PROVING LISP PROGRAMS USING TEST DATA

Timothy A. Budd

Richard J. Lipton

Yale University

Department of Computer Science

New Haven, Ct.

and

University of California at Berkeley

Computer Science Division

Berkeley, Calif.

1. INTRODUCTION

An idea proposed in [1] is the concept of proving individual programs correct with respect to some larger class of programs. That is, instead of proving a program correct we prove that either a) the program is correct, OR b) no program in this
class realizes the intended function. It is assumed that most programmers at least know if the function they are trying to compute can be realized in some large class of programs, and therefore from a theoretical point of view the introduction of this disjunction may make the task of validating programs vastly easier.

A previous paper has analysed programs written in a decision table format [4]. In this paper we will be concerned with lisp programs composed of CAR, CDR and CONS with lisp predicates composed of CAR, CDR and ATOM. Similar classes of programs have been studied in [5,6,7].

Associated with each S-Expression X we can construct a binary tree as follows: Consider the infinite binary tree where each left arc is marked CAR and each right arc CDR (call this the complete CAR/CDR tree.) Starting with X at the root of the tree, travel down each arc in turn taking the appropriate CAR or CDR. Prune the complete tree each time you reach an atom. The resulting finite binary tree will be called the projection of X (or PROJ[X]). An example is shown in figure 1. Notice the PROJ[X] is a representation of the structure of X, and in invariant under the renamings of the atoms of X.
We can define a relation $<$ as follows. Given two S-expressions $X$ and $Y$ we will say $X < Y$ if PROJ[$X$] is the intersection of PROJ[$X$] and PROJ[$Y$]. Using this relation one can show the set of lisp structures form a lattice. (The proofs can be adapted from Summers[7], although he defines the projection slightly differently.)

We will make the convention that all S-Expressions (we will use the less clumsy expression point) have unique atoms. Certainly if two programs agree on all such points they must agree on all inputs. Hence we can do this without loss of generality.

We will call a lisp program a Selector program if it is composed of just CAR and CDR. We will call it a Straight line program if it is a selector program or is formed by CONS on either selectors or other straight line programs. We will call it a Predicate program if it has the following form

\[
\text{COND} \left( \text{ATOM}(G_1(X)) \rightarrow P_1(X) \right)
\]

\[
T \rightarrow P_2(X)
\]

Where the G's are selectors and the P's are straight line programs or other predicate programs.

Assume we have a function $F$ which we know can be computed by a program in some schemata class $S$.  

376
We have a program $P$ in $S$ which we wish to show computes $F$. We assume we have some method of verifying that $P(X) = F(X)$ on a finite number of test cases (say by hand calculation.) We wish to show that there exists a finite set of test cases $T$ such that if $P$ correctly computes $F$ on every element of $T$ then either 1) $P$ correctly computes $F$ for all inputs, or 2) no program in the schemata class $S$ correctly computes $F$. This goal is similar to that of mutation analysis [1-4].

Call such a test set Adequate.

We then wish to discover conditions under which we can construct adequate test data.

2. STRAIGHT LINE PROGRAMS

We will say a program $P(X)$ is Well formed if for every occurrence of the construction $\text{CONS}(A,B)$ it is the case that $A$ and $B$ do not share an immediate parent in $X$. The intuitive idea of the definition should be clear: a program is well formed if it is not doing any more work than it needs to. Notice that being well formed is an observable property of programs, independent of testing.

We can define a measure of the complexity of straight line programs by their CONS-depth, where
CONS-depth is defined as follows:

1) The CONS-depth of a selector function is zero.

2) The CONS-depth of a straight line program \( P(X) = \text{CONS}(P_1(X), P_2(X)) \) is \( 1 + \text{MAX} \left( \text{CONS-depth}(P_1(X)), \text{CONS-depth}(P_2(X)) \right) \).

Lemma 1: If any two selector programs compute identically on any point \( X \), they must compute identically on all points.

Proof: The only power of a selector program is to choose a subtree out of its input and return it. We can view this process as selecting a position in the complete CAR/CDR tree and returning the subtree rooted at that position. Since there is a unique path from the root to this position, there is a unique predicate which selects it out. Since atoms are unique by merely observing the output we can infer the subtree which was selected. The result then follows.

Lemma 2: If two well formed programs compute identically on any point then they must have the same CONS-depth.

Proof: Assume we have two programs \( P_1 \) and \( P_2 \) and a point \( X \) such that \( P_1(X) = P_2(X) \) yet the CONS-depth\((P_1) < \text{CONS-depth}(P_2) \). This then
implies that there is at least one subtree in the structure of P₂ which was produced by CONS-ing two straight line programs while the same subtree in P₁(X) was produced by a selector. But then the objects P₂ CONSed must have an immediate ancestor in X, contradicting the fact that P₂ is well formed.

THEOREM 1: If two well formed straight line programs agree on any point X then they must agree on all points.

PROOF: The proof will be by induction on the CONS-depth. By lemma 2 any two programs which agree at X must have the same CONS-depth. By lemma 1 the theorem is true for programs of CONS-depth zero. Hence we will assume it is true for programs of CONS-depth n and show the case for n+1.

If program P₁ has CONS-depth n+1 then it must be of the form CONS(P₁₁, P₁₂) where P₁₁ and P₁₂ have CONS-depth no greater then n. Assume we have two programs P₁ and P₂ in this fashion. Then for all Y:

\[ P₁(Y) = P₂(Y) \iff \]
\[ CONS(P₁₁(Y), P₁₂(Y)) = CONS(P₂₁(Y), P₂₂(Y)) \iff \]
\[ P₁₁(Y) = P₂₁(Y) \text{ and } P₁₂(Y) = P₂₂(Y) \]

Hence by the induction hypothesis P₁ and P₂
must agree for all Y.

We define a test point to Generic if by itself it constitutes an adequate test set as defined in the introduction.

Corollary: For any well formed straight line lisp program, and unique atomic point for which the function is defined is generic.

3. PREDICATE PROGRAMS

We can view the structure of a predicate program as a binary tree. Associated with each interior node is a predicate and associated with each leaf is a straight line program (see figure.)

We will call a predicate program Well formed if

1) each of the straight line programs associated with each leaf are well formed, and

2) for each leaf on the space of all possible inputs there is at least one item which passes all conditions leading to that leaf and causes the associated straight line program to be executed.

Notice that whether a program is well formed or not is an observable fact independent of testing.
For notation we will denote the leaves going from left to right by \( l_i \), \( i = 1, \ldots, n \). Let \( e_i \), \( i = 1, \ldots, n \) be the set of straight line programs associated with the leaves. We will assume that for no \( i, j \) if \( i \neq j \) is it the case that \( e_i \) is equivalent to \( e_j \). Notice again theorem 1 gives us an effective method to test this.

Given a well formed predicate program \( P \) is \( S \) we construct a set of \( n \) data points \( d_1, \ldots, d_n \) such that \( d_i \) follows the path to leaf \( l_i \) and executes the program \( e_i \) correctly. Call this set \( T_i \). There is an obvious effective procedure to generate such a test set.

**LEMMA 3:** Given any well formed program \( P' \) in 3 which evaluates correctly on each element of \( T \), at least one data point \( d_i \) in \( T \) must exercise every straight line leaf program in \( P' \).

**PROOF:** Assume we have a program \( P' \) satisfying the hypothesis but for which the conclusion is false. By the pigeon hole principle there must be at least two points \( d_i \) and \( d_j \) which were evaluated by different leaves in \( P \) but which are evaluated by the same leaf in \( P' \). Let \( f \) denote the straight line program which evaluates these points in \( P' \). Since the \( d \) points are generic this implies that \( e_i \) is equivalent to \( f \). But also \( e_j \) is equivalent to \( f \). Hence \( e_i \) must be equivalent to \( e_j \) which is a contradiction.
Corollary: Given any well formed program $P'$ in $S$ which evaluates correctly on each element of $T$, the leaf programs of $P'$ are simply a permutation of those of $P$.

It might seem that exercising all the paths of $P'$ is sufficient to show it is equivalent to $P$. But this is not the case. We might simply have consistently chosen the right path for the wrong reason. To rule out this possibility requires a more stringent set of test cases. We construct this test set in the following manner.

For each leaf $l_i$ and for each element $d_j$ in $T_j$ construct a point $d_{ij}$ in the following way. Consider the infinite CAR/CDR tree. Color each point RED which is tested and found to be atomic on the path leading to the leaf $l_i$. Color the points which are tested and found to be non atomic BLUE. As long as it is not contained in a subtree rooted at a red point and does not contain a blue point in its subtree, color a point red if it is atomic in $d_j$. As long as it is not contained in a subtree rooted at a red point, color a point blue if it is not atomic in $d_j$. $d_{ij}$ is then the smallest unique atomic point where the red colored vertexes are atomic and the blue vertexes non atomic.
Denote by $T$ the set $T_1$ augmented with these points.

THEOREM 2: Any well formed program $P'$ in $S$ which agrees with $P$ on $T$ must agree with $P$ on all points.

PROOF: Assume we have a program $P'$ which satisfies the hypothesis, yet there is a point $X$ such that $P(X)$ and $P'(X)$ differ.

The point $X$ must be evaluated by some leaf $l_i$ in $P$, hence it must satisfy all the constraints associated with that leaf.

This point is also evaluated by a leaf program $e_k$ in $P'$. By lemma 6 some data item $d_j$ in $T$ also executes this leaf program. This implies that no matter what the constraints are on this path in $P'$ (and we make no assumptions about what they might be) they cannot interfere with the constraints along the path leading the $l_i$.

But this then necessarily implies that point $d_{ij}$ would be evaluated by $e_i$ in $P$ and $e_k$ in $P'$ where $k \neq i$. Since $d_{ij}$ is also generic using the earlier theorems a contradiction is obtained.

Corollary: There is an effective procedure to construct an adequate test set for predicate programs.
4. recursive programs

We will define a class of programs \( \mathcal{D}_n \) as follows:
The input to the program shall consist of two sets of variables: Selector variables, denoted \( x_1, \ldots, x_m \) and Constructor variables, denoted \( y_1, \ldots, y_p \).
a program will consist of two parts, a program body and a recursor.
A program body consists of \( n \) statements, each statement composed of two parts. The first part is a predicate of the form \( \text{ATOM}(t(x_1)) \) where \( t(x_1) \) is a selector function and \( x_1 \) a selector variable. The second part is a straight line output function over the selector and constructor variables.
A recursor is divided into two parts. The constructor part is composed of \( p \) assignment statements for each of the \( p \) constructor variables where \( y_i \) is assigned a straight line function of the selector variables and \( y_i \). The selector part is composed of \( m \) assignment statements for the \( m \) selector variables so that \( x_1 \) is assigned a selector function of itself. The following diagram should give a more intuitive picture of this class of programs.

Program \( P(x_1, \ldots, x_m, y_1, \ldots, y_p) = \)

\[
p_1(x_1) \rightarrow f_1(x_1, \ldots, x_m, y_1, \ldots, y_p)
\]
\[ p_2(x_{i2}) \rightarrow f_2(x_1, \ldots, x_m, y_1, \ldots, y_p) \]
\[
\ldots
\]
\[ p_n(x_{in}) \rightarrow f_n(x_1, \ldots, x_m, y_1, \ldots, y_p) \]
\[ y_1 \leftarrow g_1(y_1, x_1, \ldots, x_m) \]
\[
\ldots
\]
\[ y_p \leftarrow g_p(y_p, x_1, \ldots, x_m) \]
\[ x_1 \leftarrow h_1(x_1) \]
\[
\ldots
\]
\[ x_m \leftarrow h_m(x_m) \]

Given such a program, execution proceeds as follows: Each predicate is evaluated in turn. If any predicate is undefined so is the result of the execution, otherwise if any predicate is TRUE the result of execution is the associated output function. Otherwise if no predicate evaluates true then the assignment statements in the recurer and constructor are performed and execution continues with these new values.

We will say a variable is a predicate variable if it is tested by at least one predicate. Similarly it is an output variable if it is used in at least one output function. A variable can be both a predicate and an output variable.

We will make the following restrictions on the programs we will consider:
1) every recursion selector and every constructor must be
non trivial.
2) every variable is either a predicate or an output variable.
3) there is at least one output variable
4) (freedom) for and 1<k<n and l>0 there exists a set of inputs which cause the program to recurse l times before correctly exiting by output function k.
5) each output function is unique.
6) every constructor variable appears totally in at least one output function.

Given a program P in \( \mathcal{D}_n \), let \( \mathcal{D} \) be the union of \( \mathcal{D}_i \) for \( i=1,n \).

Let us assume we know, on independent grounds, that a correct program \( P^* \) exists in \( \mathcal{D} \), furthermore that no predicate, output function, selector or constructor in \( P^* \) has a depth greater than some constant \( u>3 \).

**GOAL:** We wish to construct a set of test inputs with the property that any program \( P \) in \( \mathcal{D} \) which executes correctly on these values must then be equivalent to \( P^* \). The existence of such a test set would then imply (under the assumption that at least one correct program exists in \( \mathcal{D} \)) that \( P \) is correct.

We will use capital letters from the end of the alphabet (X, Y and Z) to represent vectors of inputs.
Hence we can refer to \( P(X) \) rather than \( P(x_1, \ldots, x_m, y_1, \ldots, y_p) \). Similarly we can abbreviate the simultaneous application of constructor functions by \( C(X) \) and recursion selectors by \( S(X) \).

We will use the initial greek letters to represent positions in a variable, where a position is defined by a finite CAR-CDR path from the root. When no confusion can arise we will frequently refer to "position \( \xi \) in \( X \)" whereby we mean position \( \xi \) in some \( x_i \) in \( X \).

We can form a lattice on the space of inputs by saying \( X \preceq Y \) if and only if for all selector variables \( x_i \) in \( X \) are smaller than their respective variables in \( Y \), and similarly the constructor variables.

We can define the notion of "Pruning \( X \) at position \( \xi \)" as follows: We will say \( Y \) is \( X \) "pruned at position \( \xi \) if \( Y \) is the largest input \( \preceq X \) where \( \xi \) is atomic. This process can be viewed as simply taking the subtree in \( X \) rooted at \( \xi \) and replacing it by a unique atom.

If a position \( \xi \) (relative to the original input) is tested by some predicate we will say that the position in question has been touched.

The assumption of freedom asserts only the existence of inputs \( X \) which will cause us to recurse a specific number of times and exit by a specific output function.
Our first lemma shows that this can be made constructive.

LEMMA 1. Given $l \geq 0$ and $1 \leq i \leq n$ we can construct an input $X$ such that $P(X)$ is defined and while executing $X \ P$ recurses $l$ times before exiting by output function $i$.

PROOF: Consider $m+p$ infinite trees corresponding to the $m+p$ input variables. Mark in BLUE every position which is touched by a predicate function and found to be non-atomic in order for $P$ to recurse $l$ times and reach the $i^{th}$ predicate. Then mark in RED the point touched by the $i^{th}$ predicate after recursing $l$ times.

The assumption of freedom implies that no blue vertex can appear in the infinite subtree rooted at the red vertex, and that the red vertex can not also be marked blue.

Now mark in YELLOW all points which are touched by constructor functions in recursing $l$ times, and each position touched by the $i^{th}$ output function after recursing $l$ times. The assumption of freedom again tells us that no yellow vertex can appear in the infinite subtree rooted at the red vertex. The red vertex may, however, also be colored yellow, as may the blue vertexes. It is a simple matter to then construct an input $X$ such that

1) all BLUE vertexes are non atomic in $X$,

2) The RED vertex is atomic, and

3) all YELLOW vertexes are contained in $X$ (they may be
atomic)

It is trivial to verify that such an \( X \) satisfies our requirements. \( \triangle \)

Notice that the procedure given in the proof of lemma 1 allows us to find the smallest \( X \) such that the indicated conditions hold. If \( \alpha \) is the position touched by the \( i \)th predicate after recursing \( l \) times call this point the minimal \( \alpha \) point, or \( X_\alpha \).

Freedom implies no point can be twice touched, hence the minimal \( \alpha \) point is a well defined concept.

Given an input \( X \) such that \( P(X) \) is defined, let \( F_X(Z) \) be the straight line function such that \( F_X(X) = P(X) \). Note that by the property of being generic, \( F_X \) is defined by this single point.

LEMMA 2: For any \( X \) for which \( P(X) \) is defined, we can construct an input \( Y \) with the properties that \( P(Y) \) is defined, \( Y \geq X \) and \( F_X \neq F_Y \).

PROOF: There exist some constants \( l \) and \( i \) such that on input \( X \) \( P \) recursed \( l \) times before exiting by output function \( i \). Let the predicate \( P_i \) test variable \( x_j \) and let \( s_j \) be the recursion selector for this variable.

There are two cases, depending upon whether the output function \( f_i \) is constant or not. If \( f_i \) is not a con-
stant then since $X$ is bounded there must be a minimal $k > 1$ such that the predicate $p_i(s^k(x_j))$ is undefined.

By lemma 1 we can find an input $Z$ which causes $P$ to recurse $k$ times before exiting by output function $i$. Let $Y = X$ union $Z$. Since $Y > Z$ $P$ must recurse at least as much on $Y$ as it did on $Z$. Since the final point tested is still atomic $P(Y)$ will recurse $k$ times before exiting by output function $i$.

It is simple to verify the fact that $F_X \neq F_Y$.

The second case arises when $f_i$ is a constant function. By assumption 6 there is at least one output function which is not a constant function. Let $f_i$ be this function. Let the predicate $p_i$ test variable $x_j$. The same argument as before goes through with the exception that is may happen by chance the $P(Y) = P(X)$ (i.e. $P(Y)$ returns the constant value.) In this case we increment $k$ by 1 and perform the same process and it cannot happen that $P(Y) = P(X)$. 

**Lemma 3:** If $P$ touched a location $\alpha$, then we can construct two inputs $X$ and $Y$ such that $P(X)$ and $P(Y)$ are defined, and for any $P'$ in $\delta$, if $P(X) = P'(X)$ and $P(Y) = P'(Y)$ then $P'$ must touch $\alpha$.

**Proof:** Let $Z$ be the minimal $\alpha$ point. By lemma 2 we can construct an input $X$ such that $P(X)$ is defined, $X > Z$ and $F_X \neq F_Z$. Let $Y$ be $X$ pruned at $\alpha$. 390
We first assert that $P(Y)$ is defined and $F_Y = F_Z$. To see this we note that every point which was tested by $P$ is computing $P(Z)$ and found to be non atomic is also non atomic in $Y$. If is atomic in both, and if the output function was defined on $Z$ then it must be defined on $Y$ which is strictly larger.

Now suppose there existed some program $P'$ such that $P'(X)$ and $P'(Y)$ were computed correctly but $P'$ did not touch $d$. We see immediately that this cannot happen since all other positions are either the same in $X$ and in $Y$ or they exist in $X$ but not in $Y$. Hence if $P'(Y)$ is defined it would imply $F_X = F_Y$, a contradiction. \(\Delta\)

Define the positions which $P$ touches without going into recursion to be the primary positions of $P$.

Given a program $P$ to test our first task is then to construct a set of test inputs using theorem 1 which demonstrate that each of the primary positions must be touched.

Observe that this set contains at most $2n$ elements.

We will say a selector function $f$ factors a selector function $g$ if $g$ is equivalent to $f$ composed with itself some number of times. For example CADR factors CADADADR. We will say that $f$ is a simple factor of $g$ if $f$ factors $g$ and no function factors $f$, other than $f$ itself.
Let us denote by $\sigma_i$ $i=1,\ldots,m$ the simple factors of each of the $m$ recursion selectors. That is, for each $i$ there is a constant $l_i$ such that the recursion selector $s_i = \sigma_i^l_i$.

Let $q = \text{GCD}(l_i \; i=1,\ldots,m)$.

Let $S$ be the simultaneous recursion selector where the $i^{th}$ term is $\sigma_i^{l_i/q}$. Hence the recursion selectors of $P$ can be written as $S^q$.

We now construct a second set of data points in the following fashion:

For each selector variable $x_i$:

1) $x_i$ is an output variable used in output function $f_j$. Let $d$ be the position first tested by $p_j$ after $P(X)$ has recursed to a depth of at least $u^2$. Then we generate the minimal $d$ point.

2) $x_i$ is not an output variable, but is a predicate variable. Let $d$ be the first time a position with depth greater than $u^2$ is touched in $x_i$. First generate the minimal $d$ point, then using lemma 3 generate two inputs which demonstrate that position $d$ must be touched.

Notice that we have added no more than $3m$ points.

THEOREM 1: If $P'$ is in $\tilde{\Phi}$ and $P'$ computes correctly on all data points computed so far, then the recursion selectors of $P'$ must be powers of $\sigma_i$. 392
PROOF: Observe the fact that if $x_i$ is an output variable in $P$, it must appear as a result in at least one input $X$ in our test data space, hence if $P'(X)$ is correct $x_i$ must be an output variable for $P'$ also.

The proof of theorem 1 will then rest on the following two cases.

Case 1. If $x_i$ is an output variable. By construction there exists some $X$ in our test data space such that $P(X)$ recursively to a depth of at least $3u (<U^2)$ before exiting by the $j^{th}$ output function, where $x_i$ is an output variable in $f_j$.

Assume that the $i^{th}$ recursion selector in $P'$ is not a power of $\sigma_i$. Then somewhere before the $i^{th}$ variable has recursed to a depth of $u$ their paths must diverge.

Once the $i^{th}$ variable steps past the points where the paths in the two programs diverge it can never have access to the subtrees used in $P$ by $f_j$ in its output. Hence $P'$ on $X$ must halt before the $i^{th}$ variable has recursed to a depth of $u$.

But if that is the case then its output functions cannot access subtrees rooted any deeper then $2u$. By construction the correct output requires trees which can only be accessed by going at least $3u$ deep, hence a contradiction is obtained.

Case 2: If $x_i$ is not used as an output variable.
Assume the recursion selector of $x_i$ in $P'$ is not a power of $\sigma_i$. Then once the variables $x_i$ have recursed past the depth $u$ they will be in totally different subtrees of their input (see figure 3.)

By construction it is required that $P'$ touch a point whose depth is at least $3u$. $P'$ must therefore touch this point before the $i\text{th}$ variable diverges from the path taken by $P$, hence before it has reached a depth of $u$. But by definition $P'$ cannot touch any points deeper than $2u$ in this region, hence a contradiction is obtained. $\triangle$

Theorem 1 gives us a way to demonstrate that a program $Q$ must have the same recursion selectors, up to a power, as does $P$. We now wish to derive a slightly stronger result. We will show that there exists a constant $r$ such that the recursion selectors of $P'$ are exactly $S^r$.

Note that by definition we know that $|S^r|$ (that is, the maximum depth of any function in $S^r$) is less than $u$.

**Theorem 2**: If $P'$ is in $\delta$ computes correctly on all the points we have so far computed, then there exists a constant $r$ such that the recursion selectors of $P'$ are exactly $S^r$.

**Proof**: We know by theorem 2 that the recursion selectors of $P'$ must be powers of $\sigma_i$. For each $1 < i < m$
construct the ratio of the power of $\sigma_i$ in $P'$ to that of $P$. Let $x_i$ be the variable with the smallest such ratio and $x_j$ be the variable with the largest. From the fact that these ratios are different we will obtain a contradiction.

Case 1: $x_i$ is an output variable. By construction there is an input $X$ such that $P'(X)$ must recurse on $X$ to a depth of at least $u^2$ before outputting by a output function which uses $x_i$. This implies that $P'$ must recurse at least $u$ times. Since in comparison to the program $P$ the variable $x_j$ is gaining at least one level each recursion we have that either 1) $P'(X)$ is undefined because $x_j$ ran off the end of its input, or 2) $P'(X)$ must halt before it has recursed to a depth of $u(u-1)$ in $x_i$ in which case it cannot have produced the correct output.

The argument in the case where $x_i$ is a predicate variable, but not an output variable is almost the same and is hence omitted. △

By lemma 3 we know that if $P$ touches a location $d$, then we can construct a pair of inputs with the property that any program $P'$ in $\emptyset$ which executes correctly on these two inputs must also touch $d$. We now present the converse lemma.

**Lemma 4:** If $P$ works correctly on the test data so far constructed, and does not touch a location $d$, then we can
construct two inputs X and Y with the property that any P' in \( \emptyset \) which executes correctly on all this data must also not touch the position \( d \).

PROOF: Let \( x_1 \) be the variable containing \( d \). Let \( v \) the maximum depth any variable has obtained just after the \( i \)th recursion selector passes the depth of \( d \). Let \( X \) be a set of complete trees of depth \( v+2u \), pruned at \( d \).

There are two cases, depending upon whether \( P(X) \) is defined or not.

Case 1: \( P(X) \) is not defined. Assume \( P' \) touches \( d \). Let \( Z \) be the minimal \( d \) point in \( P' \) (we need not be able to construct this point.) We see that \( Z < X \). But this then implies that \( P'(X) \) must be defined, a contradiction.

Case 2: \( P(X) \) is defined. By lemma 1 we can construct an input \( Z > X \) so that \( F_X \not= F_Z \). Let \( Y \) be \( Z \) pruned at \( d \).

Assume \( P(X) = P'(X) \) and \( P(Y) = P'(Y) \) and \( P' \) touches \( d \). If \( P(Y) \) is undefined we are done, since \( P'(Y) \) must be defined. So assume \( P(Y) \) is defined. In this case, since \( P \) does not touch \( d \), \( F_Y = F_Z \not= F_X \). But if \( P' \) touched \( d \), then since \( x < Y \) we would have \( F_X = F_Y \), a contradiction. \( \triangle \)

Next we show that the primary positions of \( P' \) must be exactly those of \( P \).

Let \( p_1, ..., p_n \) be an ordering of the primary positions of \( P \) such that the depth of the position tested by \( p_1 \) is less then or equal to the depth of that tested by
\( \rho_{i+1} \)

We know the recursion selectors of \( P' \) are \( S^r \) where \( |S^r| < u \). This gives us at most \( u \) possibilities. For each possibility we proceed in turn as follows:

Assume position \( \rho_i \) (\( i = 1, \ldots, n \)) is not primary in \( P' \). We can construct a point which is then tested by \( P' \) earlier then \( \rho_i \) by imagining the root input was actually the result of one recursion, and then looking at the position \( \rho_i \) in relation to the earlier root (see figure 4.)

Now one of two cases arises. Either
1) the new position is not touched by \( P \), or
2) the new position corresponds to a position \( \rho_j \) \( j < i \).

In the first case we can construct two inputs which demonstrate the position in question must not be touched. The second case immediately rules out \( S^r \) as the recursion selector, since by induction \( \rho_j \) is primary to \( P \) and hence \( P' \) would not by an element of \( \emptyset \).

Notice we have increased our test case size by no more than \( 2nu \) elements. The resulting test case then gives us the following theorem.

**Theorem 3:** If \( P'(X) = P(X) \) for \( X \) in our test set, then the primary positions of \( P \) are exactly those of \( P' \).
Notice also that by the generic property that this also implies the following corollary:

THEOREM 4: The output functions of \( P' \) are exactly those of \( P \).

Once we have that the primary positions of \( P' \) are exactly those of \( P \), we can now return to the problem of showing that the selector functions of \( P' \) must be \( S^q \). Consider each of the alternative possibilities for \( S^P \) (no more then \( U \) of them.) Since the rates of recursion of \( P \) and \( P' \) differ, one of three cases must arise. Either

1) \( P' \) touches the same point twice (which means \( P' \) is not in \( \delta \) and is out of the running.)
2) \( P' \) touches a point which \( P \) fails to touch, or
3) \( P \) touches a point which \( P' \) fails to touch.

Since we only need to test for the last two conditions we need augment out test case with no more then \( 2u \) points.

we then have the following theorem:

THEOREM 5: The recursion selectors of \( P' \) must be exactly those of \( P \).

Pushing onward we next want to consider the recursion constructors. Once we have the other elements fixed, however, the constructors are almost given free. All we
need do is to construct p data points so that the $i^{th}$
data point causes the program $P$ to recurse once and exit
using an output function which uses the $i^{th}$ constructor
variable. By the generic property and the fact that the
entire $i^{th}$ constructor variable is then open to inspec-
tion we have the the next theorem.

THEOREM 6: The recursion constructors of $P'$ must be
exactly those of $P$.

What remains? Well the order in which the primary
positions are tested is the only thing we have not nailed
down. For each primary position $\xi$ add $X_\xi$ to our test
data. We leave it to the reader to verify:

THEOREM 7: The order of predicate evaluation in $P'$ is
exactly that of $P$.

Counting the size of our test set, we see now that
it contains no more then $3(n+m)+2(p+u+nu)$ points. Com-
bining all the theorems proved in this section we then
have our main result, which states:

THEOREM: Given a program $P$ in $\mathcal{G}$, there exists a set of no
more then $3(n+m)+2(p+u+nu)$ elements such that if $P'$ is
any program in $\mathcal{G}$ which computes the same results on this
set as $P$ does, then $P'$ must be equivalent to $P$.

COROLLARY: Either $P$ is correct or no program in $\mathcal{G}$
realizes the intended function.

5. AN EXAMPLE

The following example, taken from [6], will be used to illustrate some of the ideas here presented.

The program is given by [6] as follows:

(REVDBL
  (LAMBDA (ARG1)
   (COND
     ((NULL ARG1) NIL)
     (T (APPEND (REVDBL (CDR ARG1))
            (LIST (CAR ARG1) (CAR ARG1)))
   ))

We will translate it into the following form.

REVDBL(X,Y) = ATOM(X) -> Y
            Y <- CONS(CAR(X),CONS(CAR(X),Y)))
            X <- CDR(X)

Using the formula given in the main theorem, we see that a test set exists for this program containing no more than 20 points. However, if one follows the arguments given in this paper, one finds that actually the three points given in figure 5 suffice. This illustrates the point that we have actually been rather liberal in our counting, and usually a much smaller test set can be found than the limit stated in our main result.


$x = (a \ (b \ c) \ d)$

$\nu_j[x] =

\begin{align*}
&\begin{tikzpicture}[level distance=1.5cm, level 1/.style={sibling distance=3cm}, level 2/.style={sibling distance=2cm}, level 3/.style={sibling distance=1.5cm}]
&\node {a} [grow=down, anchor=north] child {node {b}}
&\node {c}
&\node {d}
&\node {NIL}
&\end{tikzpicture}
\end{align*}

$P_1, \ldots, P_8$ are Predicates
$e_1, \ldots, e_8$ are straight line programs

Figure 1

Figure 2

\begin{align*}
&\begin{tikzpicture}
&\node {\mathcal{U}} [grow=up, anchor=south] child {node {\sigma}}
&\node {operation not a power of $\sigma$}
&\node {original program}
&\end{tikzpicture}
\end{align*}

Figure 3

Figure 4
Figure 5