FORMAL GRAMMARS AS MODELS OF LOGIC DERIVATIONS

Sharon Sickel
Information Sciences
University of California
Santa Cruz CA 95064

Keywords and phrases: Automatic theorem proving, clause interconnectivity graphs, context-free grammars, attribute grammars, proof theory.

This work supported in part by the Office of Naval Research under contract 76-C-0681.

Abstract

Context-free attribute grammars are proposed as derivational models for proofs in the predicate calculus. The new representation is developed and its correspondence to resolution-based clause interconnectivity graphs is established. The new representation may be used to transform a predicate calculus characterization of a problem into a regular algebra characterization of the solutions.

The new representation can be used to simplify the search for proofs. It allows us to express and derive predicate calculus proofs as a constraining function that serves as a filter to the set of candidate proofs that ignore the arguments to predicates. The effect of this is to separate the underlying propositional structure from the restrictions imposed by the required unifications.

While previous theorem proving methods have been able to enumerate all proofs of a theorem, the method reported here is unique in being able to characterize all proofs of some theorems, representing even an infinite set of proofs with a finite formula. This work has implications for proof theory as well as providing a useful tool in the analysis of programs specified in logic.

1. Introduction

This section gives definitions of clause interconnectivity graphs and context-free grammars. The definitions have been extended where needed to express the additional structure treated in this paper. A more detailed description of clause interconnectivity graphs is given by Sickel[4]. Common definitions in theorem proving used here are given by Chang and Lee[l],

A substitution 0 is a set of ordered pairs [tl/xl, t2/x2,...tn/xn] where each ti is an arbitrary term and each xi is a distinct variable. For an arbitrary literal L, Lo denotes the literal L with all occurrences of xi replaced by ti for l^isn. Similiar definitions apply for terms to, and clauses C.

A directed substitution is given by $e_{s1\ s2}$ where 0 is a substitution [tl/xl,...,tn/xn], and si and s2 form a partitioning of the variables of 0. For example, $[f(y)/x,\ 9(^{a})/^{z}J_{(x)(y,z)}$ is a directed substitution.

A variant $o_{i,j}$ of a directed substitution $\Theta_{s1\ s2}$ is a substitution derived from Θ in which each variable $y \in s1$ is replaced by y_i in Θ, and each variable $z \in s2$ is replaced by z_j in Θ. Variant $\alpha_{i,-}$ replaces only variables from sl; variant $\alpha_{-,j}$ replaces only variables from s2. $\alpha_{-,-} = \Theta$.

Given a set of clauses that are variable disjoint, we can construct a clause interconnectivity graph (CIG). A CIG is a quadruple:
<Nodes, Edges, Subst, Clause> where

Nodes is a set of graph nodes, one for each literal in each clause. Even if two literals in separate clauses are identical, they correspond to different nodes.

Edges is a symmetric relation between pairs of nodes such that <A,B> $\in$ Edges iff the literals associated with nodes A and B have opposite signs and unifiable atoms. We write A<>B if <A,B> $\in$ Edges and A<+>B if either A<>B or if $\exists$C,D such that A<>C, C $\in$ Clause(D) but C$\neq$D, and D<+>B.

Subst is a mapping: Edges → Directed Substitutions such that $Subst(<A,B>)=\Theta_{s1\ s2}$ where Θ is a most general unifier of the atoms of the literals associated with nodes A and B, sl is the set of variables appearing in both A and Θ, and s2 is the set of variables appearing in both B and Θ. In the ground case, Subst maps to the empty substitution.

Clause is a mapping: Nodes → Powerset(Nodes). Clause partitions the nodes so that literals in the same clause have corresponding nodes in the same partition.

The start clauses are one or more clause partitions which may be chosen arbitrarily. Usually they are the clauses representing the negation of the theorem to be proved.

Residual literals is a mapping: Nodes → Powerset(Nodes) where Residual_literals(B) = Clause(B) - {B}.

For example, Figure 1 shows a simple CIG. Throughout the discussion of this example we refer to nodes by the literals they represent. If two or more literals were identical, it would be necessary to distinguish between them. For this example

Nodes = the set of circled literals

Edges = {<A,B>| A and B are connected by an edge of the graph}. I.e., {<A(x), $\overline{A}$(y)>, <$\overline{A}$(y), A(x)>, etc.}

A(x)<+>C(g(u)) since A(x)<>$\overline{A}$(y), $\overline{A}$(y) $\in$ Clause($\overline{C}$(g(y))) and $\overline{C}$(g(y))<>C(g(u)). Intuitively P<+>Q means that from P we can get to Q by first traversing an edge and then alternating between selecting a residual of the destination and from there traversing an

edge, etc.

Subst maps each edge to a directed substitution, e.g., Subst(<A(x), $\overline{A}(y)$>)=$[x/y]_{(x)(y)}$, Subst(<$\overline{A}(y)$, A(x)>)=$[x/y]_{(y)(x)}$, etc.

Undirected substitutions are shown on the undirected edges of the figure.

Clause maps each node to the set of nodes in the same clause, e.g., Clause($\overline{A}(f(z))$) = {$\overline{A}(f(z))$, C(g(u)), $\overline{D}(w)$}.

Residual_literals maps a node to the other nodes in the same clause, e.g., Residual_literals($\overline{A}(f(z))$) = {C(g(u)), $\overline{D}(w)$}.

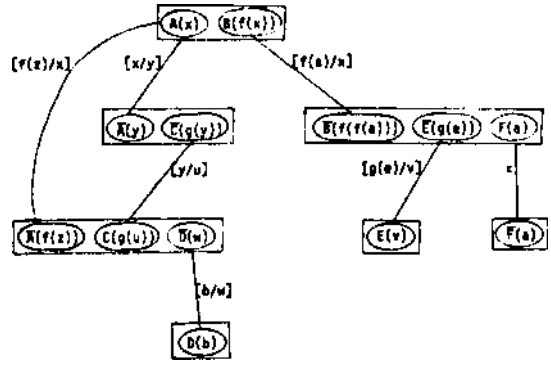


Figure 1: A CIG for clauses {A(x) B(f(x)), $\overline{A}(y)$ $\overline{C}(g(y))$, D(b), $\overline{A}(f(z))$ C(g(u)) $\overline{D}(w)$, $\overline{B}(f(f(a)))$ $\overline{E}(g(e))$ F(a), E(v), $\overline{F}(a)$}.

Continuing with the definitions,

Unifying composition ($\odot$) is a mapping: Substitutions×Substitutions → Substitutions where $\alpha\odot\beta = \gamma$ such that γ is a most general unifier satisfying, for an arbitrary term t,

$$(t\alpha)\gamma = (t\gamma)\alpha = t\gamma = (t\beta)\gamma = (t\gamma)\beta.$$

If no such γ exists, $\alpha\odot\beta$ is undefined. $\odot$ is commutative and associative, i.e.,

$\alpha\odot\beta = \beta\odot\alpha$

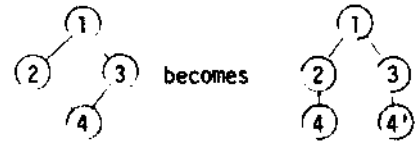
$\alpha\odot(\beta\odot\gamma) = (\alpha\odot\beta)\odot\gamma$

Composition of substitutions is normally defined in the context of applying substitutions sequentially. However, for this application we are looking for evidence that substitutions are compatible. Here, substitutions may be applied in any order; therefore they must be commutative and associative.

Suppose we have a sequence of edges ei=<Ai,Bi> $1\le i\le n$, such that A1 = Bn and for $1\le i<n$, A(i+1) $\in$ Residual_literals(Bi), and for $i\neq j$ Ai $\notin$ Clause(Aj). Then the sequence leads from node A1 back to itself (i.e., A1<+>A1) without visiting any other clause more than once. The base node is A1. If the substitutions along the way allow us to use the same instance of A1 at both ends of the sequence (i.e., if α = Subst(e1)$\odot$Subst(e2)$\odot$...$\odot$Subst(en) is defined) then m is called a merge loop. Sub(m), the substitution of the loop, is the directed substitution $\alpha_{s1,s2}$ such that s1 is the list of variables from the clause of base node A1 and s2 is the list of all other variables appearing in the loop.

For a merge loop <A1,B1>, <A2,B2>, ...<An,Bn> the undeleted residuals = {L| L $\in$ Residual_literals(Bk) -{A(k+1)} where $1\le k\le n-1$}.

In order to find proofs, we must find ways to delete all literals in a start clause. If a CIG is a tree to begin with, then it models a proof in which the root is the start clause. However, if the CIG is not a tree, then it must be made into one. For example,



The definitions for deletion tree and solution tree do that transformation, along with checking for substitution consistency.

A deletion tree for node N, denoted T(N) is:

a) a finite tree
- (i) having root N
- (ii) where N has a single child L such that <N,L> $\in$ Edges
- (iii) and L has k children T(L1), ... T(Lk) where Residual_literals(L) = {L1,...,Lk}
- (iv) and Sub(T(N)) = $Subst(<N,L>)_{-,j}\odot Sub_j(T(L1))\odot...\odot Sub_j(T(Lk))$ is defined where j is an integer not used before†, and where $Sub_j(T(N))$ = Sub(T(N)) with all previously unsubscripted variables now subscripted with i. The subscripts are used to distinguish between different occurrences of the variables.

Note: if k=0, then L is a leaf.

b) a finite tree
- (i) having root N
- (ii) where N has a single child m such that m is a merge loop with base node N
- (iii) and m has k children T(L1), ... T(Lk) where {L1,...Lk} are the undeleted residuals of m
- (iv) and $Sub(T(N)) = Sub(m)_{-,j}\odot Sub_j(T(L1))\odot...\odot Sub_j(T(Lk))$ is defined where j is an integer that has not been used before and where $Sub_j(T(N))$ is as in case a.

Note: if k=0, then m is a leaf.

---

† The reason for this condition is to guarantee that subtrees T(L1)...T(Lk) do not have common renaming constants. The choice of j is by a global function similar to GENSYM in LISP.

c) derived only by means of a) and b).

A solution tree, Ts(C), for a CIG C is a tree such that:

(i) it has root "."

(ii) and "." has as children deletion trees for all the literals of a start clause {L1, ... , Lk}.

(iii) and $Sub(Ts(C)) = Sub_j(T(L1)) \odot \ldots \odot Sub_j(T(Lk))$ is defined where j is an integer that has not been used before.

Again, consider the example in Figure 1. The sequence <A(x), $\overline{A}$(y)>, <$\overline{C}$(g(y)), C(g(u))>, <$\overline{A}$(f(z)), A(x)> is a merge loop. The base node is A(x). The substitution of the loop is

$$[x/y]_{(x)(y)} \odot [y/u]_{(y)(u)} \odot [f(z)/x]_{(z)(x)}$$

$$= [f(z)/x,\ f(z)/y,\ f(z)/u]_{(x)(u,y,z)}.$$

$\overline{D}$(w) is the only undeleted residual. For the start clause {A(x), B(f(x))}, Figure 2 shows the solution tree for this example.

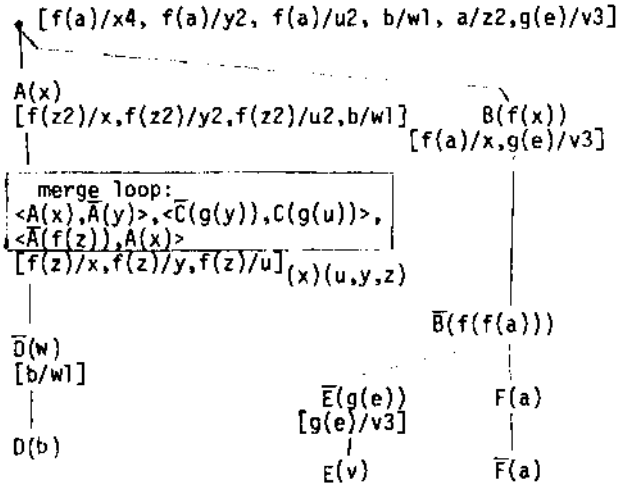


Figure 2: The only solution tree for Figure 1 with start clause {A(x),B(f(x))} Nonempty substitutions of the subtrees are denoted at the roots.

A context-free grammar is a quadruple: <Nonterminals, Terminals, Productions, Start symbol> in which

1) Nonterminals, Terminals and Productions are finite sets
2) Nonterminals ∩ Terminals = ϕ
3) ∀p ∈ Productions, p is of the form: N → s1 s2 ... sk for any finite k where N ∈ Nonterminals for 1≤i≤k, si ∈ Nonterminals ∪ Terminals. N is known as the left-hand side (l.h.s.) and s1...sk as the right hand side (r.h.s) of the production p.
4) Start symbol ∈ Nonterminals.

E.g., <N={S,A}, T={0,1,2}, P, S> where P is:

{S → 0 A 2
S → 0 2
A → 1 A
A → 1}

If we always use upper case Latin letters to denote nonterminals and never to denote terminals, and restrict the start symbol to be "S", then the production set will fully specify the grammar. We sometimes use this shortcut notation.

For any set of characters C, the set of all strings of those characters is denoted C*. E.g., for the above grammar, T* equals all strings made up only of symbols 0, 1, 2. The empty string, a string consisting of no symbols, is denoted ε. ε ∈ T* for all T.

If α and β are strings of symbols and there exists a production A → B1 ... Bn and by replacing a single occurrence of symbol A in α by the string B1 ... Bn we get β, then we say α ⇒ β. In the preceding grammar, 0A2 ⇒ 01A2. If α and β are strings such that for finite n≥1 $\alpha = \beta 1 \Rightarrow \beta 2 \Rightarrow \ldots \Rightarrow \beta n = \beta$, then we say α can be derived from β and denote it α⇒*β. A special case of this definition is that α⇒*α. S ⇒* 0111112 in the previous grammar.

The language L(G) of a context-free grammar G = <N,T,P,S> is the set of all terminal strings derivable from G, i.e., {x | x ∈ T* and S ⇒* x}.

A tree is a derivation tree[2] for G if:

1) Every node has a label which is in N ∪ T.
2) The label of the root is S.
3) If a node s has at least one descendant, and s has label A, then A must be in N.
4) If nodes s1, s2,...,sk are the direct descendants of node s, in order from the left, with labels A1,...,Ak respectively, then A → A1 A2...Ak must be a production in P.

Denote by D(A) a derivation tree for a string derivable from nonterminal A.

2. CIG's to Grammars

Grammars provide concise representations for *very* rich sets of objects. For example, all computable functions are grammatically describable. The functions that we are interested in are the ones that take statements of theorems as input and produce proofs of those theorems as output. This paper attacks that problem by transforming the input into a grammar that will generate exactly the set of proofs to the theorem.

Ground case. In the ground case, all substitutions are empty and are therefore all mutually compatible. Ignoring substitutions simplifies the problem so we will start here. From a ground case CIG:

C = <Nodes, Edges, Subst, Clause>,

construct a context-free grammar G. Let

G = <Nodes ∪ {S}, Edges, P, S>

where S ∉ Nodes ∪ Edges. P consists of:

a) S → L1...Lk for each starting clause {L1,...Lk}.
b) B → eC1...Ck for each edge e=<B,C> and Residual_literals(C)={C1,...Ck}.
c) B → mL1...Lk for each merge loop m and where L1...Lk are the undeleted residuals of m.

For example, consider the ground case CIG of Figure 3. The edges of Figure 3 are named a-d and the nodes A-G for purposes of exposition. Let the start clause be {A,B} and let e' denote edge <Y,X> if e denotes edge <X,Y>. Edges are directed to

designate in which direction the label applies.

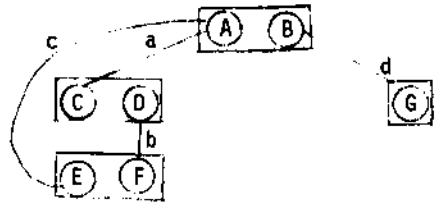


Figure 3: A ground CIG

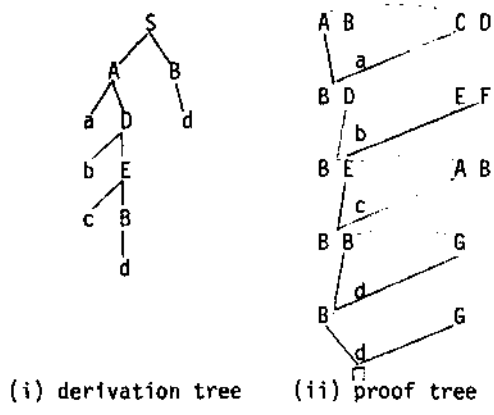
Then G is:

<{S,A,B,C,D,E,F,G}, {a,b,c,d}, P, S> where P is:

S → A B	C → a' B
A → a b c (a merge loop)	D → b E
A → a D	E → c B
A → c' F	F → b' C
B → d	G → d' A

L(G) = {abcdd, c'b'a'dd, abcd}.

A derivation tree for abcdd is shown in Figure 4.



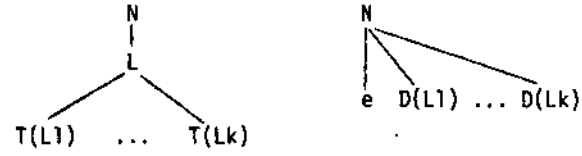
(i) derivation tree (ii) proof tree

Figure 4: Trees for abcdd

The string abcdd corresponds to the proof in which the sequence of resolutions corresponding to edges a, b, c, d, d are performed as shown in Figure 4(ii). A-G are merely labels, not the literals themselves. Because edges represent complementary pairs, A ^ E, D ^ F, B = G; dashed lines connect these pairs in the proof for the convenience of the reader. Each edge traversed is denoted at the corresponding resolution step, also for convenience.

Thm 1. For the ground case, the set of deletion trees for node N of CIG C is equivalent to the set of derivation trees for strings derivable from nonterminal N using the productions of the grammar G derived from C.

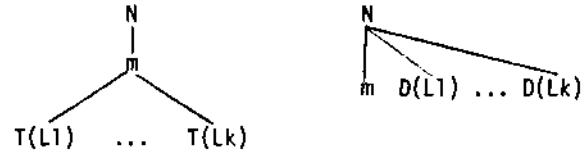
Proof The proof is by induction on the depth of the trees. Both cases contain a basis step as an instance. There are two operations for constructing each of the deletion trees and derivation trees. We shall show that the two pair correspond. The first production for G does not apply since we have no way of generating S from any other nonterminal. Part a Construction rule a for deletion trees corresponds to production rule b of grammars constructed from CIG's. Figure 5 shows the corresponding constructions.



(i) a construction rule for deletion trees (ii) a construction rule for derivation trees

Figure 5: A corresponding pair of construction rules for deletion and derivation trees.

For k=0, it is clear that L, as a child of N, can be mapped onto <N,L> and vice versa. For k>0, we assume that if T(Li) is equivalent to D(Li) for 1≤i≤k then Figure 5(i) is equivalent to Figure 5(ii). The only transformations to be made are between L and <N,L> and a simple tree reshaping. Part b Construction rule b for deletion trees corresponds to production rule c of grammars constructed from CIG's. Figure 6 shows the corresponding constructions.



(i) a construction rule for deletion trees (ii) a construction rule for derivation trees

Figure 6: A corresponding pair of construction rules for deletion and derivation trees

For k=0 the two trees are identical. For k>0, we assume that if T(Li) is equivalent to D(Li) for I^isk then Figure 6(i) is equivalent to Figure 6(ii) with the only required transformations being a simple tree reshaping.

From parts a and b we conclude that for any deletion tree we can construct a corresponding derivation tree and vice versa. Q.E.D.

Thm 2 For the ground case, the set of solution trees of a CIG is equivalent to the set of derivation trees for the language of the grammar G constructed from the CIG.

Proof Figure 7 shows the form of solution trees and derivation trees for L(G). For each Li, Isi^k, we can construct equivalent T(Li) and D(Li). It is therefore trivial to show that for any solution tree we can construct an equivalent derivation tree from S.

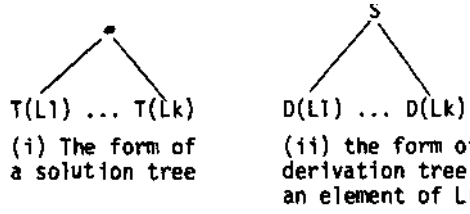


Figure 7: Forms of solution trees and derivation trees for L(G).

Thm 3 Every proof by resolution of the unsatisfiability of a set S of ground clauses can be mapped onto an element in the language L(G) where G is the grammar constructed from the CIG C constructed from S.

Proof Any proof by resolution of the unsatisfiability of S can be mapped onto a solution tree for C, [5]. By Thm 2 we know that every solution tree for C can be mapped onto a derivation tree of an element of G. Every derivation tree can be mapped onto the element of L(G) that consists of the leaves of the tree in the same left-to-right order.

Thm 4 (Soundness) Suppose G is the grammar derived from CIG C, derived in turn from the set of clauses S. If L(G) is non-empty, then S is unsatisfiable and any element of L(G) can be mapped onto a proof of the unsatisfiability of S.

Proof Suppose L(G) contains some string s. Every member of L(G) has a derivation tree that describes the process of deriving s. Let d be a derivation tree for s. From Thm 2 we can map d onto a solution tree of C. That solution tree maps onto a proof by resolution of the unsatisfiability of S, [5].

Thm 5 (Completeness) If a set S of clauses is unsatisfiable then the language L(G) is nonempty and any element of L(G) can be used to construct a proof of the unsatisfiability of S.

Proof Assume S unsatisfiable. Then there exists a refutation r of S by resolution[3]. By Thm 3 we know that r can be mapped onto an element s of L(G) and L(G) must therefore be nonempty. Furthermore, by Thm 4, s can be used to find a proof of the unsatisfiability of S.

Note: The terminals making up the string s are edge names. Those edge names may be used to represent resolution steps that collectively will generate the empty clause.

General case In deriving a string in a context-free language, any production may be applied whenever the current, derived string contains the nonterminal that is the l.h.s. of that production. We shall now define a similar type of grammar in which application of productions is further restricted.

A context-free attribute grammar is a context-free grammar in which the productions are replaced by production-attribute pairs: (P,A) where P is a production and A is a predicate. When applying a production the corresponding attribute must be true.

E.g., G = <{S}, {0,1}, P_A, S> where P_A consists of the two elements:

	P	A
1)	S → 0S1	derived string is of length less than or equal to 6
2)	S → 01	True

Then L(G) = {01, 0011, 000111, 00001111}

The method for constructing a grammar in the general case is basically the same as that in the ground case except that we add attributes to the productions. Each attribute corresponds to the substitution that must be made at that step and its compatibility with the substitutions that have already been made in the derivation.

Assume a given CIG. Find all merge loops. Now construct a context-free attribute grammar, <Nonterminals, Terminals, P_A, S> where:

S is a new symbol, i.e., S ∉ Nodes ∪ Edges
Nonterminals = {S} ∪ {Ni | N ∈ Nodes, i>0}
Terminals = Edges

P_A = (P,A)

1) P is S → L1 ... Lk for each start clause (L1,...,Lk); A is true. The empty substitution is the substitution of the newly derived string, or
2) P is B_j → e $C1_j$...Ck_j where e = <B,C> ∈ Edges, Residual_literals(C) = {C1...Ck}
A is true iff $\beta^T = \beta \odot \alpha_{ij}$ is defined where
β denotes the accumulated substitution of the current, derived string and α= Subst(e). β' is the substitution of the newly derived string, or
3) P is B_j → m $L1_j$...Lk_j where m is a merge loop, (L1,...Lk) are the undeleted literals of m. A is as in case 2 except that α = Sub(m).

The indexing of the nonterminals is used to keep track of different copies of the same variable. If more than one instance of a clause is used in the proof then the variables in those clauses must be distinguishable. We assume that variables in different clauses already have different names, so that the only possible ambiguity arises from multiple instances of a given clause.

Production-attribute pairs are really templates.† If nonterminal B_h appears in a derived string, then we may use the template

B_j → e $C1_j$...Ck_j

to create

B_h → e $C1_n$...Ck_n

where n is an integer constant not previously used

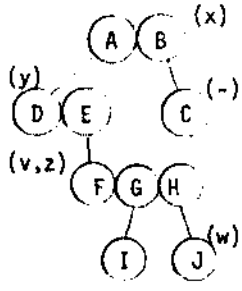
---

† The number of production templates is finite, but the number of actual productions obtainable will be infinite. It would have been possible, however, to limit ourselves to a finite set of productions and nonterminals by complicating the attributes.

as a nonterminal subscript. This production may now be applied to the derived string s if the attribute of the production is true, i.e., if substitution(s)$\odot a_{hn}$ is defined.

For example, consider the CIG C of Figure 8. Again, if e is edge <X,Y> let e' denote <Y,X>; if Subst(e) = $\alpha_{s1\ s2}$ then Subst(e') = $\alpha_{s2\ s1}$. The grammar constructed from C is G = <{S,A,B,C,D,E,F, G,H,I,J}, {1,2,3,4,5,1',2',3',4',5'},P_A,S> where P_A:

	P	A: true if $\beta' = \beta \odot \alpha$ defined where α is
1)	$S \rightarrow A_j B_j$	ϵ
2)	$A_i \rightarrow 2\ E_j$	$[f(x)/y]_{(x)(y)}$
3)	$B_i \rightarrow 1$	$[a/x]_{(x)(-)}$
4)	$E_i \rightarrow 3\ G_j\ H_j$	$[y/z]_{(y)(z)}$
5)	$G_i \rightarrow 4$	$[f(a)/z]_{(z)(-)}$
6)	$H_i \rightarrow 5$	$[b/w,a/v]_{(v)(w)}$
7)	$C_i \rightarrow 1'\ A_j$	$[a/x]_{(-)(x)}$
8)	$D_i \rightarrow 2'\ B_j$	$[f(x)/y]_{(y)(x)}$
9)	$F_i \rightarrow 3'\ D_j$	$[y/z]_{(z)(y)}$
10)	$I_i \rightarrow 4'\ F_j\ H_j$	$[f(a)/z]_{(-)(z)}$
11)	$J_i \rightarrow 5'\ F_j\ G_j$	$[b/w,a/v]_{(w)(v)}$



Edge e	Subst(e)
1	$[a/x]_{(x)(-)}$
2	$[f(x)/y]_{(x)(y)}$
3	$[y/z]_{(y)(z)}$
4	$[f(a)/z]_{(z)(-)}$
5	$[b/w,a/v]_{(v)(w)}$

Figure 8: A CIG with variable lists attached to clause partitions

L(G) equals the single element set {23451}.

The productions in our constructed attribute grammar tell us how to generate strings, their corresponding derivation trees and associated substitutions, top-down. In order to prove the completeness theorem for the general case, we need to define the bottom-up construction of substitutions of derivation trees. The following is such a definition. Proof of equivalence of this to the original definition is simple and left to the reader.

For the derivation tree shown in Fig. 9(i),
$Sub(D(B_i)) = Subst(e)_{i,j} \odot Sub(D(Cl_j)) \odot \ldots \odot Sub(D(Ck_j))$.

For the derivation tree shown in Fig. 9(ii),
$Sub(D(B_i)) = Sub(m)_{i,j} \odot Sub(D(Ll_j)) \odot \ldots \odot Sub(D(Lk_j))$.

For the derivation tree shown in Fig. 9(iii),
$Sub(D(S)) = Sub(D(Ll_j)) \odot \ldots \odot Sub(D(Lk_j))$.

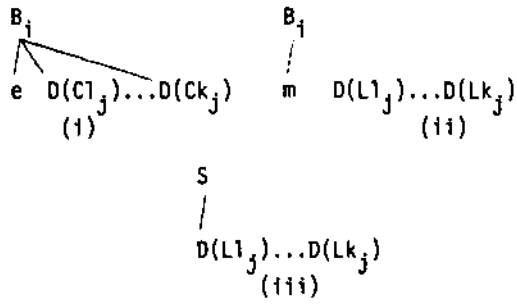


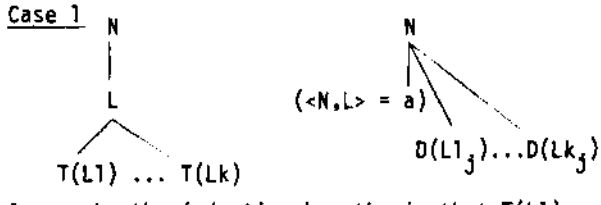
Figure 9: All possible configurations of derivation trees

Thm 6. (General case, Completeness and Soundness) The set of solution trees of a CIG is equivalent to the set of derivation trees for the language of the attribute grammar constructed from the CIG.

Proof Consider first the ground structure, i.e. the CIG without substitutions and the grammar with the attributes ignored. In this case, productions enjoy unrestricted use. Looking only at the ground structure, we are reduced to the situation of thms 1 and 2. Therefore the ground structure of the two are equivalent. The general case puts restrictions on those ground structures. What we need to show is that the restrictions allow the equivalent structures to be admitted at the general level.

For the same case breakdown as before, we shall show that the structures the two systems admit are equivalent and have equivalent substitutions. The proof will be by induction on the depth of the trees. The basic steps are the special instances of cases 1 and 2 in which k=0.

Case 1



Assume by the induction hypothesis that T(Ll),..., T(Lk) and D(Ll),...,D(Lk) are admitted by their respective systems, and that $Sub(T(Lh)) = Sub(D(Lh_j))$ with a possible change of variables for $1 \leq h \leq k$. Then let

$\alpha = Sub(T(N)) =$

$Subst(<N,L>)_{-,j} \odot Sub_j(T(Ll)) \odot \ldots \odot Sub_j(T(Lk))$

$\beta = Sub(D(N_1)) =$

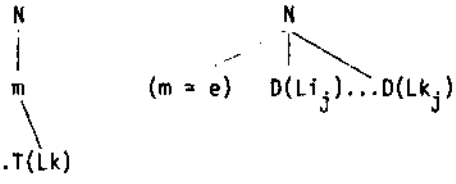
$Subst(<N,L>)_{i,j} \odot Sub(D(Ll_j)) \odot \ldots \odot Sub(D(Lk_j))$.

For $1 \leq h \leq k$, all variables in $Sub(D(Lh_j))$ are subscripted since every substitution applied under the grammar is fully subscripted; furthermore, all variables of Lh_j are subscripted by j. All variables of Sub(T(Lh)) are subscripted except for those in Lh, since by the definition of dele-

tion tree, before forming Sub(T(Lh)) we subscript all unsubscripted variables. $Sub_j(T(Lh))$ then equals Sub(T(Lh)) with all variables of Lh now subscripted. Therefore, if by the induction hypothesis, Sub(D(Lh_j)) equals Sub(T(Lh)) up to a change, cv, of variables, then Sub(D(Lh_j)) equals Sub_j(T(Lh)) up to cv', which is equal to cv minus the changes involving unsubscripted variables.

Therefore, Sub(D(N)) and Sub(T(N)) are either both or neither defined, and Sub(D(N)) = Sub(T(N)) with cv' plus change in Sub(T(N)) the unsubscripted variables x of N to x_i.

Case 2



α = Sub(T(N)) =

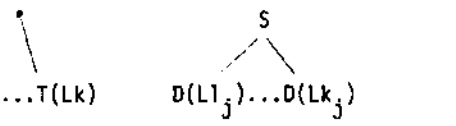
$Sub(m)_{-,j} \cup Sub_j(T(L1)) \odot \ldots \cup Sub_j(t(Lk))$ and

β = Sub(D(N)) =

$Sub(m)_{i,j} \cup Sub(D(L1_j)) \cup \ldots \cup Sub(D(Lk_j))$

$Sub(D(Lh_j)) = Sub_j(T(Lh))$ with change cv' for $1 \leq h \leq k$ as in Case 1. Therefore Sub(D(N)) and Sub(T(N)) are either both or neither defined, and Sub(D(N)) = Sub(T(N)) with cv' plus change in Sub(T(N)) the unsubscripted variables x of m to x_i.

Case 3



α = Sub(Ts(C)) = $Sub_j(T(L1)) \cup \ldots \cup Sub_j(T(Lk))$ and

β = Sub(D(S)) = $Sub(D(L1_j)) \cup \ldots \cup Sub(D(Lk_j))$

$Sub(D(Lh_j)) = Sub_j(T(Lh))$ with change cv' for $1 \leq h \leq k$ as in Case 1. Therefore Sub(D(S)) and Sub(Ts(C)) are either both or neither defined, and Sub(D(S)) = Sub(Ts(C)) with change cv'.

3. Grammar G to Language L(G)

In the theorem proving application just described, a context-free grammar is defined whose language represents proofs of a theorem. A description of that language is a description of the set of objects we desire. We describe a concise, closed form and discuss how to derive it.

In the case of regular grammars (a special case of context-free grammars), we can describe the generatable language as a regular expression. Regular expressions are defined as follows:

a) Every terminal symbol is a regular expression.
b) If A and B are regular expressions, then (A), A|B, AB, and A* are also.
c) No sequence of symbols not satisfying a) or b) above is a regular expression.

Parentheses are used to enclose subexpressions, "A|B" is used to denote the choice of A or B, "AB" to denote concatenation of regular expressions A and B, and "A*" denotes that A is repeated zero or more times.

E.G. for grammar G:

S → 0 S	A → 1 B
S → 1 A	B → 2
A → 1 A	B → 3

L(G) is represented by the regular expression 0* 1 1 1* (2|3), i.e. any element of L(G) consists of zero or more "0"'s followed by two or more "1"'s followed by either a "2" or a "3".

Context-free grammars sometimes generate languages that are not representable by regular expressions. For example, grammar:

S → (S)
S → e

generates language: $(^n\ e\ )^n$, $n \geq 0$, i.e. all strings that consist of zero or more open parens followed by "e" followed by the same number of close parens as open parens. There is no way to express $(^n\ e\ )^n$ strictly as a regular expression since the only repetition operator, "*", does not carry a count that can be matched. E.g., (* e)* would allow "(((e)". In order to eliminate this difficulty, we add to the regular expression notation the positive integer exponent. We define a regular algebra R that admits the following expressions:

a) Every terminal symbol 1s in R.
b) If A and B are in R, then (A), A|B, AB, A^* and A^n are in R.
c) No sequence not satisfying a) or b) is in R.

The addition of exponents is essential since the inherent power of context-free grammars allows balanced bracketing of expressions. Some context-free grammars naturally create expressions in R. Several simple operations can be used to generate the expression in those cases.

1) Back-substitution. If for any nonterminal A all productions having A as the l.h.s. have no nonterminals on the r.h.s., then replace all references to A by the alternative terminal strings or expressions in R that A derives. Then remove all productions having A as the l.h.s.. E.g.,

A → a		
A → b* c	⇒	B → a (a b*c)c
B → a A c		

2) Simple recursion. If for any nonterminal A the only productions having A as the l.h.s. are of the form:

(type 1) A → ti, $1 \leq i \leq n$ or
(type 2) A → 1i A, $1 \leq i \leq k$ or
(type 3) A → A ri, $1 \leq i \leq j$

where ti, 1i, and ri represent expressions in R, where $n \geq 1$, $k,j \geq 0$, then

$A \triangleq (1_1|\ldots|1k)^* (t_1|\ldots|tn) (r_1|\ldots|rj)^*$.

Replace every reference to A in other productions by this expression representing strings derivable

from A. Eliminate all productions having A as the l.h.s.. E.g.,

A → A 0
A → b 1* A
A → 2 A
A → a ⇒ S → 0 (b 1* | 2)* (a|b) 0*
A → b
S → 0 A

3) Internal recursion. If for any nonterminal A, the only productions having A as the l.h.s. are of the form:

(type 1) A → ti, 1≤i≤n
(type 2) A → 1 A r

where ti, 1 and r represent expressions in R, where n≥1, and there is a single production of type 2, then $A \rightarrow 1^n (t1|...|tn)\ r^n$.
Replace A as before and eliminate the above productions. E.g.,

A → (a|b)
A → 1* 2 ⇒ $B \rightarrow b\ 0^n\ (1^*\ 2\ \ (a|b))\ 1^n\ 1$
A → 0 A 1
B → b A 1

The above three replacement rules will not suffice for all context-free grammars. For example the grammar G:

S → 0 S 1 S → S 3
S → 2 S S → a

has language (0|2)* a (1|3)* such that the number of O's equals the number of l's. We cannot, strictly speaking, represent that by our regular algebra. However we can loosen the ordering restriction on the r.h.s. of productions since the r.h.s.'s represent subgoals to be solved and the order is unimportant. This allows us to represent the language as (01|2)* a 3*. A later paper will discuss this *ordering* relaxation and double recursion, e.g. A → 0 A 1 A.

Loosening the ordering restriction also allows the elimination of some redundancy. The first grammar derived from a CIG that appears in this paper generates the following language: (abcdd, c'b'a'dd, abed, cbad). Allowing reordering and representing edges <X,Y> and <Y,X> similarly, then the set reduces to (abcdd, abed}. For further discussion of minimizing the set representing the proof schemata, see [7].

4. Conclusions

We have described a method that represents proofs in predicate logic by a formal language. This language includes the full set of proofs for a given theorem. If we can describe the language with a regular algebra then we have a closed form for a possibly infinite set of proofs. This representation gives an analysis of the flow of the derivation. This analysis can be used on programs specified in logic to describe required execution flow of the program that leads to termination with the proper result.

The regular algebra representation can also be used to analyze the values that variables may have if the derivation is to terminate. In the logic program application, this analysis will be used to clarify the possible values of input and output variables, i.e. the domain and range of the function computed by the program.

Studies are underway to apply this regular algebra representation of logic specifications. Some of the areas presently being considered are:

1) program synthesis
2) plan formation
3) question-answering
4) machine learning.

References

1) Chang, C.L. and R.C.T. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

2) Hopcroft, J. and J. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, Menlo Park, 1969.

3) Robinson, J.A., "A machine-oriented logic based on the resolution principle", JACM 12, 1(Jan,1965).

4) Sickel, S.,"A search technique for clause interconnectivity graphs", IEEE Transactions on Computers, Aug. 1976.

5) Sickel, S., "Completeness of clause interconnectivity graphs", Technical Rpt., University of California, Santa Cruz, 1977.

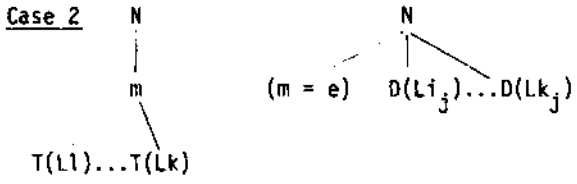
6) Sickel, S., "A linguistic approach to automatic theorem proving", CSCSI/SCEIO Summer Conference Proceedings, 1976.

7) Sickel, S., "A proof description language for first-order predicate calculus", Technical Rpt., University of California, Santa Cruz, 1977.

tion tree, before forming Sub(T(Lh)) we subscript all unsubscripted variables. $Sub_j(T(Lh))$ then equals Sub(T(Lh)) with all variables of Lh now subscripted. Therefore, if by the induction hypothesis, Sub(D(Lh_j)) equals Sub(T(Lh)) up to a change, cv, of variables, then Sub(D(Lh_j)) equals Sub_j(T(Lh)) up to cv', which is equal to cv minus the changes involving unsubscripted variables.

Therefore, Sub(D(N)) and Sub(T(N)) are either both or neither defined, and Sub(D(N)) = Sub(T(N)) with cv' plus change in Sub(T(N)) the unsubscripted variables x of N to x_i.

Case 2



α = Sub(T(N)) =

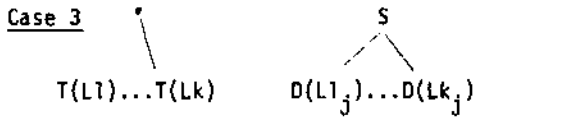
$Sub(m)_{i,j} \cup Sub_j(T(Ll)) \cup \ldots \cup Sub_j(t(Lk))$ and

β = Sub(D(N)) =

$Sub(m)_{i,j} \cup Sub(D(Ll_j)) \cup \ldots \cup Sub(D(Lk_j))$

$Sub(D(Lh_j)) = Sub_j(T(Lh))$ with change cv' for $l \leq h \leq k$ as in Case 1. Therefore Sub(D(N)) and Sub(T(N)) are either both or neither defined, and Sub(D(N)) = Sub(T(N)) with cv' plus change in Sub(T(N)) the unsubscripted variables x of m to x_i.

Case 3



$\alpha = Sub(Ts(C)) = Sub_j(T(Ll)) \cup \ldots \cup Sub_j(T(Lk))$ and

$\beta = Sub(D(S)) = Sub(D(Ll_j)) \cup \ldots \cup Sub(D(Lk_j))$

$Sub(D(Lh_j)) = Sub_j(T(Lh))$ with change cv' for $l \leq h \leq k$ as in Case 1. Therefore Sub(D(S)) and Sub(Ts(C)) are either both or neither defined, and Sub(D(S)) = Sub(Ts(C)) with change cv'.

3. Grammar G to Language L(G)

In the theorem proving application just described, a context-free grammar is defined whose language represents proofs of a theorem. A description of that language is a description of the set of objects we desire. We describe a concise, closed form and discuss how to derive it.

In the case of regular grammars (a special case of context-free grammars), we can describe the generatable language as a regular expression. Regular expressions are defined as follows:

a) Every terminal symbol is a regular expression.
b) If A and B are regular expressions, then (A), A|B, AB, and A* are also.
c) No sequence of symbols not satisfying a) or b) above is a regular expression.

Parentheses are used to enclose subexpressions, "A|B" is used to denote the choice of A or B, "AB" to denote concatenation of regular expressions A and B, and "A*" denotes that A is repeated zero or more times.

E.G. for grammar G:

S → 0 S	A → 1 B
S → 1 A	B → 2
A → 1 A	B → 3

L(G) is represented by the regular expression 0* 1 1 1* (2|3), i.e. any element of L(G) consists of zero or more "0"'s followed by two or more "1"'s followed by either a "2" or a "3".

Context-free grammars sometimes generate languages that are not representable by regular expressions. For example, grammar:

S → (S)
S → e

generates language: $(^n\ e\ )^n$, $n \geq 0$, i.e. all strings that consist of zero or more open parens followed by "e" followed by the same number of close parens as open parens. There is no way to express $(^n\ e\ )^n$ strictly as a regular expression since the only repetition operator, "*", does not carry a count that can be matched. E.g., (* e)* would allow "(((e)". In order to eliminate this difficulty, we add to the regular expression notation the positive integer exponent. We define a regular algebra R that admits the following expressions:

a) Every terminal symbol is in R.
b) If A and B are in R, then (A), A|B, AB, A^* and A^n are in R.
c) No sequence not satisfying a) or b) is in R.

The addition of exponents is essential since the inherent power of context-free grammars allows balanced bracketing of expressions. Some context-free grammars naturally create expressions in R. Several simple operations can be used to generate the expression in those cases.

1) Back-substitution. If for any nonterminal A all productions having A as the l.h.s. have no nonterminals on the r.h.s., then replace all references to A by the alternative terminal strings or expressions in R that A derives. Then remove all productions having A as the l.h.s.. E.g.,

A → a		
A → b* c	⇒	B → a (a b*c)c
B → a A c		

2) Simple recursion. If for any nonterminal A the only productions having A as the l.h.s. are of the form:

(type 1) A → ti, $1 \leq i \leq n$ or
(type 2) A → 1i A, $1 \leq i \leq k$ or
(type 3) A → A ri, $1 \leq i \leq j$

where ti, 1i, and ri represent expressions in R, where $n \geq 1$, $k, j \geq 0$, then

A ⇒ $(1_1|\ldots|1k)^*\ (t_1|\ldots|tn)\ (r_1|\ldots|rj)^*$.

Replace every reference to A in other productions by this expression representing strings derivable

from A. Eliminate all productions having A as the l.h.s.. E.g.,

A → A 0
A → b 1* A
A → 2 A
A → a ⇒ S → 0 (b 1* | 2)* (a|b) 0*
A → b
S → 0 A

3) Internal recursion. If for any nonterminal A, the only productions having A as the 1.h.s. are of the form:

(type 1) A → ti, 1≤i≤n
(type 2) A → 1 A r

where ti, 1 and r represent expressions in R, where n≥1, and there is a single production of type 2, then $A \Rightarrow 1^n (t1|...|tn)\ r^n$.

Replace A as before and eliminate the above productions. E.g.,

A → (a|b)
A → 1* 2
A → 0 A 1 ⇒ B → b 0^n (1* 2 (a|b)) 1^n 1
B → b A 1

The above three replacement rules will not suffice for all context-free grammars. For example the grammar G:

S → 0 S 1 S → S 3
S → 2 S S → a

has language (0|2)* a (1|3)* such that the number of O's equals the number of l's. We cannot, strictly speaking, represent that by our regular algebra. However we can loosen the ordering restriction on the r.h.s. of productions since the r.h.s.'s represent subgoals to be solved and the order is unimportant. This allows us to represent the language as (01|2)* a 3*. A later paper will discuss this ordering relaxation and double recursion, e.g. A → 0 A 1 A.

Loosening the ordering restriction also allows the elimination of some redundancy. The first grammar derived from a CIG that appears in this paper generates the following language: {abcdd, c'b'a'dd, abed, cbad}. Allowing reordering and representing edges <X,Y> and <Y,X> similarly, then the set reduces to {abcdd, abed}. For further discussion of minimizing the set representing the proof schemata, see [7],

4. Conclusions

We have described a method that represents proofs in predicate logic by a formal language. This language includes the full set of proofs for a given theorem. If we can describe the language with a regular algebra then we have a closed form for a possibly infinite set of proofs. This representation gives an analysis of the flow of the derivation. This analysis can be used on programs specified in logic to describe required execution flow of the program that leads to termination with the proper result.

The regular algebra representation can also be used to analyze the values that variables may have if the derivation is to terminate. In the logic program application, this analysis will be used to clarify the possible values of input and output variables, i.e. the domain and range of the function computed by the program.

Studies are underway to apply this regular algebra representation of logic specifications. Some of the areas presently being considered are:

1) program synthesis
2) plan formation
3) question-answering
4) machine learning.

References

1) Chang, C.L. and R.C.T. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

2) Hopcroft, J. and J. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, Menlo Park, 1969.

3) Robinson, J.A., "A machine-oriented logic based on the resolution principle", JACM 12, 1(Jan,1965).

4) Sickel, S.,"A search technique for clause interconnect!"vity graphs", IEEE Transactions on Computers. Aug. 1976.

5) Sickel, S., "Completeness of clause interconnectivity graphs", Technical Rpt., University of California, Santa Cruz, 1977.

6) Sickel, S., "A linguistic approach to automatic theorem proving", CSCSI/SCEIO Summer Conference Proceedings, 1976.

7) Sickel, S., "A proof description language for first-order predicate calculus", Technical Rpt., University of California, Santa Cruz, 1977.