \rangle$ to be the denotation of u_1 if P is true and to be the denotation of u_2 if P is false. Now for a conditional expression w and a Boolean formula Q, $|w|_Q$ is defined to be the set of non-conditional expressions which equal w in all interpretations which make Q true. This definition yields the following relation:

$$|\langle \text{if } P\ u_1\ u_2 \rangle|_Q = |u_1|_{P \wedge Q} \cap |u_2|_{(\neg P) \wedge Q}$$

Given the results of this section the above relation provides a way of computing a grammar representing $|w|_Q$ for any conditional expression w and consistent Boolean formula Q. Note that since elements of $|w|_Q$ must be non-conditional expressions $|w|_Q$ may be empty.

4. SIMPLIFICATION

A simplicity order on expressions is a partial order such that u is simpler than v just in case u is less than v under that order. For the purposes of this section let $\approx$ be an arbitrary congruence relation on expressions and for any expression u let $|u|_{\approx}$ denote the equivalence class of u under $\approx$. Note that since the simplicity order need not be total or well founded there need not be a unique simplest expression in $|u|_{\approx}$. The following definitions are important in discussing simplification under congruence relations.

> *Definitions:* A *simplicity bound* under $\approx$ is an expression w such that there is no expression in $|w|_{\approx}$ which is simpler than w. An expression w will be called *simplified* under $\approx$ just in case it and all of its subexpressions are simplicity bounds under $\approx$. A simplification of an expression u under $\approx$ is an expression in $|u|_{\approx}$ which is simplified under $\approx$.

The main result of this section is that for any "well behaved" simplicity order and any consistent equational Boolean formula Q the set of simplifications of an expression u under $\approx_Q$ is a context free expression language. However it is important to note that there are "pathological" simplicity orders for which this result does not hold. In particular let Σ denote the single equation $\langle f\ a\rangle = a$. Clearly $|a|_{\Sigma}$ is the set of all expressions of the form $\langle f^n\ a\rangle$. Now suppose that the simplicity order was such that larger expressions were always simpler than smaller expressions. Since the set $|a|_{\Sigma}$ contains arbitrarily large expressions it would not contain a simplicity bound. To eliminate this problem one might required that the simplicity order be well founded, i.e. that there does not exist any sequence of ever simpler expressions. However even under this restriction there are pathological orderings. Suppose for example that all expressions other than a are equally simple and but simpler than a. In this case any expression of the form $\langle f^n\ a\rangle$ (for n greater than 0) is a simplicity bound for $|a|_{\Sigma}$. However all expressions in $|a|_{\Sigma}$ contain a which is not a simplicity bound and therefore no expression in $|a|_{\Sigma}$ is a simplification of a. The following definition establishes a class of non-pathological or *well behaved* simplicity orders. Note that well behaved orders need not be well founded.

> *Definition:* A *well behaved* simplicity order on expressions is defined here as a partial order on expressions satisfying the following conditions:
>
> 1) *The order is pseudo-total*, which means that there is a function L from expressions to a totally ordered set such that u is simpler than v just in case L(u)<L(v). Such a function L will be called a *totalizer* for the simplicity order and L(u) will be called the *simplification level* of u. Note that not all partial orders are pseudo-total, in particular consider an order and three objects x, y, and z such that x is less than y but z is unordered with respect to both x and y. If such an order were pseudo-total then since z and x are unordered L(z) would have to equal L(x). Similarly L(z) would equal L(y). But then L(x) would have to equal L(y) which conflicts with the ordering between x and y.

2) *The order satisfies the following monotonicity condition:* A *context* is defined to be an expression with an "open slot" in it. For any expression w and any context C let C[w] be the result of replacing the open slot of C with w. For example if C is the context <f <g a> · b> then C[<g b>] is the expression <f <g a> <g b> b>. The monotonicity condition is that for any two expression u and v and any context C, if $L(u) \leq L(v)$ then $L(C[u]) \leq L(C[v])$ where L is any totalizer for the simplicity order.

3) *No expression is simpler than one of its subexpressions.*

Conditions on simplicity orders, such as the monotonicity condition above, can often be most readily expressed in terms of a totalizing function L. Such conditions can always be thought of as direct conditions on the simplicity order and therefore the truth of such conditions is independent of the choice of the totalizing function. In the remainder of this section L will always denote a totalizer for the simplicity order.

The monotonicity condition on well behaved simplicity orders warrants some investigation. First it is easily shown that the monotonicity condition implies that if L(u) equals L(v) then for any context C, L(C[u]) equals L(C[v]) (if L(u) equals L(v) then $L(C[u]) \leq L(C[v])$ and $L(C[v]) \leq L(C[u])$). Thus the level of an expression is determined by the levels of its subexpressions. This implies that any well behaved simplicity order can be characterized by a triple $\langle D, L_1, L_2 \rangle$ where D is a totally ordered set of "simplicity levels". L_1 is a function from symbols to D, and L_2 is a mapping from n-tuples of elements of D into D such that the following relations hold:

$$L(a) = L_1(a) \quad \text{for symbols a}$$

$$L(\langle u_1\, u_2 \ldots u_n \rangle) = L_2(\langle L(u_1)\, L(u_2) \ldots L(u_n) \rangle)$$

Note that the above relations could be taken as a definition of L in terms of L_1 and L_2. The condition that no expression is simpler than one of its subexpressions is equivalent to the following condition on L_2.

$$L_2(\langle L(u_1)\, L(u_2) \ldots L(u_n) \rangle) \geq \max(L(u_1)\, L(u_2) \ldots L(u_n))$$

Unfortunately the above conditions on well behaved simplicity orders are not quite strong enough to ensure that every expression has a simplification. There are now two ways of ensuring this. The first is to require that the simplicity order be well founded. The second is to require that the congruence relation be $\approx_Q$ for some consistent Boolean formula Q. The next theorem establishes that the well foundedness condition is sufficient for arbitrary congruence relations.

Theorem 4.1: For any well founded well behaved simplicity order and any congruence relation every expression has a simplification.

Proof: An expression w will be called "cleaner" than an expression u just in case the pair $\langle L(w)\ w\rangle$ is less than the pair $\langle L(u)\ u\rangle$ under a lexicographical ordering where the second components of the pairs are compared under the subexpression order. In other words w is cleaner than u just in case L(w) is less than L(u) or L(w) equals L(u) but w is proper a subexpression of u. Since well behaved simplicity orders are well founded the cleanliness ordering is also well founded, i.e. there are no infinite chains of ever cleaner expressions.

The proof that a simplification always exists will be done by induction on cleanliness. For an arbitrary expression u it is assumed that all expressions cleaner than u have simplifications. It is sufficient to show that this implies that u has a simplification.

If there is an expression w in $|u|_{\approx}$ which is simpler than u then w is cleaner than u and therefore has a simplification which must also be a simplification of u. On the other hand suppose that there is no expression in $|u|_{\approx}$ which is simpler than u (i.e. u is a simplicity bound). Because no expression can be simpler than one of its subexpressions if v is a proper subexpression of u then $L(v) \leq L(u)$ and therefore v must be cleaner than u. Thus by the induction hypothesis all subexpressions of u have simplifications. Let w be the result of replacing all of the top level subexpressions of u with simplifications of those expressions. By substitutivity w is in $|u|_{\approx}$ and all proper subexpressions of w are simplicity bounds. To show that w is a simplification of u it remains only to show that w is a simplicity bound for $|u|_{\approx}$. However since u is a simplicity bound and since the monotonicity condition implies that $L(w) \leq L(u)$, w must also be a simplicity bound.

For a congruence closure $\approx_Q$ of a consistent Boolean formulae Q the well foundedness condition can be removed from the above theorem. The first step in establishing this result is to show that for any consistent Boolean formula Q and expression u there is a grammar G which captures all of the information in Q relevant to simplifying u. The precise conditions placed on the grammar G is defined below in terms of u and $\approx_Q$.

Definition: Let G be any normal grammar with maximal generator function Γ. The grammar G will be said to *cover* an expression v just in case $\Gamma(v)$ is a non-terminal symbol. The grammar G will be said to *approximate* a congruence relation $\approx$ just in case $|v|_{\approx}$ equals the language generated by $\Gamma(v)$ for all expressions v which are covered by G.

lemma 4.2: For any consistent equational Boolean formula Q and any expression u there is a normal grammar G which approximates $\approx_Q$ and which covers u.

Proof: It was demonstrated in the previous section that if Q is consistent then there exists a finite set $\{\Sigma_1\ \Sigma_2 \ldots \Sigma_n\}$ of finite sets of equations such that the relation $\approx_Q$ equals the intersection of the relations $\approx_{\Sigma_i}$. For each Σ_i there exists a normal grammar G_i with maximal generator function Γ_i such that $\approx_{\Gamma_i}$ equals $\approx_{\Sigma_i}$. It can be assumed without loss of generality that each G_i covers u.

Now using the technique developed in the proof of lemma 3.4 one can construct the grammar $G_1 \cap G_2 ... \cap G_n$ such that the language generated by each non-terminal $X_1 \cap X_2 ... \cap X_n$ of $G_1 \cap G_2 ... \cap G_n$ is precisely the intersection of the languages generated by each X_i. From the definition of this grammar given in the proof of lemma 3.4 it is easy to show that this grammar is normal. To show that this grammar approximates $\approx_Q$ let v be any expression covered by $G_1 \cap G_2 ... \cap G_n$. The non-terminal of $G_1 \cap G_2 ... \cap G_n$ which generates v must be $\Gamma_1(v) \cap \Gamma_2(v) ... \cap \Gamma_n(v)$ and the language generated by this is precisely the intersection of the sets $|v|_{\Sigma_i}$ which is precisely $|v|_Q$. Finally since u is covered by each G_i the nonterminal $\Gamma_1(u) \cap \Gamma_2(u) ... \cap \Gamma_n(u)$ generates u and thus u is covered by $G_1 \cap G_2 ... \cap G_n$.

It is interesting to note that a normal grammar G can approximate a congruence relation $\approx$ even if $\approx$ is not finitely equational. In particular the relation $\approx_\Gamma$, where Γ is the maximal generator function of G, need not be the same as $\approx_Q$; if w is not covered by G then $|w|_{\approx_\Gamma}$ may be a proper subset of $|w|_\approx$.

The following lemma tightens the relationship between a congruence relation $\approx$ and a grammar G which approximates it.

lemma 4.3: If a normal grammar G approximates a congruence relation $\approx$ and u is an expression covered by G then the set of simplifications of u under $\approx$ equals the set of simplifications of u under $\approx_\Gamma$ where Γ is the maximal generator function of G.

Proof: First note that an expression covered by G is a simplicity bound under $\approx$ just in case it is a simplicity bound under $\approx_\Gamma$. This is because if v is an expression covered by G then $|v|_\approx$ equals $|v|_{\approx_\Gamma}$. Second note that if v is an expression covered by G then since G is normal (and therefore one level) each subexpression of v must be generated by some non-terminal symbol of G. Therefore each subexpression of an expression covered by G is covered by G. The above two observations imply that an expression covered by G is simplified under $\approx$ just in case it is simplified under $\approx_\Gamma$. Finally note that every expression in $|u|_\approx$ is covered by G since $|u|_\approx$ is precisely the set generated by the non-terminal symbol $\Gamma(u)$. Thus the set of simplifications of u under $\approx$ consists of those expressions in $|u|_{\approx_\Gamma}$ which are simplified under $\approx_\Gamma$, i.e. the set of simplifications of u under $\approx_\Gamma$.

The following theorem finally establishes the desired result concerning the existence of simplifications under consistent Boolean formulae.

Theorem 4.4: For any consistent Boolean formula Q, any well behaved simplicity order, and any expression u, u has a simplification under $\approx_Q$.

Given lemmas 4.2 and 4.3 the above theorem follows from the following lemma:

Lemma 4.5: Let G be any normal grammar with maximal generator function Γ. For any well behaved simplicity order every expression covered by G has a simplification under $\approx_\Gamma$.

The above lemma will be shown by constructing an algorithm which takes a normal grammar and assigns each non-terminal symbol X an expression s(X) which is simplified under $\approx_\Gamma$ and which is generated by X. Then for any expression u covered by G, s(Γ(u)) is a simplification of u.

The procedure for constructing the assignment s is presented below. At each point in the computation s is defined on a subset of non-terminal symbols. Such a partial assignment s can be extended to expression forms of the form $\langle \alpha_1\ \alpha_2 \ldots \alpha_n \rangle$ by setting $s(\langle \alpha_1\ \alpha_2 \ldots \alpha_n \rangle)$ equal to $\langle s(\alpha_1)\, s(\alpha_2) \ldots s(\alpha_n) \rangle$ whenever $s(\alpha_i)$ is defined for each α_i.

At each point in the computation there is also a queue of pairs <X, u> where X is a non-terminal symbol and u is an expression over terminal symbols. Each pair <X, u> on this queue should be thought of as a constraint on the value of s(X) which states that L(s(X)) must be less than or equal to L(u). This queue is always sorted with respect to the terminal expressions such that for any two pairs <X, u> and <Y, v> on the queue if u is simpler than v then the pair <X, u> appears earlier on the queue than the pair <Y, v>. Thus if <X, u> is the pair at the head of the queue then u must be at least as simple as any other terminal expression appearing in any other pair on the queue. The procedure maintains the following three invariants:

1) For each non-terminal X such that s(X) is defined, X generates s(X) under G.

2) For each pair <X, u> on the queue X generates u under G.

3) Let <X, u> be the first pair on the queue. For any non-terminal symbol Y if s(Y) is defined then $L(s(Y)) \leq L(u)$.

The procedure is given below, initially s is totally undefined.

1) For each production of the form $X \Rightarrow a$ where a is a terminal symbol place the pair <X, a> on the queue.

2) Take the first pair <X, u> off the queue. If s(X) is undefined do the following:

i) Set s(X) to u.

ii) For each production of the form $Y \Rightarrow \alpha$ such that X appears in α and $s(\alpha)$ is defined, place the pair $\langle Y, s(\alpha) \rangle$ on the queue.

3) If the queue is not empty go to 2)

The algorithm terminates because there are only finitely many non-terminal symbols which can be

assigned simplifications. The first two invariants are that if s(X) is defined then X generates s(X) and if <X, u> is a pair on the queue then X generates U. These two invariants are easily shown to be true in the initial state since the function s in initially totally undefined and every pair on the queue is of the form <X, a> where a is a terminal symbol and X⇒a is a production of G. Given that these invariants hold it can easily be shown that no step of the procedure causes either invariant to be violated.

The final invariant is that for any non-terminal Y, if s(Y) is defined and <X, u> is the first pair on the queue then $I_\Gamma(s(Y)) \leq I_\Gamma(u)$. Because the queue is sorted if <X, u> is the first pair on the queue then $I_\Gamma(u)$ is the least simplicity level assigned any terminal expression on the queue. Thus the final invariant is equivalent to the statement that whenever s(Y) is defined, s(Y) is at least as simple as any terminal expression on the queue. This invariant holds trivially in the initial condition since the function s is totally undefined. Furthermore simply removing a pair from the queue can never cause this invariant to be violated. The only times the invariant might be violated is when s(X) is defined for some X or when a pair <Z, w> is added to the queue.

Consider an execution of step 2) of the procedure in which a pair <X, u> is removed from the queue and s(X) gets set to u. Let <Y, v> be the pair which is next on the queue when the pair <X, u> is removed. Given that the invariant is already in force $I_\Gamma(s(Z))$ must be less than or equal to $I_\Gamma(u)$ for any non-terminal symbol Z such that s(Z) was defined. Furthermore $I_\Gamma(u)$ is less than or equal to $I_\Gamma(v)$ and thus $I_\Gamma(s(Z))$ is less than or equal to $I_\Gamma(v)$ for any non-terminal symbol Z such that s(Z) was defined. Further since $I_\Gamma(u)$ is less than or equal to $I_\Gamma(v)$, $I_\Gamma(s(X))$ will be less than or equal to $I_\Gamma(v)$ when s(X) is set to u. Thus the removal of <X, u> from the queue and the setting of s(X) to u does not violate the invariant.

Now consider the queueing of any pair of the form <Z, s(α)>. Let <Y, v> be the first pair on the queue before the pair <Z, s(α)> is added. For every non-terminal symbol W such that s(W) is defined $I_\Gamma(s(W)) \leq I_\Gamma(v)$. Given this fact the monotonicity condition on the simplicity order can be used to show (via a simple induction on expression forms) that for any expression form α such that s(α) is defined $I_\Gamma(s(\alpha)) \leq I_\Gamma(v)$. Thus when the pair <Z, s(α)> is queued $I_\Gamma(s(\alpha))$ is less than or equal to $I_\Gamma(v)$ so the simplicity level of the first pair on the queue remains the same and the invariant is not violated.

It will now be shown that if X is any coherent non-terminal (i.e. the language generated by X is not empty) then s(X) gets defined by the above procedure and that s(X) is a simplicity bound under $\approx_\Gamma$. It is sufficient to show by induction on expressions that if v is any expression covered by G then s(Γ(v)) gets defined and $I_\Gamma(s(\Gamma(v))) \leq I_\Gamma(v)$. If v is a terminal symbol covered by G then the pair <Γ(v), v> will be placed on the queue in step 1) of the procedure. This ensures that s(Γ(v)) will become defined and that $I_\Gamma(s(\Gamma(v))) \leq I_\Gamma(v)$. Now suppose v is an expression covered by G which is of the form $\langle v_1\ v_2 \ldots v_n\rangle$. Since v is covered by G each v_i must also be covered by G and the production $\Gamma(v) \Rightarrow \langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n)\rangle$ must be a production of G. Since each v_i is covered by G the induction hypothesis implies that $s(\Gamma(v_i))$ gets defined and that $I_\Gamma(s(\Gamma(v_i))) \leq I_\Gamma(v_i)$. Thus the monotonicity condition on the simplicity order implies that:

$$I_\Gamma(\langle s(\Gamma(v_1))\ s(\Gamma(v_2)) \ldots s(\Gamma(v_n))\rangle) \leq I_\Gamma(\langle v_1\ v_2 \ldots v_n\rangle).$$

Now since the production $\Gamma(v) \Rightarrow \langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n)\rangle$ is in G and since $s(\Gamma(v_i))$ gets defined for

each v_i the pair $\langle\Gamma(v)\langle s(\Gamma(v_1))\, s(\Gamma(v_2)) \ldots s(\Gamma(v_n))\rangle\rangle$ gets placed on the queue. Therefore $s(\Gamma(v))$ gets defined such that:

$$L(s(\Gamma(v))) \leq L(\langle s(\Gamma(v_1))\, s(\Gamma(v_2)) \ldots s(\Gamma(v_n))\rangle)$$

Combining the above two relations we get:

$$L(s(\Gamma(v)) \leq L(v)$$

It has been shown that for each coherent non-terminal symbol X that s(X) gets defined and that s(X) is a simplicity bound under $\approx_\Gamma$. Note that given the way pairs are placed on the queue and the way s(X) gets assigned, if s(X) is not a terminal symbol then it can be written as $\langle s(Y_1)\, s(Y_2) \ldots s(Y_n)\rangle$ where each Y_i is a non-terminal symbol. Thus every subexpression of s(X) is also a simplicity bound and therefore s(X) is simplified under $\approx_\Gamma$. This implies that for any expression v covered by G, $s(\Gamma(v))$ is a simplification of v.

In analyzing the running time of this algorithm let |G| be the total size of the grammar G. For each pair placed on the queue there is a particular production $X \Rightarrow \alpha$ of G such that u is $s(\alpha)$. A given production can be responsible for at most the queueing of one pair so that the number of pairs placed on the queue is order |G|. The time required to queue and remove order |G| elements from a sorted queue (a priority queue) is order |G|log(|G|) plus the time it takes to compute order |G|log(|G|) comparisons. A given production $X \Rightarrow \langle X_1\, X_2 \ldots X_n\rangle$ can be examined by part ii) of step 2) of the algorithm at most n times. Thus the total time spent in part ii) of step 2 (ignoring queueing time) is order |G|. Thus the total time taken by the above procedure is order |G|log(|G|) plus the time it takes to perform order |G|log(|G|) comparisons between expressions.

The next theorem provides a representation for the set of simplifications of an expression under the congruence relation imposed by a normal grammar G. It turns out that the set of simplifications can be described by a grammar which is a subset of the productions of G.

Lemma 4.6: For any well behaved simplicity order and normal grammar G with maximal generator function Γ there is a subset G' of the productions of G such that for any expression w the set of simplifications of w under $\approx_\Gamma$ is the language generated by $\Gamma(w)$ under G'.

Proof: Without loss of generality it may be assumed that every non-terminal of G is coherent. Thus for each non-terminal symbol X of G, L(X) can be defined to be the simplification level of any simplicity bound generated by X, i.e. L(X) is the minimum simplicity level assigned any expression generated by X. The totalizer L can be represented as a triple $\langle D\, L_1\, L_2\rangle$ as described above. Given such a representation $L(\alpha)$ can be defined for an arbitrary expression form α via the

following relation:

$$L(\langle \alpha_1\ \alpha_2 \ldots \alpha_n \rangle) = L_2(\langle L(\alpha_1)\ L(\alpha_2) \ldots L(\alpha_n) \rangle)$$

The monotonicity condition on the simplicity order ensures that in general $L(\alpha)$ is the minimum level assigned any expression generated by an expression form α.

Let G' be that subset of the productions $X \Rightarrow \alpha$ of G such that $L(\alpha)$ equals $L(X)$. Now it will be shown by induction on expressions that for any expression v covered by G, if $\Gamma(v)$ generates v under G' then $L(v)$ equals $L(\Gamma(v))$. For any expression v covered by G, $\Gamma(v)$ equals some non-terminal symbol X of G. If v is a constant symbol covered by G then if X generates v then the production $X \Rightarrow v$ is in G' and by the definition of G' this implies $L(v)$ equals $L(X)$. If v is of the form $\langle v_1\ v_2 \ldots v_n \rangle$ and X generates v under G' then there must be a production $X \Rightarrow \langle X_1\ X_2 \ldots X_n \rangle$ *of G' such that* $X_i \Rightarrow^* v_i$ *for each i and thus by the definition of G' the following relations must* hold:

$$\begin{aligned} L(X) &= L(\langle X_1\ X_2 \ldots X_n \rangle) \\ &= L_2(\langle L(X_1)\ L(X_2) \ldots L(X_n) \rangle) \end{aligned}$$

and by the induction hypothesis:

$$\begin{aligned} &= L_2(\langle L(v_1)\ L(v_2) \ldots L(v_n) \rangle) \\ &= L(v) \end{aligned}$$

It will now be shown that for any expression form α, if $\alpha \Rightarrow^* v$ under G' then $L(v)$ equals $L(\alpha)$. This is done by induction on expression forms. If α is a terminal symbol then if $\alpha \Rightarrow^* v$ under G' then α must equal v and the result is trivial. If α is a non-terminal symbol and $\alpha \Rightarrow^* v$ under G' then $\Gamma(v)$ must equal α and the result follows from the above induction. If α is of the form $\langle \alpha_1\ \alpha_2 \ldots \alpha_n \rangle$ then v must of the form $\langle v_1\ v_2 \ldots v_n \rangle$ where $\alpha_i \Rightarrow^* v_i$ for each α_i. In this case the result follows from the following relations:

$$\begin{aligned} L(\alpha) &= L(\langle \alpha_1\ \alpha_2 \ldots \alpha_n \rangle) \\ &= L_2(\langle L(\alpha_1)\ L(\alpha_2) \ldots L(\alpha_n) \rangle) \end{aligned}$$

and by the induction hypothesis:

$$\begin{aligned} &= L_2(\langle L(v_1)\ L(v_2) \ldots L(v_n) \rangle) \\ &= L(v) \end{aligned}$$

It follows from the above induction that if $\Gamma(v) \Rightarrow^* v$ under G' then $L(v)$ is the least level assigned any expression generated by $\Gamma(v)$ under G and therefore v is a simplicity bound under $\approx_\Gamma$. It will now be show that if $\Gamma(v) \Rightarrow^* v$ under G' then v is simplified under $\approx_\Gamma$. Form the above observations it is sufficient to show that if $\Gamma(v) \Rightarrow^* v$ under G' and if v is of the form $\langle v_1\ v_2 \ldots v_n \rangle$ then for each v_i, $\Gamma(v_i) \Rightarrow^* v_i$ under G'. The expression form $\Gamma(v)$ is either a non-terminal symbol or the n-tuple $\langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$. In the latter case $\Gamma(v_i) \Rightarrow^* v_i$ for each v_i. In the former case since G' is a one level grammar each v_i must be generated under G' by some non-terminal symbol of G, but the only non-terminal which could possibly generates v_i under G' is $\Gamma(v_i)$ and therefore $\Gamma(v_i) \Rightarrow^* v_i$ for each v_i.

For any maximal expression form α the language generated by α under G is an equivalence class of $\approx_\Gamma$. Furthermore it has been shown that if $\alpha \Rightarrow^* v$ under G' then v is simplified under $\approx_\Gamma$. Thus for any expression u the language generated by $\Gamma(u)$ under G' is a subset of the simplifications of u. It now remains only to show that every simplification of u is generated by $\Gamma(u)$ under G'.

It will be shown by induction on expressions that if v is an expression which is simplified under $\approx_\Gamma$ then $\Gamma(v) \Rightarrow^* v$ under G'. If v is a constant symbol which is simplified under $\approx_\Gamma$ then either $\Gamma(v)$ is v in which case the result is trivial or $\Gamma(v)$ is a non-terminal symbol such that the production $\Gamma(v) \Rightarrow v$ is in G. But since v is simplified under $\approx_\Gamma$ $L(v)$ must equal $L(\Gamma(v))$ and thus the production $\Gamma(v) \Rightarrow v$ is in G'. Now let v be a simplified expression of the form $\langle v_1\ v_2 \ldots v_n \rangle$. since v is simplified so is each v_i and therefore by the induction hypothesis for each v_i, $\Gamma(v_i) \Rightarrow^* v_i$ under G'. Furthermore $\Gamma(v)$ is either $\langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ or a non-terminal symbol X such that the production $X \Rightarrow \langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ is in G. Thus it is sufficient to prove that in the case where $\Gamma(v)$ is a non-terminal symbol X the production $X \Rightarrow \langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle$ is in G', or equivalently to prove that $L(\langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle)$ equals $L(X)$. However since v is simplified $L(X)$ equals $L(v)$ and furthermore since each v_i is simplified $L(\Gamma(v_i))$ equals $L(v_i)$. This implies the following relations which establish the desired result:

$$
\begin{aligned}
& L(\langle \Gamma(v_1)\ \Gamma(v_2) \ldots \Gamma(v_n) \rangle) \\
& \quad = L_2(\langle L(\Gamma(v_1))\ L(\Gamma(v_2)) \ldots L(\Gamma(v_n)) \rangle) \\
& \quad = L_2(\langle L(v_1)\ L(v_2) \ldots L(v_n) \rangle) \\
& \quad = L(v) \\
& \quad = L(X)
\end{aligned}
$$

It is important to note that the subset G' of the productions of G can be computed in a

straightforward way given an assignment s of simplified expressions to non-terminal symbols such as the one produced in the procedure described earlier. Specifically a production $X \Rightarrow \alpha$ is in G' just in case s(X) is not simpler than $s(\alpha)$.

The following theorem has now been established:

Theorem 4.7: For any consistent equational Boolean formula Q, any expression u, and any well behaved simplification order the set of simplifications of u under $\approx_Q$ is a context free expression language.

5. CONCLUSIONS AND OPEN PROBLEMS

The notion of an expression is more general than the notion of first order term. Many different logical formalisms consist at least in part of expressions in which the substitution of equals for equals is valid. Thus the concepts and algorithms discussed here can be used for valid inference in a wide variety of deductive systems.

The relationship which has been established between finite sets of equations and context free expression grammars may have applications beyond the deduction techniques and simplification algorithms discussed here. For example consider unification under a finitely equational congruence relation. More precisely let u and v be two expressions containing variables and let Σ be a finite set of equations between ground expressions. For any substitution σ and expression u let $\sigma(u)$ be the result of simultaneously replacing each variable of u with its image under σ. The unification problem for the finite set of equations Σ is to characterize the set of ground substitutions σ such that $\sigma(u) \approx_{\Sigma} \sigma(v)$. An obvious first step in approaching this problem is to consider a grammar G which represents Σ and consider substitutions over the maximal expression forms of G (remember that each maximal expression form of G represents an equivalence class under Σ). The details of a unification algorithm have not been worked out and it is not clear what the complexity of this problem is. Another extension would be to develop a unification algorithm for equational Boolean formulas (remember that for a Boolean formula Q, $\approx_Q$ need not be finitely equational).

It is hoped that the material presented here will be applicable to various domains including program verification, compiler optimization techniques, and deductive data bases.

6. REFERENCES

1. Weyhrauch, R. W., Prolegomena to a Theory of Mechanized Formal Reasoning. Artificial Intelligence 13 (1,2) (April 1980) 81-133.

2. Moore. R. C. Reasoning about Knowledge and Action. MIT Ph.D. Dissertation (February 1979).

3. Sussman, G.J., Steele, G. L. Jr. CONSTRAINTS A Language for Expressing Almost-Hierarchical Descriptions. Artificial Intelligence 14 (1) (1980). 1-39

4. McAllester, D. A., The Use of Equality in Deduction and Knowledge Representation. MIT AI Lab Technical Report 520, February 1980.

5. Downey, P. J., Sethi, R, Tarjan, R. E. Variations on the Common Subexpression Problem. J. ACM 27, 4 (Oct. 1980) 758-771.

6. Nelson, G., Oppen, D. C. Fast Decision Procedures based on Congruence Closures. J. ACM 27, 2 (April 1980), 356-364.

7. Oppen, D. C. Reasoning about Recursively Defined Data Structures. J. ACM 27, 2 (April 1980), 356-364.

END

FILMED

10-83

DTIC