

ALGORITHMIC PROGRAM DIAGNOSIS

Ehud Y. Shapiro¹

Department of Computer Science
Yale University
New Haven, CT 06520

Abstract.

The notion of program correctness with respect to an interpretation is defined for a class of programming languages. Under this definition, if a program terminates with an incorrect output then it contains an incorrect procedure. Algorithms for detecting incorrect procedures are developed. These algorithms formalize what experienced programmers may know already.

A logic program implementation of these algorithms is described. Its performance suggests that the algorithms can be the backbone of debugging aids that go far beyond what is offered by current programming environments.

Applications of algorithmic debugging to automatic program construction are explored.

1. Introduction

Program debugging is composed of program diagnosis, the process of finding a bug, and program correction, the process of fixing the bug. In this paper we develop algorithms that mechanize program diagnosis. They apply in the following setting. A programmer is given a program P that returned on input x an incorrect output y . The goal of the programmer is to locate erroneous procedures in P and fix them. We assume the programmer knows what the program P is supposed to compute, either because she wrote it or because she is familiar with its documentation, and that she can effectively use this knowledge to correctly answer questions like "is y a correct output for procedure p on input x ?" and "what is a correct output for procedure p on input x ?", for every procedure p in P . The algorithms we develop can, by asking such questions, detect erroneous procedures in the program P , and provide the programmer with useful information as to how to correct them.

¹The author acknowledges the support of the National Science Foundation, grant No. MCS8002447.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

We give an informal characterization of the programming languages to which these algorithms apply. We assume that a program is a static collection of procedures. Each procedure has input variables and output variables (a function can be viewed as a procedure with one output variable), which take values from some domain D not containing the symbol \perp . A procedure with n input variables and m output variables computes a (possibly partial) function from D^n to finite subsets of D^m . We assume some procedure call mechanism, and that a procedure call can fail to return a defined output either by not terminating or by terminating with its output undefined. In the latter case its output is denoted by \perp . We require the output of a procedure call to be undefined if any of its inputs is undefined or any subordinate procedure call returned an undefined output. The programming language can be nondeterministic (with bounded nondeterminism), and in such a case a procedure can terminate with its output undefined only if all its possible computations terminate, and none of them returns a defined output.

We also assume that the control structure within procedures is rather simple, by requiring that a computation can fail to terminate only by having an infinite chain of procedure calls, but not by looping indefinitely inside one procedure.

"Purified" versions of many existing and proposed programming languages satisfy the above description. For example, pure Lisp, pure Prolog, loop-free Algol-like languages with no side-effects and a provision for enforcing monotonicity, procedural APL with no goto's, Backus' applicative languages, Harel's And/Or programs, etc..

We define semantics for programs. This definition is, in a sense, an abstraction of the semantics of logic programs as defined by Van Emden and Kowalski [6]. An interpretation of a procedure with a name p , n input variables and m output variables is a set of triples $\langle p, x, y \rangle$, where p is a procedure in P , x is in D^n and y is in D^m . We assume that for any p and x there are only finitely many y 's such that $\langle p, x, y \rangle$ is in M . An interpretation of a program is the union of the interpretations of its procedures.

We say that y is a *correct output* of a procedure call $\langle p, x \rangle$ in M if $\langle p, x, y \rangle$ is in M , or, in case $y = \perp$, if for no y' , $\langle p, x, y' \rangle$ is in M . Otherwise, y is said to be an *incorrect output* in M .

We define two types of oracles for M , which correspond to

the questions we have assumed the programmer is capable of answering about a program. A *ground oracle* for M is a device that, on input $\langle p, x, y \rangle$, outputs “yes” if $\langle p, x, y \rangle$ is in M and “no” otherwise. A *functional oracle* for M is a device that on input $\langle p, x \rangle$ nondeterministically outputs a y which is a correct output for $\langle p, x \rangle$ in M .

Definition 1.1: Let p be a procedure and M an interpretation. An *oracle simulation of p on x with M* is a computation of p on input x where every procedure call immediately subordinate to $\langle p, x \rangle$ is simulated by a call to a functional oracle for M .

An oracle simulation of a procedure call is simply a computation with one-level, correctness-guaranteed procedure invocation. The goal of an oracle simulation is to isolate the execution of a procedure call from possible errors in subordinate procedure calls. Note that an oracle simulation is a safeguard even from errors in recursive calls to the procedure being simulated.

We say that a procedure p is *correct in M* if, for any x , the output of any oracle simulation of p on x with M is correct in M , and that p is *incorrect in M* otherwise. If a procedure p is incorrect in M , then there is some x such that p has an oracle simulation on x that returns an incorrect output y . Such a simulation is called a *counterexample* to the correctness of p in M . Naturally, a program is said to be correct in M if all its procedures are correct in M .

Our notion of correctness is stronger than the standard notion of *partial correctness* [13]. A correct program is also partially correct, but the opposite is not always true. For example, any everywhere nonterminating program is partially correct, but the nonterminating program

$$f(x) \text{ -- if } 2x \text{ is an integer then } f(2x)$$

is incorrect in the interpretation $M = \{\langle f, x, 1 \rangle \mid x \text{ is an integer}\}$. The oracle simulation of f on 0.5 with M returns 1, which is an incorrect output in M . Another important difference between the two definitions is that partial correctness of a program can not, in general, be reduced to the partial correctness of its procedures. For example, a self-contained procedure p in a partially correct program can be modified in a way that preserves p 's partial correctness but not the partial correctness of the program as a whole. In contrast to this property of partial correctness, program correctness can be effectively reduced to procedure correctness, given a ground oracle for M , as Algorithm 1 below shows.

We say that a program P is *complete for M* if for any procedure p in P and any $\langle p, x, y \rangle$ in M there is a computation of p with input x that outputs y , and that P is *incomplete for M* otherwise. A program that is both partially correct and complete is sometimes called *totally correct*. Reducing the notion of program completeness to some property of its procedures is not as easy as for program correctness. We describe a partial solution to this problem.

We say that a procedure p is *sufficient for $\langle x, y \rangle$ with respect to M* if there is an oracle simulation of p on x that outputs y . If no such simulation exists we say that p is *insufficient for $\langle x, y \rangle$* . A procedure p is sufficient for M if for any $\langle p, x, y \rangle$ in M , p is sufficient for $\langle x, y \rangle$ with respect to M . A program is sufficient for M if all its procedures are sufficient for M .

The notion of sufficiency is a weak form of completeness. Clearly, if a program is complete for M then it is also sufficient for M . The opposite is not always true. For example, the program

$$f(x) \text{ -- if } x-1 \text{ is an integer then } f(x-1) \text{ else } 0$$

is sufficient for the interpretation $\{\langle f, x, y \rangle \mid x \text{ is an integer and } y=1 \text{ or } x \text{ is not an integer and } y=0\}$, but is incomplete for it.

Similar to program correctness, program sufficiency can be effectively reduced to procedure sufficiency, although here a functional oracle is needed. Algorithm 2 below demonstrates this.

Note that if we restrict ourselves to deterministic programming languages, the theoretical framework developed can be simplified. For example, \perp can be incorporated in the domain D , the notion of program sufficiency is then subsumed by program correctness, and Algorithms 1 and 2 below can be unified into one. We do not restrict ourselves to deterministic languages, however, since our intended application is the debugging of logic programs, which are nondeterministic.

We comment on the effectiveness of our definitions. An interpretation M is said to be *computable* if the mapping from $\langle p, x \rangle$ to the finite set $\{y \mid \langle p, x, y \rangle \text{ is in } M\}$ is computable. Clearly, if M is computable then ground queries and functional queries for M are computable. However, in such a case even a stronger claim holds.

Lemma 1.2: Let M be a computable interpretation.

Then the mapping from $\langle p, x \rangle$ to the finite set $\{y \mid \text{there is an oracle simulation of } p \text{ on } x \text{ with } M \text{ that returns } y\}$, is computable for any procedure p and input x .

To compute the above set one cycles through all possible oracle simulations of p on x and collects their results. By the assumption made above, any computation within a procedure terminates, and for any pair $\langle p, x \rangle$ there are only finitely many y 's such that $\langle p, x, y \rangle$ is in M . Hence there are only finitely many possible oracle simulations, and each of them terminates.

Correctness and sufficiency of programs are semi-decidable for computable interpretations, using the following procedure. Given a program P and a computable interpretation M , enumerate in some standard way all triples $\langle p, x, y \rangle$, where p is a procedure in P , x and y are vectors over D with the appropriate arities. For any such triple $\langle p, x, y \rangle$, check whether there is an oracle simulation of p on x that returns y . If such a simulation exists and the triple is not in M , then P is incorrect in M . If such a simulation does not exist and the triple is in M , then P is insufficient for M . In both cases output an appropriate message and terminate.

We relate the notions defined to fixpoint semantics. We associate with any procedure p a transformation τ_p from interpretations to interpretations. Let M be an interpretation and p a procedure. Then $\tau_p(M)$ is defined to be the set $\{\langle p, x, y \rangle \mid y \text{ is an output of an oracle simulation of } p \text{ on } x \text{ with } M\}$. The transformation τ_P associated with a program P is the union of the transformations associated with P 's procedures. The transformations defined are monotonic, by the assumptions made above concerning undefined values in procedure calls.

It is easy to see that a program P is correct in M iff $\tau_P(M) \subset M$, and that P is sufficient for M iff $M \subset \tau_P(M)$. It

follows that a program P is both correct and sufficient with respect to M iff M is a fixpoint of τ_P . Using an argument similar to the one in [6], it can be shown that P is both partially correct and complete with respect to M iff M is the least fixpoint of τ_P .

With these notions we can draw the limits of our approach: the debugging algorithms described below are applicable only when the intended interpretation M of a program P is *not* a fixpoint of the transformation τ_P .

2. Algorithms for Program Debugging

We describe two algorithms. Algorithm 1 is applicable when a program returns a defined but incorrect output. It can then detect an incorrect procedure in the program, using a simple debugging technique: trace the execution of the program; the first procedure to return a wrong output is wrong.

Algorithm 1: Tracing an incorrect procedure

Input: A procedure p in P and an input x such that p on x returns an output $y \neq \perp$ incorrect in M .

Output: A counterexample to the correctness in M of some procedure p' in P .

Algorithm: The algorithm simulates an execution of $\langle p, x \rangle$ that returns y . As a procedure call $\langle p', x \rangle$ returns with output y' , the algorithm calls the ground oracle with $\langle p', x', y' \rangle$. If the oracle answers “yes”, the algorithm proceeds with the simulation. If the oracle answers “no”, then p' is an incorrect procedure. The algorithm returns the counterexample to the correctness of p' implied by x' and y' and terminates. ■

We argue that Algorithm 1 is correct. Consider the ordered tree of procedure calls associated with the computation of $\langle p, x \rangle$ that returns y . The nodes of the tree are triples $\langle p, x, y \rangle$ for every invocation of the procedure p on input x that returned output y in the computation. The parent relation in the tree reflects the procedure invocation relation in the computations, and sons in the tree are ordered according to the order in which the procedures are invoked.

The order in which Algorithm 1 queries the oracle corresponds to the post-order traversal of the tree thus defined. Consider the first node $\langle p', x', y' \rangle$ of the tree in post-order, for which the ground oracle answers “no”. By the definition of Algorithm 1, all sons of this node were already tested and found correct, hence the procedure p is incorrect in M , and the subtree of depth 1 rooted at the node $\langle p', x', y' \rangle$ corresponds to a counterexample to the correctness of p in M . This establishes the following theorem.

Theorem 2.1: Let P be a program and M an interpretation. If a procedure p in P has a computation on input x that returns an output $y \neq \perp$ incorrect in M , then Algorithm 1 applied to p and x returns a counterexample to the correctness in M of some procedure p' in P .

Algorithm 2 below is applicable when a program terminates with its output incorrectly undefined. It can then detect an insufficient procedure in the program.

We explain the procedure ip using the notion of tree of procedure calls associated with a computation, defined above.

Algorithm 2: Tracing an insufficient procedure

Input: A procedure p in P and an input x such that p on x incorrectly returns \perp .

Output: A triple $\langle p', x', y' \rangle$ in M such that p' is a procedure in P which is insufficient for $\langle x', y' \rangle$ with respect to M .

Algorithm: The algorithm first calls a functional oracle for M with $\langle p, x \rangle$. Since \perp is an incorrect output, the oracle returns some $y \neq \perp$. It then calls a recursive procedure ip with input $\langle p, x, y \rangle$.

The procedure ip uses a functional oracle for M . It operates as follows. On input $\langle p, x, y \rangle$ it first nondeterministically tries to construct an oracle simulation of the call $\langle p, x \rangle$ that returns y . If it fails, then by definition p is insufficient for $\langle x, y \rangle$, and ip returns $\langle p, x, y \rangle$.

If the oracle simulation succeeds, ip iterates through all the oracle calls performed during the simulation. For any oracle call $\langle p', x' \rangle$ with output y' performed during the simulation, ip calls itself recursively with $\langle p', x', y' \rangle$. If any of ip 's recursive calls returns with output $\langle q, u, v \rangle$, then ip returns immediately with output $\langle q, u, v \rangle$. If either no oracle calls were done (the oracle simulation did not perform oracle calls), or all the recursive calls to ip returned “ok”, then ip returns with output “ok”. ■

Whenever ip is called with $\langle p, x, y \rangle$ it tries to construct such a tree, rooted at $\langle p, x, y \rangle$. When ip succeeds it returns *ok*. Since ip uses a functional oracle for M to construct the tree, once it succeeds, the correctness in M of the simulated computation is guaranteed. Lemma 2.2 summarizes this.

Lemma 2.2: Assume that the procedure ip in Algorithm 2 is called with input $\langle p, x, y \rangle$ in M . Then ip returns “ok” iff there is a computation of p on x that returns y in which all procedure calls return an output correct in M .

Algorithm 2 calls ip with a triple $\langle p, x, y \rangle$ that is in M . It is easy to see that triples in subsequent recursive calls of ip are also in M . Also, if at some point the procedure ip fails to construct an oracle simulation of p on x that returns y , then ip returns $\langle p, x, y \rangle$, and, by the definition of sufficiency, p is insufficient for $\langle x, y \rangle$. This establishes Lemma 2.3.

Lemma 2.3: Assume that the procedure ip in Algorithm 2 is called with input $\langle p, x, y \rangle$, which is in M . If ip returns a triple $\langle p', x', y' \rangle$, then $\langle p', x', y' \rangle$ is in M and p' is insufficient for $\langle x', y' \rangle$.

To establish the correctness of Algorithm 2, as stated in Theorem 2.4, note that ip is called initially with a triple $\langle p, x, y \rangle$ in M , and that ip cannot succeed in simulating a computation of p on x that returns y , since such a computation does not exist by the assumption that p on x originally returned \perp . Hence, by Lemma 2.2, ip cannot return *ok*. By Lemma 2.3, if ip returns a triple $\langle p', x', y' \rangle$ then this triple is in M , and p' is insufficient for $\langle x', y' \rangle$.

Theorem 2.4: Let P be a program and M an interpretation. If a procedure p in P on input x terminates and incorrectly returns \perp , then Algorithm 2 applied to $\langle p, x \rangle$ returns a triple $\langle p', x', y' \rangle$ in M such that p' is insufficient for $\langle x', y' \rangle$ with respect to M .

3. An Implementation for Logic Programs

We recall what logic programs are [10], relate the program semantics defined above to the standard semantics of logic programs [6], and develop logic programs that implement Algorithms 1 and 2.

A logic program is a collection of *definite clauses*, which are universally quantified logical sentences of the form

$$A \leftarrow B_1, B_2, \dots, B_k \quad k \geq 0$$

where the A and the B 's are atoms. Such a sentence is read “ A is implied by the conjunction of the B 's”, and is interpreted procedurally “to satisfy goal A , satisfy goals B_1, B_2, \dots, B_n ”. A is sometimes called the *procedure name* and the B 's the *procedure body*. If the B 's are missing, the sentence reads “ A is true” or “goal A is satisfied”. A sentence $\leftarrow B_1, B_2, \dots, B_k, k \geq 1$, is called a *negative clause*, and is read “the B 's are false”, or “satisfy the B 's”. Given a negative clause, a collection of definite clauses can be executed as a program, using this procedural interpretation.

As an example, a logic program that implements the quicksort algorithm is shown in Figure 1.

Figure 1: A logic program for quicksort

```

qsort([X/L], L0) ←
  partition(L, X, L1, L2),
  qsort(L1, L3), qsort(L2, L4),
  append(L3, [X/L4], L0).
qsort([], []).

partition([X/L], Y, L1, [X/L2]) ← X > Y, partition(L, Y, L1, L2).
partition([X/L], Y, [X/L1], L2) ← X ≤ Y, partition(L, Y, L1, L2).
partition([], X, [], []).

append([X/L1], L2, [X/L3]) ← append(L1, L2, L3).
append([], L, L).

```

We use upper-case strings as variable symbols, and lower-case strings for all other symbols. The term $[]$ denotes the empty list, and the term $[X/Y]$ stands for a list whose head (car) is X and tail (cdr) is Y . The results of unifying the term $[A/X]$ with the list $[1,2,3,4]$ is $A=1, X=[2,3,4]$, and unifying $[X/Y]$ with $[a]$ results in $X=a, Y=[]$.

The procedure $qsort(X, Y)$ computes the relation “ Y is the result of sorting the list X ”. Its first clause is read, procedurally: “to sort the list whose head is X and tail is L into a list $L0$, partition the list L according to X into lists $L1$ and $L2$, recursively sort $L1$ into $L3$ and $L2$ into $L4$, and append $L3$ to the list whose head is X and tail is $L4$ to get $L0$ ”. The procedure $partition(L, X, L1, L2)$ computes the relation “ $L1$ contains the elements of L which are less than X and $L2$ contains the elements of L which are greater than or equal to X ”. The procedure $append(L1, L2, L3)$ computes the relation “ $L3$ is the result of appending the list $L1$ to the List $L2$ ”.

We demonstrate how this program works on the Prolog-10 [15] with some examples. User input is in italics. Below are several possible ways of using *append*,

```
| ?- append([1,2,3],[4,5],X).
```

```
X = [1,2,3,4,5]
```

```
yes
| ?- append(X, [4,5], [1,2,3,4,5]).
```

```
X = [1,2,3]
```

```
yes
| ?- append([1,2,3],[4,5],[1,5]).
```

```
no
an example of using partition,
```

```
| ?- partition([4,1,5,8,2], 3, L1, L2).
```

```
L1 = [1,2],
L2 = [4,5,8]
```

```
yes
and a traced execution of qsort, where subordinate procedure calls are skipped. The first number associated with a procedure invocation is its sequential number, the second is its depth. Numbers preceded by “_” are names of internal Prolog variables.
```

```

| ?- trace, qsort([2,1,3],X).
(1) 0 Call : qsort([2,1,3],_55) ?
(2) 1 Call : partition([1,3],2,_137,_138) ? s
> (2) 1 Exit : partition([1,3],2,[1],[3])
(8) 1 Call : qsort([1],_139) ? s
> (8) 1 Exit : qsort([1],[1])
(13) 1 Call : qsort([3],_140) ? s
> (13) 1 Exit : qsort([3],[3])
(18) 1 Call : append([1],[2,3],_55) ? s
> (18) 1 Exit : append([1],[2,3],[1,2,3])
(1) 0 Exit : qsort([2,1,3],[1,2,3])

```

```
X = [1,2,3]
```

```
yes
```

We describe an interpreter for logic programs, written as a logic program. We assume that the conjunction B_1, B_2, \dots, B_n is represented as $(B_1, (B_2, (\dots, B_n), \dots))$, that the clause “ $P \leftarrow$ ” is represented as $P \leftarrow true$, and that a program P is represented as set of clauses $clause(C, P) \leftarrow$ for any clause C in P . The semantics of $execute(A, P)$ is “ A is provable from P ”

Figure 2: A logic program interpreter

```

execute(true, P).
execute((A, B), P) ← execute(A, P), execute(B, P).
execute(A, P) ← clause((A ← B), P), execute(B, P).

```

The reader should not be misled by the simplicity of this interpreter. This is executable code, and, modulo implementation-dependent syntactic conventions, it can be loaded and run as it is by a standard Prolog interpreter, e.g. the Prolog-10. To understand how it works, it might help to realize that the “real” computation is done in the call $clause((A \leftarrow B), P)$, in which the clause to be invoked is chosen, its head is unified with A (by the interpreter that executes this interpreter) and its body is instantiated with the unifying substitution, giving B . The logic programs below that implement Algorithms 1 and 2 are extensions to this interpreter.

We define the semantics of logic programs. Since in logic programs there is no explicit distinction between input variables and output variables, we rephrase the definition of an interpretation, and define a new kind of oracle. An *interpretation of a logic program* is a set of ground (variable-free) atoms. An *existential oracle* for an interpretation M is a device that, on input $B_1, B_2, \dots, B_n, n \geq 1$, nondeterministically returns a substitution θ such that $B_i\theta$ is in M for all $1 \leq i \leq n$, if such a θ exists, and answers “ \perp ” otherwise.

Substituting “existential oracle” for “functional oracle” in the definitions above, correctness of a logic program in M is simply its truth in M as a logical sentence, and a counterexample to procedure correctness in M is a ground instance of a clause which is false in M . Sufficiency also has a natural model-theoretic definition: a program is sufficient for a ground atom A if it has a clause $A' \leftarrow B_1, B_2, \dots, B_n, n \geq 0$, for which there is a substitution θ that unifies A and A' , and $B_i\theta$ is in M , for all $1 \leq i \leq n$. Program completeness is its completeness as a logical theory with respect to M , and the transformation associated with a program is identical to the transformation defined in [6]. It should be mentioned that procedure termination with undefined output, as defined for general programs, corresponds to *finite failure* of goals in logic programs [1]. We say that a goal $\leftarrow A_1, A_2, \dots, A_n, n \geq 1$, *immediately fails* in P if there is no clause $A' \leftarrow B'$ in P such that A' unifies with A_1 . A goal $\leftarrow A_1, A_2, \dots, A_n, n \geq 1$, *finitely fails* in a program P if all computations of P on $\leftarrow A_1, A_2, \dots, A_n$ are finite, and each computation contains at least one goal that immediately fails.

The following logic program implements Algorithm 1. It contains one procedure, $fp(A, P, CE)$ (read “false procedure A , P , CE ”), which computes the relation “ A is provable using only correct ground instances of clauses in P and $CE=ok$, or A is provable using some false ground instance $A' \leftarrow B'$ of a clause in P , and $CE=A' \leftarrow B'$ ”. The program is best understood by viewing its first two arguments as inputs and the third as output, although this is not implied by its definition. The program also contains a procedure call $test(A, P, V)$ to a ground oracle for M . The way fp is defined the atom A in the call $test(A, P, V)$ is not necessarily ground. We assume that in such a case the ground oracle instantiates it arbitrarily to some member of the Herbrand base of P before testing it.

Figure 3: A logic program for tracing a false procedure
 $fp(true, P, ok)$.

$$fp((A, B), P, (A' \leftarrow B')) \leftarrow fp(A, P, (A' \leftarrow B')).$$

$$fp((A, B), P, X) \leftarrow fp(A, P, ok), fp(B, P, X).$$

$$fp(A, P, (A' \leftarrow B')) \leftarrow clause((A \leftarrow B), P), fp(B, P, (A' \leftarrow B')).$$

$$fp(A, P, ok) \leftarrow clause((A \leftarrow B), P), fp(B, P, ok), test(A, P, "true").$$

$$fp(A, P, (A \leftarrow B)) \leftarrow clause((A \leftarrow B), P), fp(B, P, ok), test(A, P, "false").$$

The procedure fp contains six clauses. The first one is the base case and needs no explanation. The next two clauses deal with conjunctive goals, and correspond to the second clause of the interpreter described above. They return $(A' \leftarrow B')$ if the recursive call on any conjunct returns $(A' \leftarrow B')$, and return ok otherwise. The next three clauses deal with procedure

invocation, and they correspond to the third clause of the interpreter. They return $(A' \leftarrow B')$ if the recursive call on the body of the invoked clause returned $(A' \leftarrow B')$. Otherwise they return “ ok ” if the instantiated procedure head is tested and found true, and return the (ground instance) of the clause invoked if the result of the test is false.

The procedure fp as defined is not very efficient, and contains many unnecessarily repeated computations. Well known transformations (which may take the program outside the “pure” part of Prolog) can easily eliminate the problem, probably at the cost of the program’s readability. Appendix I contains a more efficient implementation.

The following implementation of Algorithm 2 contains one procedure, $ip(A, P, B)$ (read “insufficient procedure A , P , B ”), which computes the relation “ A is true and provable from P and $B=ok$, or A is true but not provable from P and B is a ground atom in M such that P is insufficient for B ”. The program also contains a procedure call $satisfiable(A)$ to an existential oracle for M , that succeeds by instantiating A to a conjunction of ground atoms true in M , and fails if no such instance exists. As should be expected according to the semantics of ip , its definition incorporates a notion of unprovability. The metalogical term $not(X)$ is defined to be true just in case X is not provable. It is a built in procedure in any standard Prolog implementation, and a precise semantics for it is given in [1, 4]. The term $atom(X)$ is introduced as a hack so we can detect easily whether the recursive call to ip returns an atom for which the program P is insufficient, or the constant “ ok ”.

Figure 4: A logic program for tracing insufficiency
 $ip(true, P, ok)$.

$$ip((A, B), P, atom(A')) \leftarrow ip(A, P, atom(A')).$$

$$ip((A, B), P, X) \leftarrow ip(A, P, ok), ip(B, P, X).$$

$$ip(A, P, X) \leftarrow clause((A \leftarrow B), P), satisfiable(B), ip(B, P, X).$$

$$ip(A, P, atom(A)) \leftarrow not((clause((A \leftarrow B), P), satisfiable(B))).$$

The first three clauses in the program ip are similar to the ones in the program fp . The fourth clause is doing an “oracle simulation” of the procedure call A . If the simulation succeeds, it calls itself recursively on the body of the procedure invoked, and returns the result of this call. The last clause applies when the oracle simulation fails. Then the program P is insufficient for A , and $atom(A)$ is returned.

It is possible that a more sophisticated implementation of the procedure ip , using techniques for selective backtracking in logic programs [14], may decrease the number of existential queries performed during its execution.

4. Suggested Applications

It seems that a straightforward implementation of Algorithms 1 and 2, in which the programmer is the acting oracle, would not constitute a significant improvement over existing debugging packages such as [3, 22]. Rather, we see the prospects of our approach in mechanizing the oracle queries as much as possible. This can be done in a large variety of ways.

One way to partly mechanize the oracle queries is to accumulate a database of answers to previous queries. A new

query is first posed to the database, and only if the database fails to answer it the programmer is asked, and his answer is then added to the database. As the debugging of a program progresses, the database contains more data about the program, and the debugging process becomes more automatic. Experience with such an implementation shows that if the same test-data is used consistently, the number of queries left unanswered by the database decreases rapidly.

Another approach for reducing the number of queries the programmer has to answer is by making “correctness assumptions” about certain procedures. This is similar to what programmers do when they restrict their attention to suspicious procedures by setting “break points” on them, and temporarily assuming that other procedures are correct. The result of a wrong correctness assumption is not fatal, since when the algorithm detects an incorrect procedure it provides a counterexample to its correctness, which contains all results of immediately subordinate procedure calls. These results can be examined carefully, and if one of them is found wrong it can then be traced after reversing some correctness assumptions. The same holds for sufficiency assumptions.

Another setting in which mechanizing the oracle queries can yield considerable gains can be termed *programming by stepwise optimization*. This well known idea is to let the programmer write a simple, lucid, but maybe an inefficient program, and then transform it to a more efficient, but maybe less comprehensible code. Mechanizing the transformation process is a major research goal in the field of automatic programming [2, 9, 11, 12]. Viewing it as a manual process, it is a reminiscent of Wirth’s programming by stepwise refinement [23], but with one important difference: a program in an intermediate stage of refinement can not be executed and debugged, while an unoptimized program can. This is one major obstacle for creating an environment that effectively supports programming by refinement, as reported in [17, 21].

Algorithmic program debugging can aid programming by stepwise optimization in the following way: the later, more efficient version of the program can be debugged with the debugging algorithms, using an earlier version of the program as an oracle. The information in the database from debugging the earlier programs can be used as a source for test-data. Clearly, the greater the overlap between the procedural structure of the programs, the more queries the earlier program can answer while debugging the later one.

Algorithmic debugging can also contribute to mechanizing the process of program optimization. One reason for the practical weakness of current program transformation techniques may be their insistence on using correctness-preserving transformations only. This requirement can be relaxed. Even if the result of a transformation is not always correct, the new program can be debugged with respect to the old one, using the debugging algorithms. The idea can be summarized with a slogan: “Don’t be compulsive about correct realization of your dreams”. We speculate that such a relaxation of requirements may enable the development of more powerful, but not always correct transformations.

This approach to programming seems most easy to implement for logic programs, since a logic program

specification is just a (very slow) logic program, and hence the process of program derivation from specifications and program optimization are the same [9]. Since even specifications may be wrong [7], there is a need to debug them. If the specifications are executable, as in logic programs, then the debugging algorithms can be applied.

Another possible application of algorithmic debugging is the debugging of the rule-base of an expert system, a task confronted so far only with heuristic approaches [5]. The two components in a standard architecture of an expert system are a rule-base and an inference mechanism. Rules are supplied by the expert, and their intent is to realize the expert’s knowledge of her domain of expertise. The system uses its inference mechanism in an attempt to reach the same conclusions the expert would. Algorithmic debugging allows the expert to debug the rules she proposed without fully understanding the inference procedure of the system, as the debugging algorithms simulate the inference procedure, and query the expert only for “declarative” information.

Expert systems sometimes incorporate a certainty specification and evaluation mechanism, which enables the expert to qualify the rules she purposes, and the system to qualify its conclusions from these rules. Such a mechanism can be implemented in logic programs with uncertainties [18], and since logic programs with uncertainties have semantics, algorithmic debugging is applicable.

Most of our experience with the debugging algorithms was obtained, however, not using them as a stand-alone program, but as a component of the Model Inference system. The Model Inference system is a Prolog program that inductively infers logic programs from examples and non-examples of their input-output behavior. The system implements the general inductive inference algorithm described in [19, 20], specialized to infer logic programs, and a detailed account of its performance is provided there.

The system starts with an empty program, and “debugs its way” to a correct and sufficient program for the given interpretation. Algorithms 1 and 2 are used by the system to detect incorrect and insufficient procedures in the currently conjectured program. Once such a procedure is detected, it is modified by the system, using the additional information supplied by the algorithms (a counterexample to procedure correctness; an input-output relation for which the procedure is insufficient). The system can identify in the limit [8] various syntactic classes of logic programs; the particular class depends on a parameter that can be tuned.

Algorithm 1 specialized to logic programs is an instance of the Contradiction Backtracing algorithm [19, 20], which is an essential part of the general inductive inference algorithm. Algorithm 2 is not part of the general inference algorithm, but incorporating it in the Model Inference system greatly improves its performance.

5. An Example: Debugging a Quicksort Program

The implementation of Algorithms 1 and 2 which we demonstrate is based on the programs *fp* and *ip* described above, and a complete listing of it is provided in appendix II. We demonstrate its performance in debugging a faulty quicksort program. We start by typing into the Prolog interpreter a (hopelessly?) buggy program for quicksort. A reader with a keen eye can find six differences between this program and the correct program in Figure 1.

```

?- [user].
qusort([X/L],L0) ←
  partition(L,X,L1,L2),
  qusort(L1,L3), qusort(L2,L4),
  append([X/L3],L4,L0).

partition([X/L],Y,L1,[X/L2]) ←
  partition(L,Y,L1,L2).
partition([X/L],Y,[X/L1],L2) ←
  Y ≤ X, partition(L,Y,L1,L2).
partition([],X,[],[]).

append([X/L1],L2,[X/L3]) ← append(L1,L2,L2).
append([],L,[]).

```

user consulted 152 words 1.12 sec.

We try the program,

```

?- qusort([2,1],X).

```

no

and it finitely fails, although it should have succeeded with $X = [1,2]$. Recall that in such a case the program *ip* can be applied, detecting an insufficient procedure in the program being debugged. We invoke *ip* on $qusort([2,1],X)$. In the course of its execution, *ip* queries the user for satisfiability of atoms in the model. If the user answers “true”, she is required to supply the satisfying instance. The result of the queries are accumulated in Prolog’s internal database, and are used to answer future instances of the same queries.

```

?- ip(qusort([2,1],X),A).
Query: qusort([2,1],_47)?
|: true.
Value of _47?
|: [1,2].

```

```

Query: partition([1],2,_770,_771)?
|: true.
Value of _770?
|: [1].
Value of _771?
|: [].

```

```

Query: qusort([1],_772)?
|: true.
Value of _772?
|: [1].

```

```

Query: qusort([],_773)?
|: true.
Value of _773?
|: [].

```

```

Query: append([2,1],[1,2])?
|: false.

```

```

A = atom(qusort([2,1],[1,2])),
X = [1,2]

```

yes

We examine why the clause for *qusort* is insufficient for the atom $qusort([2,1],[1,2])$, and find that this is because the atom $append([2,1],[1,2])$ is false. The problem is that we added the number X partitioned upon to the head of the $L3$, the list of the smaller numbers, instead of between $L3$ and $L4$, the list of the larger numbers. So we fix that,

```

qusort([X/L],L0) ←
  partition(L,X,L1,L2),
  qusort(L1,L3), qusort(L2,L4),
  append(L3,[X/L4],L0).

```

and try again. Note the top-down debugging style of the procedure *ip*: among the six bugs in the program, the one in the top-level procedure is discovered first.

```

?- qusort([2,1],X).

```

no

```

?- ip(qusort([2,1],X),A).
Query: append([1],[2],[1,2])?
|: true.
Query: ≤(2,1)?
|: false.

```

```

A = atom(partition([1],2,[1],[])),
X = [1,2]

```

yes

We find that the atom $partition([1],2,[1],[])$ has no sufficient procedure. The second clause of *partition* should have taken care of it. We examine it more closely, and discover that the arguments of the \leq test are switched. We correct this

```

partition([X/L],Y,[X/L1],L2) ←
  X ≤ Y, partition(L,Y,L1,L2).

```

try again,

```

?- qusort([2,1],X).

```

no

```

?- ip(qusort([2,1],X),A).
Query: ≤(1,2)?
|: true.
Query: partition([],2,[],[])?
|: true.

```

```

Query: partition([],1,_1441,_1442)?
|: true.
Value of _1441?
|: [].
Value of _1442?
|: [].

```

Query: `append([], [1], [1])?`
`! : true.`

`A = atom(qsort([], [])),`
`X = [1, 2]`

yes

and find that the base case `qsort([], [])` is missing. We add it, and try again.

`! ?- qsort([2, 1], X).`

`X = []`

yes

This time the call succeeds, but with a wrong answer. So we apply the procedure `fp`, that will detect a false clause in the program.

`! ?- fp(qsort([2, 1], X), CE).`
 Query: `partition([1], 2, [], [1])?`
`! : false.`

`X = _47,`
`CE = (partition([1], 2, [], [1]) ← partition([], 2, [], []))`

yes

A counterexample to the first clause of `partition` was found. The `>` test is missing. We fix this bug,

`! partition([X/L], Y, L1, [X/L2]) ←`
`! X > Y, partition(L, Y, L1, L2).`

and try `qsort` again.

`! ?- qsort([2, 1], X).`

`X = []`

yes

`! ?- fp(qsort([2, 1], X), CE).`
 Query: `append([], [1], [1])?`
`! : false.`

`X = _47,`
`CE = (append([], [1], [1]) ← true)`

yes

And the fact that the base clause for `append` is wrong is detected. Note that only one query was needed. We fix the base clause to be `append([], L, L)`, and try again.

`! ?- qsort([2, 1], X).`

`X = [1|_258]`

yes

The answer we got is too general. It has the correct answer, `X = [1, 2]` as an instance, but is also has wrong answers as an instance. This means that the program still contains a false clause. We choose the false instance `X = [1]` of the answer and call `fp` with it.

`! ?- fp(qsort([2, 1], [1]), CE).`
 Query: `append([], [2], [2])?`
`! : true.`
 Query: `append([1], [2], [1])?`
`! : false.`

`CE = (append([1], [2], [1]) ← append([], [2], [2]))`

yes

And a counterexample to the main clause of `append` is discovered. Note the bottom-up debugging style of the procedure `fp`. The base clause for `append` was found false before the main clause. Also, note that by now the database has almost all the information needed to monitor the execution of `qsort([2, 1], X)` without querying the user. For example, in the course of this execution of `fp`, the database answered eight out of the ten queries performed. We fix the bug in `append`,

`! append([X/L1], L2, [X/L3]) ← append(L1, L2, L3).`

try again.

`! ?- qsort([2, 1], X).`

`X = [1, 2];`

no

`! ?- qsort([2, 1, 3], X).`

`X = [1, 2, 3];`

no

`! ?- qsort([2, 1, 3, 55, 3, 1414, 43, 65, 6, 44], X).`

`X = [1, 2, 3, 3, 6, 43, 44, 55, 65, 1414];`

no

and it works.

We summarize the main points exemplified in this session:

1. Any logic program that returns an incorrect result can be debugged, no matter how “buggy” it is.
2. There is no need to finish debugging “low level” procedures before one can debug “higher level” procedures, or vice-versa. A program can be debugged as a whole. The procedure `fp` detects bugs in a top-down order, and `fp` in a bottom-up order.
3. The number of queries the programmer needs to answer is small, and it decreases as the debugging progresses. For example, to detect the six bugs in the quicksort program, the programmer had to answer 15 queries, with distribution per bug of 5-2-4-1-1-2. The total number of queries performed during the debugging was 40, with distribution of 5-6-11-3-6-10.
4. Counterexamples to procedure correctness and examples to procedure insufficiency are useful clues as to how to fix the wrong procedure.

6. Conclusions

We have developed a theoretical framework for algorithmic program debugging, and shown that a practical implementation of it is within the reach of current programming technology. We suggested that the debugging algorithms developed can improve current debugging facilities, support automatic program

derivation from specifications, and be the basis for program inference from input-output behavior.

The major theoretical limitation of the debugging algorithms is that they are applicable only when the incorrect computation terminates. We do not yet know how to treat non-terminating computations.

The major practical limitation of the algorithms is that they apply only to programs with no side-effects. This is prohibitive considering current programming practices. One may argue, however, that developing debugging algorithms for programs with side-effects is not a promising approach. A better one may be to develop enough computerized support for “pure” programming, so that one who refrains from “impure” programming techniques not only will be saved on Judgement Day, but also be more productive as a programmer.

We would like to reflect on the choice of logic programs both as the target language and the implementation language of the debugging algorithms. It is easy to specialize the general algorithms for logic programs because their semantics is simple. It is easy to implement them in a logic program² because it is simple to write a logic program interpreter in Prolog. We believe that these two properties, in addition to the simple syntax of logic programs, make Prolog an ideal language for developing programming aids and for automatic programming.

We can apply these arguments to any programming language L . If the syntax of L is simple then it is easy for programs to manipulate L -programs. If the semantics of L is simple, it is easy for programs to reason about L -programs. For many obvious reasons, it is desirable that the implementation language for such algorithms be L itself. This can be accomplished only if L can interpret L -programs in a natural way, as suggested by Sandewall [16]. Having this in mind, it is clear that extensively hard-wiring arbitrary features into a programming language is the wrong direction to pursue.

Acknowledgments

I am thankful to Dana Angluin for comments and suggestions that greatly improved the correctness of this work, and to Gregory Sullivan for pointing to me that Algorithms 1 and 2 can be unified for deterministic programs. Luis Pereira suggested the applicability of selective backtracking to reduce the number of existential queries in the logic program implementation of Algorithm 2.

²For a reader not familiar with the relationship between logic programming and Prolog, one may say that logic programming is to Prolog what the lambda calculus is to Lisp.

Appendices

I. The Debugging System

The following is a complete listing of the debugging system demonstrated in section 3. It is written for the Edinburgh Prolog-10 [15]. It is different from the logic programs described in section 3 in the following aspects:

1. There is no explicit mentioning of the program being debugged. The whole internal database of clauses is considered as the program.
2. The procedures fp and ip are optimized, to prevent unnecessarily repeated computations.
3. The procedures fp and ip include provision to allow the program being debugged to contain compiled or built-in procedures. The execution of such procedures is not being simulated. This has the effect of making “correctness-assumptions” about compiled procedures, as discussed in section 4.
4. The procedure $test$ asks the user to instantiate the atom being tested, and assumes that it has only one true instance. (This is close to assuming the functionality of the program, but not quite the same).

```

fp(true,ok) ← !.
fp((A,B),X) ← !,
    fp(A,X1),
    (X1=(←←), X=X1 ;
     X1=ok, fp(B,X)).
fp(A,X) ←
    clause(A,B),
    fp(B,X1),
    (X1=(←←), X=X1 ;
     X1=ok,
     (test(A,V),
      (V=true, X=ok ; V=false, X=(A←B))
     )).
fp(A,ok) ← not(clause(A,B)), A.

ip(P,A) ← test(P,true), ip1(P,A).

ip1(true,ok) ← !.
ip1((A,B),X) ← !,
    ip1(A,X1),
    (X1=atom(←), X=X1 ; X1=ok, ip1(B,X)).
ip1(A,X) ← clause(A,B), satisfiable(B), ip1(B,X), !.
ip1(A,ok) ← not(clause(A,B)), A, !.
ip1(A,atom(A)).

test(P,V) ←
    recorded(fact,(P,true),←), !,
    V=true.
test(P,V) ←
    numbervars(P,0,←), % instantiates P with Skolem constants
    recorded(fact,(P,false),←), !,
    V=false.
test(P,V) ←
    write('Query: '),
    ask_for(P,V1),
    (V1=true, ask_for_var(P) ; V1=false),
    recordz(fact,(P,V1),←), !,
    V=V1.

```

satisfiable(true) ← !.
satisfiable((P,Q)) ← !, *test(P,true)*, !, *satisfiable(Q)*.
satisfiable(P) ← *test(P,true)*.

ask_for_var(X) ←
var(X), !, *write('Value of ')*, *ask_for(X,X)*, !.
ask_for_var(P) ← *P* = .[*F/A*], *ask_for_var1(A)*.

ask_for_var1([X/LJ]) ←
ask_for_var(X), *ask_for_var1(L)*.
ask_for_var1([]).

ask_for(Request, Answer) ←
repeat,
display(Request), *display('?)*, *nl*,
read(Answer).

References

- [1] K. R. Apt and M. H. van Emden.
Contributions to the Theory of Logic Programming.
 Technical Report CS-80-12, Department of Computer
 Science, University of Waterloo, February, 1980.
- [2] R. M. Burstall and J. Darlington.
 A Transformation System for Developing Recursive
 Programs.
JACM 24(1):44-67, January, 1977.
- [3] Lawrence Byrd.
 Prolog Debugging Facilities.
 Technical note, Department of Artificial Intelligence,
 Edinburgh University.
 1980.
- [4] Keith L. Clark.
 Negation as Failure.
 In H. Gallaire and J. Minker, editor, *Logic and Data
 Bases*, Plenum, 1978.
- [5] Randall Davis.
*Applications of Meta Level Knowledge to the
 Construction, Maintenance and Use of Large
 Knowledge Bases*.
 Technical Report STAN-CS-76-552, Computer Science
 Department, Stanford University, July, 1976.
- [6] M. H. van Emden and R. A. Kowalski.
 The Semantics of Predicate Logic as a Programming
 Language.
JACM 23:733-742, October, 1976.
- [7] Susan L. Gerhart and Lawrence Yelowitz.
 Observations of Fallibility in Applications of Modern
 Programming Methodologies.
IEEE Transactions on Software Engineering
 SE-2:195-207, September, 1976.
- [8] E. M. Gold.
 Language identification in the limit.
Information and Control 10:447-474, 1967.
- [9] C. J. Hogger.
 Derivation of Logic Programs.
JACM 27:372-392, April, 1981.

- [10] Robert A. Kowalski.
 Predicate Logic as a Programming Language.
 In *Proceedings of the IFIP Congress*, pages 569-574.
 IFIP, Amsterdam, 1974.
- [11] Zohar Manna and Richard Waldinger.
*Artificial Intelligence Series. : Studies in Automatic
 Programming Logic*.
 North-Holland, 1977.
- [12] Zohar Manna and Richard Waldinger.
 Synthesis: Dreams → Programs.
IEEE Transactions on Software Engineering
 SE-5:294-328, July, 1979.
- [13] Zohar Manna.
Mathematical Theory of Computation.
 McGraw-Hill, 1974.
- [14] L. M. Pereira and A. Porto.
 Selective Backtracking for Logic Programs.
 In W. Bibel and R. Kowalski, editor, *Fifth Conference on
 Automated Deduction*, pages 306-317. Springer.
 1980.
- [15] L. Pereira, F. Pereira and D. Warren.
User's Guide to DECsystem-10 PROLOG.
 Technical Report 03/13/5570, Laboratório Nacional De
 Engenharia Civil, Lisbon, September, 1978.
 Provisional version.
- [16] Erik Sandewall.
 Programming in an Interactive Environment: The LISP
 Experience.
Computing Surveys, March, 1978.
- [17] E. Shapiro, G. Collins, L. Johnson, J. Ruttenberg.
 PASES: A Programming Environment for Pascal.
SIGPLAN Notices, August, 1981.
- [18] Ehud Y. Shapiro.
 Execution and Debugging of Logic Programs With
 Uncertainties.
 In preparation.
 1981.
- [19] Ehud Y. Shapiro.
Inductive Inference of Theories from Facts.
 Technical Report 192, Yale University, Department of
 Computer Science, February, 1981.
- [20] Ehud Y. Shapiro.
 An Algorithm that Infers Theories from Facts.
 In *IJCAI-81, No. 7*. IJCAI, August, 1981.
- [21] T. Teitelbaum and T. Reps.
*The Cornell Synthesizer: a Syntax-Directed
 Programming Environment*.
 Technical Report TR79-381, Department of Computer
 Science, Cornell University, July, 1979.
- [22] Warren Teitelman.
INTERLISP Reference Manual.
 Technical Report, Xerox Palo Alto Research Center,
 September, 1978.
- [23] Niklaus Wirth.
 Program Development by Stepwise Refinement.
CACM 14:221-227, April, 1971.

List of Algorithms

Algorithm 1: <i>Tracing an incorrect procedure</i>	5
Algorithm 2: <i>Tracing an insufficient procedure</i>	5

List of Figures

Figure 1: <i>A logic program for quicksort</i>	7
Figure 2: <i>A logic program interpreter</i>	8
Figure 3: <i>A logic program for tracing a false procedure</i>	9
Figure 4: <i>A logic program for tracing insufficiency</i>	10