This research was supported by the National Science Foundation, grant No. MCS8002447

# Algorithmic Program Debugging

May, 1982

Research Report 237

Ehud Y. Shapiro

Author's current address: Department of Applied Mathematics, Weizmann Institute of Science, Rehovot 76000, ISRAEL.

This document reproduces a dissertation presented to the Faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy, on May, 1982.

Apart from some stylistic changes and the correction of typographical errors, the only difference in content is in Sections 2.1.3 and 4.5. To these sections we added a description, without proof, of results obtained in [82, 86].

*To Karl R. Popper*
*for his inspiring intellectual courage and clarity*

*and*

*to my parents, Shimon and Miriam*
*for bringing me up knowing that*
*the world is a great place to be*

## ABSTRACT

## Algorithmic Program Debugging

Ehud Y. Shapiro

Yale University, 1982

In this thesis we lay a theoretical framework for program debugging, with the goal of partly mechanizing this activity. In particular, we formalize and develop algorithmic solutions to the following two questions:

1. *How do we identify a bug in a program that behaves incorrectly?*
2. *How do we fix a bug, once one is identified?*

We develop interactive diagnosis algorithms that identify a bug in a program that behaves incorrectly, and implement them in Prolog for the diagnosis of Prolog programs. Their performance suggests that they can be the backbone of debugging aids that go far beyond what is offered by current programming environments.

We develop an inductive inference algorithm that synthesizes logic programs from examples of their behavior. The algorithm incorporates the diagnosis algorithms as a component. It is incremental, and progresses by debugging a program with respect to the examples. The Model Inference System is a Prolog implementation of the algorithm. Its range of applications and efficiency is comparable to existing systems for program synthesis from examples and grammatical inference.

We develop an algorithm that can fix a bug that has been identified, and integrate it with the diagnosis algorithms to form an interactive debugging system. By restricting the class of bugs we attempt to correct, the system can debug programs that are too complex for the Model Inference System to synthesize.

## Acknowledgements

First and foremost, this thesis owes its shape and content to my advisor, Dana Angluin. She has directed my research since I came to Yale, and have had a great influence on what I have accomplished in these years, and what I know today. Her thoroughness and sincerity in doing this will be a model for me in the years to come. She also tried to teach me her high professional and personal standards; although I resisted as strongly as I could, there are still some traces of them in this thesis.

Drew McDermott escorted my first steps in Prolog, and has remained skeptical ever since. He gave me some insightful suggestions, which made me wish he had given me lots more. Just to name a few, the definition of incremental inductive inference, the idea of mechanizing oracle queries, and the methods that led to the algorithm that diagnoses finite failure, were all originated in some off-hand comments he made when he failed to avoid me in the corridor.

Alan Perlis also contributed to the final form of the thesis, especially to its introduction and conclusions. Without his encouragement, I would not have had the nerve to make some of the bold claims I have made, and his flamboyant manner helped to counter-balance Dana's caution and serenity.

Mike Fischer did not really know whether he should read my thesis wearing his Hacker's Hat or his Theoretician's Hat. His deliberations continued almost past the deadline, but I think that now he knows that logic is not only good to do theory with, but also fun to hack.

I got a lot of support, and a chance to see the world, from the people of the logic programming community. For the past year or so we have been saying: "so long until the International Logic Programming Conference in Marseille", but every three months someone in another corner of the world gets interested in logic programming, and we meet again. During those workshops I have enjoyed being with, and learned a lot from, Lawrence Byrd, Ken Bowen, Alan Colmerauer, Maarten van Emden, Bob Kowalski, Frank McCabe, Fernando Pereira, and David Warren, to name a few.

David Warren's Prolog implementation was an indispensible research tool. First across the ocean, and then across the Arpanet, he reminded me that I am not the only

Prolog hacker in the world, even though that's how it feels at Yale.

Donald Michie's enthusiasm about my research was an ultimate source of pleasure, and helped me feel that what I am doing may be worth while. Together with Alan Perlis, he showed me that intellectual curiosity and openness are ageless and statusless.

Bob Nix kept me informed on what is going on in the world, while I was too preoccupied with my work to pay attention to anything else. I think I have learnt more about computer science from discussions with him than from all of the courses I have taken together. I wish I could take him with me as an office mate wherever I go. He also devoted a lot of his time to reading and correcting my thesis, which compensated for some of my ignorance of English and Combinatorics.

Many people made specific technical contributions to the thesis, which I would like to acknowledge: Frank McCabe was the first to suggest that the query complexity of the diagnosis algorithms may be improved; only after this realization, did I begin to believe that they can be more than an exercise in formalizing some vague intuitions, but a real debugging tool. He also found how to diagnose Prolog programs with negation.

Ryszard Michalski insisted on understanding what I was doing, which made things clearer to me too; he offered some terminological improvements that made the formal machinery I used more intuitive and appealing.

Bob Moore debugged my debugging algorithm while I was at SRI, and refuted my naive belief that a binary search technique could be readily used in the diagnosis of nontermination.

David Plaisted suggested an improvement to the algorithm that diagnoses termination with incorrect output, which I accepted without hesitation. He then refuted his improvement, but finally we made it to together to the current divide-and-query algorithm.

Last but not least, the Tools group at Yale, and in particular John Ellis, Steve Wood, and Nat Mishkin, made the use of our DEC-20 and its surrounding machinery tolerable, and even fun. I cannot imagine what my productivity as a programmer and as a writer would have been without Z, SM, and the kind help they provided me. They have also contributed to the literary aspect of the thesis, by telling me (or, more accurately, by allowing me to pick and put) the Zen parable about the drawing of the fish.

In addition to the individuals mentioned above, I wish to thank the National Science Foundation, whose grant, numbered MCS8002447, supported me during the last two years.

## Table of Contents

## List of Algorithms

## List of Programs

## List of Figures

# Chapter 1

# INTRODUCTION

## 1.1 The problem

It is evident that a computer can neither construct nor debug a program without being told, in one way or another, what problem the program is supposed to solve, and some constraints on how to solve it. No matter what language we use to convey this information, we are bound to make mistakes. Not because we are sloppy and undisciplined, as advocates of some program development methodologies may say, but because of a much more fundamental reason: we cannot know, at any finite point in time, all the consequences of our current assumptions. A program is a collection of assumptions, which can be arbitrarily complex; its behavior is a consequence of these assumptions; therefore we cannot, in general, anticipate all the possible behaviors of a given program. This principle manifests itself in the numerous undecidability results, that cover most interesting aspects of program behavior for any nontrivial programming system [79].

It follows from this argument that the problem of program debugging is present in any programming or specification language used to communicate with the computer, and hence should be solved at an abstract level. In this thesis we lay theoretical foundations for program debugging, with the goal of partly mechanizing this activity. In particular, we attempt

to formalize and develop algorithmic solutions to the following two questions:

1. *How do we identify a bug in a program that behaves incorrectly?*
2. *How do we fix a bug, once one is identified?*

An algorithm that solves the first problem is called a *diagnosis algorithm*, and one that solves the second a *bug-correction algorithm.*

To debug an incorrect program one needs to know the expected behavior of the target program. Therefore we assume the existence of an agent, typically the programmer, who knows the target program and may answer queries concerning its behavior. The algorithms we develop are interactive, as they rely on the availability of answers to such queries.

A diagnosis algorithm and a bug-correction algorithm can be integrated into a *debugging algorithm*, following the scheme in Figure 1 below. A debugging algorithm accepts as input a program to be debugged and a list of input/output samples, which partly define the behavior of the target program. It executes the program on the inputs of the samples; whenever the program is found to return an incorrect output, it identifies a bug in it using a diagnosis algorithm, and fixes it with the correction algorithm. Note that an algorithm for program synthesis from examples can be obtained from a debugging algorithm by fixing the initial program to be the empty one.

```
read P, the program to be debugged.
repeat
    read the next input/output sample.
    while P is found to behave incorrectly on some input do
        identify a bug in P using a diagnosis algorithm;
        fix the bug using a correction algorithm.
    output P.
until no samples left to read. []
```

**Figure 1:** A scheme for a debugging algorithm

## 1.2 Results

The main result of the thesis is a theoretical framework for program debugging. We describe a computational model, which is an abstraction of some common functional programming languages, including Prolog. Within this model a program can have three types of errors: termination with incorrect output; termination with missing output; and nontermination. We develop diagnosis algorithms that can isolate an erroneous procedure, given a program and an input on which it behaves incorrectly.

These algorithms are interactive: they query the programmer for the correctness of intermediate results of procedure calls, and use these queries to diagnose the error. Here are some examples of queries these algorithm pose during the diagnosis process:

- Is [*a*,*b*,*c*] a correct output for ***append*** on input [*a*,*b*] and [*c*,*d*]?
- What are the correct outputs of ***partition*** on input [2,5,3,1] and 4?
- Is it legal for ***sort***, on input [1,2,3], to call itself recursively with input [1,1,2,3]?

Queries of the first kind are posed by the algorithm that diagnoses termination with incorrect output; the second by the algorithm that diagnoses termination with missing output; and the third by the algorithm that diagnoses nontermination.

Our goal in developing the diagnosis algorithms was to minimize the amount of information the programmer needs to supply for them to diagnose the error. Typically, the computational overhead that is involved in optimizing the number of queries needed is acceptable. For example, we develop a diagnosis algorithm for the first error — termination with incorrect output — that is query optimal: the number of queries it performs is on the order of the logarithm of the number of procedure calls in the faulty computation. The query-complexity of the algorithm is optimal, and its computational complexity is linear in that of the faulty computation.

Even as a stand-alone diagnosis aid these algorithms provide a significant improvement over standard tracing packages, considering the

amount of human effort they require to diagnose a bug. But they also open the way for a wide spectrum of methods for automated debugging, all based on mechanizing the process of answering queries they pose. We list several such methods that can be used to partially mechanize diagnosis queries:

- Accumulate a database of answers to previous queries. Use them to answer repeated queries.
- Supply assertions and constraints on the input/output behavior, invoke the diagnosis algorithms whenever they are violated, and use them to answer diagnosis queries whenever they apply.
- Use a previous version of the program that is known to work to answer queries, when debugging the modified program on inputs on which its output is expected to remain unchanged (e.g., when the modification is only an optimization).

For example, the constraint that the number of elements in the output list of ***append*** should be the sum of the number of elements in its inputs lists can be used to answer negatively the first query in the example above. Knowing that the size of the input list to ***sort*** should decrease as the sorting progresses can be used to answer the third query in the negative. An older version of ***partition*** can be used to supply the desired outputs in the second query.

We think this approach is flexible enough to support the process of program development in all its stages: in developing prototypical, throwaway systems, one's concern is not reliability, but fast turnout. For fast changing programs answers to previous queries are worthless, and the overhead of maintaining the consistency of declarative information and documentation with what the program currently does is not cost-effective. For such systems the diagnosis algorithms provide a low-overhead development tool.

We expect that production systems, on the other hand, will exploit the full spectrum of aids the diagnosis algorithms provide. A database of answers to diagnosis queries is an invaluable source of test data, and can function as on-line documentation of the program. Debugging a new release of a system with the current working version answering diagnosis queries may also result in significant savings in human labor. The declarations of strongly-typed languages, although not obligatory, are not excluded from this framework as well: type and other declarations can be verified at runtime, when in debugging mode. Declarations that are checked at runtime can be more flexible and expressive than those designed specifically for verification by static analysis.

Our second group of results is concerned with the problem of inductive inference and program synthesis from examples. It turns out that the problems of inductively inferring theories and synthesizing programs from their input/output behavior can be addressed in much the same way as program debugging. The basic cycle of debugging — propose an hypothesis; test it; detect an error in it; correct the error — is applicable to the problem of inductive inference as well. We have developed a general algorithm that can synthesize logic programs from examples of their behavior. This algorithm uses the diagnosis algorithms as a component. In addition, it has a bug-correction component, that searches the space of possible corrections for a bug, once one is found; this component takes advantage of the intimate connection between the syntax and semantics of logic programs to prune the search.

The Model Inference System is a Prolog implementation of the inductive synthesis algorithm. In comparison to other systems for grammatical inference and program synthesis from examples, the Model Inference System proves to be superior both in its efficiency and in its range of applications.

The bug-correction strategy of the inductive synthesis algorithm is not very appropriate to interactive debugging, since it essentially removes the incorrect part of the program, and searches for a correct component from scratch. In an attempt to apply our techniques of inductive inference to interactive debugging, we have developed a more adaptive bug-correction algorithm. The complexity of programs that can be handled effectively by the interactive debugging system exceeds those that can be synthesized from examples by the Model Inference System. This gain in power is achieved at the expense of generality: the Model Inference System can correct, in a sense, arbitrary bugs in the program, but as a result the class of programs that can be practically synthesized is limited. In order to debug arbitrarily

large programs, we restricted the class of bugs that we attempt to correct automatically; more complex bugs are left to be corrected by the programmer.

## 1.3 Related work

### 1.3.1 The need for debugging

It has been suggested that one way to eliminate the need for debugging is to provide a correctness proof of the program. As Naur and Randell say (in [64], p.51, from [39]):

> "[When] you have given the proof of correctness, ... [you] can dispense with testing altogether."

In conjunction with this quotation, Goodenough and Gerhart [39] recall a simple text formatter program described and informally proved correct by Naur [63], and find seven bugs in it. Three of them can be detected immediately by running the program on a single example. So they comment (p.172):

> "the practice of attempting formal or informal proofs of program correctness is useful for improving reliability, but suffers from the same types of errors as programming and testing, namely, failure to find and validate all special cases ... relevant to a design, its specification, the program and its proof. Neither testing nor program proving can in practice provide complete assurance of program correctness..."

Gerhart and Yelowitz [35] discuss the fallibility of some of the methodologies that claim to eliminate or to reduce the need for debugging. They consider three types of errors: errors in specifications, errors in systematic program construction, and errors in program proving, and provide instances of each of these errors selected from published articles. Concerning errors in specifications, they conclude (p.199):

> "These examples clearly show that specifications must be tested in much the same way that a program is tested, by selecting data with the goal of revealing any errors that might exist."

As a program can be proven correct formally only with respect to another formal description of its intended behavior, this observation suggests that even if the effort in program verification succeeds, it does not solve the problem of program debugging, but simply reduces it to the problem of debugging specifications. If the problem of debugging specifications has not yet revealed itself as a serious one, it may be because there has been no intensive use of formal specifications in full scale programming tasks. From an abstract point of view, however, a specification language that has a partial decision procedure is just another programming language, and for any programming language there is a complex programming task for which there is no simple, self-evidently correct program (e.g. the compiler specification of Polak [70]). As soon as complex specifications are used, there will be a need to debug them.

Manna and Waldinger [56], also suggest that one can never be sure that specifications are correct, and agree that it is unlikely that program verification systems will ever completely eliminate the need for debugging.

Balzer [8] analyzes the role of debugging in the ultimate automatic programming system. He suggests that debugging is an unavoidable component in the process of "model verification", in which the system verifies that it has the right idea of what the target program is. Balzer also comments on the role of assertions in the phase of model verification (p.78):

"...we do not feel that [using assertions] should be part of this phase. If a problem can be conveniently described by a series of assertions, then that is how it should be stated originally."

An even more radical opinion on the prospects of program verification was put forth by DeMillo, Lipton and Perlis [21]:

"A program is a human artifact: a real-life program is a complex human artifact; and any human artifact of sufficient size and complexity is imperfect. The output [of a verification system] will never read VERIFIED".

### 1.3.2 The software-engineering perspective on debugging

Traditionally, the efforts in program debugging were focused on how to bridge the gap between the programmer and the executable program. Print statements and core dumps were the common means of communication between the programmer and the running program.

The problem faced by software-engineers, when trying to improve the situation, was that much of the original, human oriented, aspects of the source program were lost during the compilation step. To bridge this gap, an interactive debugger typically included features such as "memory and register initialization; ability to examine, dump, and/or modify memory locations and registers; selectively execute sections of the program by use of breakpoints; and single step capability" [49].

However, bringing the user closer to the machine was not sufficient to solve the problems of software development. Two major avenues were taken to alleviate this situation; both relying on the use of "higher level languages".

One approach, promoted by the advocates of the Algol-like languages, was that of *structured programming* [101]. The goal was to achieve the construction of reliable software through a reliable method of program development. Discipline and a systematic approach were suggested as a means of curing the program of bugs.

Structured programming marks a significant improvement of our understanding of the programming process. It has, however, a significant limitation, when implemented within the Algol-like family of languages: an Algol or Pascal program cannot be executed, until after the program is completely written and compiles successfully. The restrictive nature of compiler oriented languages limits the opportunities of experimentation, and typically results in a rigid design strategy called by Perlis the *pre-structured* method [66]:

"The pre-structured method is the traditional approach in which the final system language, data structures, control structures and modularization are fixed very early in the [software] life-cycle, usually in the design phase"

This approach is usually supplemented with management techniques and discipline enforcing methodologies at all levels of software development. At the programming language level, this approach is typically supported with a strong-typing mechanism, such as in Pascal [48], Mesa [61], and Ada [45].

We agree that structured programming, augmented with a strong typing mechanism, helps the programmer enormously in avoiding, or detecting early, many syntactic and shallow semantic errors. However, the task of preventing deeper errors — errors that can be detected only by exercising the different components of the system, or worse, by exercising the system as a whole is not supported by this design methodology. On the contrary, the correction of such errors is typically very hard and costly. The drawback of this method is its "pencil and paper" character: typically, there are no effective computerized aids that can help test and experiment with different designs.

Perlis describes a different approach to the problem, called the *prototypical method* [66]:

> "In the prototypical approach, on the other hand, the software is seen as being developed through a sequence of executable prototypes possessing increased functionality, with more and more implementation detail and alternative designs being the rationale for the successive prototype systems."

This approach is best supported with flexible, interpretive languages that have low overhead in implementation, such as Prolog [14], Lisp [95], APL [47], and SmallTalk [46]. A similar observation on the program development methodology that arises from such languages was made by Sandewall [81], who coined the term *structured growth* for it:

> "An initial program with a pure and simple structure is written, tested, and then allowed to grow by increasing the ambition of its modules... The growth can occur both "horizontally", through the addition of more facilities, and "vertically", through a deepening of existing facilities..."

Since these languages are interpretive, the problem of information loss between the source code and the executable program is not as severe. Indeed, the best programming environments and debugging tools have been developed within, and for, these languages [22, 95].

These debugging tools do what debugging systems for compiler-oriented languages did, but better: they allow the programmer to examine and alter the state of the computation, at the programming language level, rather then at the machine level. However, these systems maintain the passive nature of the debugging facility. Our debugging algorithms differ

from existing programming environments by being active: once invoked, they actively present information to the programmer, and ask him about the correctness of intermediate computation steps. Current debugging facilities measure their merit by the amount of information they can provide the programmer with, and by the flexibility in which they can respond to his particular requests. Our debugging algorithms are different: we measure their merit by the succinctness of the information they present to the programmer while diagnosing and correcting a bug.

### 1.3.3 Program testing

One question that arises in the context of program debugging is whether the behavior of the program on a given set of test data is sufficient to determine its correctness. The answer is negative in general, but Goodenough and Gerhart [39] showed that under certain assumptions one may find such a test data set, which they term *reliable*. The idea is to partition the domain of the program to a finite number of equivalence classes, such that the correct behavior of the program on a representative of each class will, by induction, test the entire class, and hence establish its correctness. The testing procedure then is to pick a representative from each class and execute the program on it. If the program behaves correctly on all the test cases, then it is concluded to be correct. The approach does not specify how to proceed in the case where the program behaves incorrectly on some input.

The approach of Goodenough and Gerhart was followed by the Program Mutation Project of Budd et al. [18, 19] and by Brooks [16], among others. Testing by program mutation is based on the assumption that the program to be tested, if wrong, is a close variant (mutant) of the target program. Hence if there is a set of test data on which a program behaves correctly, but all its mutants do not, then we can say with some confidence that the program is correct. The idea was implemented and tested for Fortran and Lisp programs. Brooks [16] applied similar ideas to pure Lisp programs. He treated a more restricted class of errors, and hence was able to prove that with respect to this class of errors, if the program behaves correctly on a certain test data set, then it is correct. He developed a system that can generate such a test data set for simple programs.

The goal of program testing is to provide some evidence for the correctness of the program after it has been developed. But the question of what to do if the program is found to behave incorrectly on some test sample is typically not addressed. We suggest that a system for generating test data can be integrated in a natural way with our debugging algorithms: the former can be used to generate the test data on which the debugging algorithm can operate. As the program changes, new or different test data may be necessary for the program to meet the desired level of reliability; but in an integrated environment, the test data can be developed incrementally, hand in hand with the program that is being debugged.

### 1.3.4 Heuristic approaches to debugging

Sussman [94] studied a model of skill acquisition based on debugging. His system, Hacker, was able to improve its performance in the blocks world by proposing plans and debugging them. After a plan (program) is proposed, it is executed in a "careful" mode, in which primitive operations are applied only if they do not violate their prerequisites, and established subgoals of the plan are protected against being violated by attempts to satisfy other goals. If such a violation occurs, then a method for diagnosing the bug is applied, and a patch to the plan is attempted. If a patch is successful, then, under certain conditions, it may be generalized and the patched, generalized plan is added to the plan library. Sussman admits, however, that the diagnosis and correction strategies incorporated in his system are domain specific, and no immediate generalizations of them are available.

Davis [31] showed how extended tracing and explanation facilities can enable an expert who is not a programmer to debug the rule base of an expert system.

Hayes-Roth, Klahr and Mostow [42] have considered the problem of debugging in the context of acquisition of knowledge for a rule-based expert system. They consider rule-debugging essential to transforming expert's knowledge into executable code, and suggest directions as to how to

automate this process. They classify the possible bugs that arise from attempt to program knowledge into categories such as *excess generality*, *excess specificity*, *invalid knowledge* and *invalid reasoning*, and suggest strategies to cope with such bugs, termed *knowledge refinement strategies*. For example, in case of excess generality, they recommend to "specialize the rule, using case analysis, proof analysis, concept hierarchy specializations". In case of invalid knowledge they recommend to "correct faulty advice, using proof analysis, diagnosis and refinement". An example of how knowledge refinement may be applied to correct invalid strategies in the case of the card game of hearts is described.

A different approach to debugging is taken by the Programmer's Apprentice at MIT [75, 77, 83], and Soloway et al. [90]. Both have developed a bug diagnosis system which can identify a restricted class of bugs. To do so, the system is supplied with a plan for the intended program, and infers the existence and type of bugs in the program by performing some kind of pattern-matching between the plan and the program. Both systems are at an experimental stage, and are currently applicable only to a narrow class of bugs.

Work in inductive inference is surveyed separately in Section 4.1.

## 1.4 Outline

The target and implementation language for the algorithms and systems developed in this thesis is Prolog. Basic concepts of logic programming and Prolog are reviewed in Chapter 2.

In Chapter 3 we develop interactive diagnosis algorithms, which apply to most known functional programming languages; the exact assumptions are spelled out in Section 3.1. The algorithms are developed for three types of errors: termination with incorrect output; termination with missing output; and nontermination. A Prolog implementation of these algorithms, capable of diagnosing pure Prolog programs, is shown, and its behavior is exemplified. This chapter also discusses extensions of the diagnosis algorithms to full Prolog, and methods for mechanizing the process of answering diagnosis queries.

We apply the diagnosis algorithms to two problems: program synthesis from examples, in Chapter 4, and interactive debugging, in Chapter 5.

The inductive program synthesis algorithm, developed in Chapter 4, uses the diagnosis algorithms as a component. Its top-level control structure is identical to the algorithm in Figure 1; it differs from this algorithm in that its initial program — the program it debugs — is the empty one.

The Model Inference System is a Prolog implementation of this algorithm, capable of synthesizing Prolog programs from examples of their behavior. The system is described in Section 4.3, and examples of programs the system has synthesized — an insertion sort program and a grammar for a subset of Pascal — are provided in Appendix I.

A bug-correction strategy that is applicable to interactive debugging is described in Chapter 5. A bug-correction algorithm that tries to modify incorrect components of the initial program is developed, and is incorporated with the diagnosis algorithms to form an interactive debugging system. Its power is demonstrated in a session in which we first interactively debug a faulty quicksort program, and then "debug" it into a quicksort program that removes duplicate elements.

# Chapter 2

# CONCEPTS OF LOGIC PROGRAMMING AND PROLOG

Prolog is both the target and the implementation language for the algorithms and systems developed in this thesis. We introduce here the concepts and definitions necessary to make the thesis self-sufficient. We suspect, however, that this chapter cannot serve as a substitute for an adequate introduction to the uninitiated reader. Several relevant references are mentioned in the following brief historical comments.

Since the introduction of the resolution principle by Robinson [78], there have been attempts to use it as the basic computation step in a logic-based programming language [25, 40]. Nevertheless, for general first order theories, neither resolution nor its successive improvements were efficient enough to make the approach practical. A breakthrough occurred when a restricted form of logical theories was considered, namely Horn theories. The works of Colmerauer, van Emden and Kowalski [27, 33, 54], set the basis for procedural interpretation to Horn-clause logic, and led to the development of the Programming language Prolog [14, 80], which is today a viable alternative to Lisp in the domain of symbolic programming [58, 97].

The model-theoretic, fixpoint and operational semantics of logic programs have been studied by Apt, van Emden and Kowalski [7, 33], among others. The computational complexity of logic programs was studied by Shapiro [86]. The more essential definitions and results of concerning logic programs are reproduced in Section 2.1. The basics of Prolog are surveyed in Section 2.2.

## 2.1 Logic programs

A logic program is a finite set 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 logical atoms, also called *goals*. 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 $B_1,B_2,\dots,B_k$". $A$ is called the clause's ***head*** and the $B$'s the clause's ***body***. If the $B$'s are missing, the sentence reads "$A$ is true" or "goal $A$ is satisfied". Given a goal, or a conjunction of goals, a set of definite clauses can be executed as a program, using this procedural interpretation.

An example of a logic program for insertion sort is shown as Program 1.

**Program 1:** Insertion sort

```
isort([X|Xs],Ys) ← isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[X,Y|Ys]) ← X ≤ Y.
insert(X,[Y|Ys],[Y|Zs]) ← X > Y, insert(X,Ys,Zs).
insert(X,[],[X]). ▯
```

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 $[A,B|X]$ with the list [1,2,3,4] is $A$=1, $B$=2, $X$=[3,4], and unifying $[X|Y]$ with $[a]$ results in $X=a$, $Y$=[].

Figure 2 establishes some relationships between logic programs and concepts from conventional programming languages.

| | |
|---|---|
| Procedures | Definite clauses |
| Procedure calls | Goals |
| Binding mechanism, data selectors and constructors | Unification |
| Execution mechanism | Nondeterministic goal reduction |

**Figure 2:** Common programming concepts in logic programs

### 2.1.1 Computations

A computation of a logic program $P$ can be described informally as follows. The computation starts from some initial (possibly conjunctive) goal $A$; it can have two result: success or failure. If a computation succeeds, then final values of the variables in $A$ are conceived of as the output of the computation. A given goal can have several successful computations, each resulting in a different output.

The computation progresses via nondeterministic goal reduction. At each step we have some current goal $A_1,A_2,\dots,A_n$. A clause $A'\leftarrow B_1,B_2,\dots,B_k$ in $P$ is then chosen nondeterministically; the head of the clause $A'$ is then unified with $A_1$, say, with substitution $\theta$, and the reduced goal is $(B_1,B_2,\dots,B_k,A_2,\dots,A_n)\theta$. The computation terminates when the current goal is empty.

We proceed to formalize these notions. We follow the Prolog-10 manual [14] in notational conventions, and Apt and van Emden [7] in most of the definitions. A *term* is either a constant, a variable or a compound term. The constants include integers and atoms. The symbol for an atom can be any sequence of characters, which is quoted if there is possibility of confusion with other symbols (such as variables, integers). Variables are distinguished by an initial capital letter. If a variable is only referred to once, it does not need to be named and may be written as an "anonymous" variable indicated by a single underline _.

A *compound term* comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is characterized by its name, which is an atom, and its arity or number of arguments. An atom is considered to be a functor of arity 0.

A *substitution* is a finite set (possibly empty) of pairs of the form $X \rightarrow t$, where $X$ is a variable and $t$ is a term, and all the variables $X$ are distinct. For any substitution $\theta = \{X_1 \rightarrow t_1, X_2 \rightarrow t_2, ..., X_n \rightarrow t_n\}$ and term $s$, the term $s\theta$ denotes the result of replacing each occurence of the variable $X_i$ by $t_i$, $1 \leq i \leq n$; the term $s\theta$ is called an *instance* of $s$.

A substitution $\theta$ is called a *unifier* for two terms $s_1$ and $s_2$ if $s_1\theta = s_2\theta$. Such a substitution is called the *most general unifier* of $s_1$ and $s_2$ if for any other unifier $\theta_1$ of $s_1$ and $s_2$, $s_1\theta_1$ is an instance of $s_1\theta$. If two terms are unifiable then they have a unique most general unifier [78].

We define computations of logic programs. Let $N = A_1, A_2, ..., A_m$, $m \geq 0$, be a (conjunctive) goal and $C = A \leftarrow B_1, ... B_k$, $k \geq 0$, be a clause such that $A$ and $A_1$ are unifiable with a substitution $\theta$. Then $N' = (B_1, ... B_k, A_2, ..., A_m)\theta$ is said to be *derived*[1] from $N$ and $C$, with substitution $\theta$. A goal $A_j\theta$ of $N'$ is said to be *derived* from $A_j$ in $N$. A goal $B_j\theta$ of $N'$ is said to be *invoked* by $A_1$ and $C$.

Let $P$ be a logic program and $N$ a goal. A *derivation* of $N$ from $P$ is a (possibly infinite) sequence of triples $<N_i, C_i, \theta_i>$, $i = 0, 1, ...$ such that $N_i$ is a goal, $C_i$ is a clause in $P$ with new variable symbols not occuring previously in the derivation, $\theta_i$ is a substitution, $N_0 = N$, and $N_{i+1}$ is

[1]The usual definition of derivation allows any atom in the goal to be resolved with a clause, not necessarily the first one. Apt and van Emden [7] showed that the restriction of derivations to resolve the first atom only does not violate the completeness of the proof procedure. We adopt this restriction to simplify the definition of total correctness below.

derived from $N_i$ and $C_i$ with substitution $\theta_i$, for $i \geq 0$.

A derivation of $N$ from $P$ is called a *refutation of N from P* if $N_l = \square$ (the empty goal) for some $l \geq 0$. Such a derivation is finite and of length $l$, and we assume by convention that in such a case $C_l = \square$ and $\theta_l = \{\}$. If there is a refutation of a goal $A$ from a program $P$ we also say that $P$ *succeeds* on $A$.

Figure 3 shows a refutation of the goal *isort*([2,1],$X$) from Program 1. It assumes that the programs for the arithmetic predicates $<$ and $\leq$ operate as though they were represented by an infinite set of unit clauses.

```
<isort([2,1],L),
      (isort([X|Xs],Ys) ← isort(Xs,Zs), insert(X,Zs,Ys)),
            {X→2,Xs→[1],L→Ys}>
<(isort([1],Zs), insert(2,Zs,Ys)),
      (isort([X1|Xs1],Ys1) ← isort(Xs1,Zs1),insert(X1,Zs1,Ys1))
            {X1→1,Xs1→[],Zs→Ys1}>
<(isort([],Zs1), insert(1,Zs1,Zs), insert(2,Zs,Ys)), isort([],[]), {Zs1→[]}>
<(insert(1,[],Zs), insert(2,Zs,Ys)), insert(X2,[],[X2], {Zs→[X2],X2→1})>
<insert(2,[1],Ys)),
      (insert(X3,[Y3|Y3s],[Y3|Z3s])←X3>Y3, insert(X3,Y3s,Z3s)),
            {X3→2,Y3→1,Y3s→[],Ys→[1|Z3s]}>
<(2>1,insert(2,[],Z3s)), 2>1,{}>
<insert(2,[],Z3s), insert(X4,[],[X4]), {Z3s→[2],X4→2})
<□,□,{}>
```

**Figure 3:** An example of a refutation

A more intuitive, though less complete way to describe successful computions of logic programs (i.e. refutations) is via the *refutation tree*. In the refutation tree, nodes are goals that occur in the computation, with their variables instantiated to their final values, and arcs represent the relation of goal invocation. The refutation of *isort*([2,1],$L$) in Figure 3 corresponds the refutation tree in Figure 4. Depth of indentation reflects depth in the tree.

```
isort([2,1],[1,2])
        isort([1],[1])
                isort([],[])
                insert(1,[],[1])
        insert(2,[1],[1,2])
                2>1
                insert(2,[],[2])
```

**Figure 4:** An example of a refutation tree

### 2.1.2 Semantics

We define semantics of logic programs, which is a special case of the standard model-theoretic semantics of first order logic [33]. An *interpretation* is a set of variable-free goals. A substitution $\theta$ *satisfies* a goal $A_1,A_2,\ldots,A_n$ in an interpretation $M$ if $A_i\theta$ is in $M$, for $1\leq i\leq n$.

**Definition 2.1:** We say that a clause $A\leftarrow B_1,B_2,\ldots,B_n$ *covers* $A'$ in $M$ if there is a substitution $\theta$ that unifies $A$ with $A'$ and satisfies $B_1,B_2,\ldots,B_n$ in $M$.

A clause is *true* in $M$ if every variable-free goal it covers in $M$ is in $M$, and *false* otherwise. A program $P$ is true in $M$ if every clause in $P$ is true in $M$. In the context of logic, we use *correct* as a synonym of true and *incorrect* as synonym of false.

Let $P$ be a program. The *Herbrand universe* of $P$, $H(P)$, is the set of all variable-free goals constructable from constants and functors that occur in $P$. We define the *interpretation of P*, $M(P)$, to be the set $\{A|\ A$ is in $H(P)$ and $P$ succeeds on $A\}$.

Van Emden and Kowalski [33] show that $M(P)$ is the minimal interpretation in which $P$ is true. They also associate a transformation $\tau_P$ with any program $P$ and show that $M(P)$ is the least fixpoint of $\tau_P$. The transformation $\tau_P$ is defined as follows. Let $M$ be an interpretation. Then a variable-free atom $A$ in $\tau_P(M)$ iff there is clause in $P$ that covers $A$ in $M$. An alternative definition of truth is that a program $P$ is true in $M$ iff

$M\subseteq\tau_P(M)$.

We say that a program $P$ is *complete* in $M$ if $M\subseteq M(P)$. A program $P$ is correct and complete in $M$ iff $M=M(P)$.

A *domain* is a set of goals. We say that a program $P$ is *everywhere terminating* over a domain $D$ if for no goal $A$ in $D$ there is an infinite derivation of $A$ from $P$.

Every program defines a domain, as follows. Let $P$ be a program. A goal $A$ is in the *domain of P* if it occurs in a derivation of $B$ from $P$, for some goal $B$ in $H(P)$. The domain of a program is all the goals that can be invoked in derivations of ground goals from $P$.

**Lemma 2.2:** Let $P$ be a program that is everywhere terminating over $H(P)$. Then $P$ is everywhere terminating over its domain.

**Proof:** If a program is not everywhere terminating over its domain then there is some goal $A$ in its domain such that there is an infinite derivation of $A$ from $P$. But $A$ occurs in a derivation of some goal $B$ in $H(P)$. Hence there is a infinite derivation of $B$ from $P$, in contradiction to the assumption that $P$ is everywhere terminating over $H(P)$. []

A program that is everywhere terminating over its domain, and is correct and complete in $M$ is called *totally correct* in $M$. The target of our debugging and synthesis algorithms are totally correct programs.

### 2.1.3 Complexity measures

We define complexity measures over refutations, using the notion of refutation tree. Let $R$ be a refutation. We define the *length* of $R$ to be the number of nodes in the refutation tree. The *depth* of $R$ is the depth of the tree. The *goal-size* of $R$ is the maximal size of any node of the refutation tree, where the size of a goal is the number of symbols in its textual representation.

**Definition 2.3:** We say that a logic program $P$ is of *goal-size complexity* $G(n)$ if for any goal $A$ in $M(P)$ of size $n$ there is a refutation $R$ of $A$ from $P$ of goal-size $\leq G(n)$.

$P$ is of *depth complexity* $D(n)$ if for any goal $A$ in $M(P)$ of size $n$ there is a refutation $R$ of $A$ from $P$ of depth $\leq D(n)$.

$P$ is of *length complexity* $L(n)$ if for any goal $A$ in $M(P)$ of size $n$ there is a refutation $R$ of $A$ from $P$ of length $\leq L(n)$.

We say that an interpretation $M$ is of goal-size complexity $G(n)$ if there is a logic program $P$ such that $M(P) \models M$ and the goal-size complexity of $P$ is $G(n)$. We assume similar definitions for the depth complexity and length complexity of interpretations.

In [86] we have established the following relationships between complexity of refutations of logic programs and complexity of computations of alternating Turing machines [24].

**Theorem 2.4:** Let $P$ be a logic program of depth complexity $D(n)$, goal-size complexity $G(n)$ and length complexity $L(n)$. Then there exists an alternating Turing machine $T$ and a constant $c$ uniform in $P$ such that $T$ operates in time $cD(n)G(n)$ space $cG(n)$ and tree size $cL(n)G(n)$, and that $L(T) \models M(P)$.

**Theorem 2.5:** Let $T$ be an alternating Turing machine that accepts a language $L$ in time $T(n)$, space $S(n)$ and tree-size $Z(n)$. Then there exists a logic program $P$ of depth complexity $T(n)$, goal-size complexity $S(n)+c$ and length complexity $Z(n)$ and a goal $A$ that contains the variable $X$ such that $L(T) \models \{X\theta \mid A\theta$ is in $M(P)\}$, where $c$ is a constant uniform in $M$.

Using the results of Chandra et. al [24], one can characterize the complexity of logic programs in terms of deterministic complexity measures.

## 2.2 Prolog

The relationship between logic programming and Prolog is a reminiscent of the relationship between the $\lambda$-calculus and Lisp. The "pure" part of Prolog is simply a realization of refutations of logic programs on a sequential machine. Prolog extends beyond this pure core in several ways, for reasons of efficiency and expressiveness. We explain how the nondeterminism of logic programs is implemented in sequential Prolog, and survey some extensions of Prolog, used in the programs and systems described below. We follow the Prolog-10 manual [14] in doing so.

### 2.2.1 The execution and backtracking mechanism

Prolog's execution mechanism is a sequential simulation of the nondeterministic computation mechanism described in Section 2.1. Instead of choosing the next clause to be invoked nondeterministically, Prolog tries all unifiable clauses sequentially, in the order they occur in the program text. When it fails to find such a clause, it backtracks to the last choice point. The implementors of Prolog in describe the backtracking mechanism as follows [14]:

"To *execute* a goal, the system searches forwards from the beginning of the program for the first clause whose head *matches* or *unifies* with the goal. The *unification* process [78] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then *activated* by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it *backtracks*, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal."

Prolog's sequential proof procedure is correct, as any refutation it finds is indeed a refutation, but is incomplete, as it may fail to find a refutation although such exists. This may happen if there are both refutations and infinite computations for a given goal, which result from different choices of clauses, and Prolog happens to start and explore an infinite computation first.

The problem does not occur if the logic program has no infinite computations. Since we restrict ourselves programs which are totally correct over their domains, this incompleteness of Prolog's proof procedure is no limitation for such programs; and incorrect programs can be debugged, as the title of this thesis suggests.

Logic programs with Prolog's sequential proof procedure are referred to as *pure Prolog*; in particular, all the extensions described below are excluded.

### 2.2.2 Running time and the "occur check"

The most basic operation of Prolog is the unification of a variable against a term. The unification algorithm [78] requires that this operation succeed only if the variable does not occur in the term. The check of whether this case holds is called the *occur check*. The unification algorithm incorporated in Prolog does not include the occur check, since with this check Prolog could not be a practical programming language. For example, the runtime of the straightforward program for appending two lists would be in the order of square of the size of the input lists, rather than linear. The lack of the occur check is not felt in most programming tasks, including the development of the systems described in this thesis. Colmerauer [28] describes a theoretical model of Prolog without the occur check, based on unification over infinite trees.

The complexity measures defined for logic programs are nondeterministic; we would like to get hold of a more practical measure to evaluate the performance of concrete Prolog programs. We define the *time* of the computation of Prolog on a give goal to be the number of clause invocations performed until all solutions are found to that goal. This definition is justified in part by the absence of the occur check in Prolog; without the occur check, the time required to invoke a clause in most practical Prolog programs is bounded by a constant that depends on the program. This claim does not hold when unification is used for tasks other than variable binding, data selection and construction, such as the check of the equality of two compound terms in a goal. This claim fails also if the program modifies itself in runtime by using *assert*.

As a consequence of this property, Prolog implementors use the number of invocations per seconds as a measure of the speed of their implementation. The Japanese Fifth Generation Computer Project suggested the term *LIPS* (logical inferences per second). Micro-Prolog [57] is known to run 110 LIPS on the z-80 microprocessor, the Prolog-10 performs about 24K LIPS on the KL-10, and the Japanese project is aiming at a personal logic programming machine of 1 mega LIPS, and a parallel machine of 1 giga LIPS [62].

To give the reader a feel for these measures, we counted the number of invocations needed to sort an ordered list using quicksort (an $n^2$ process for the naive choice of the partition element). The number of clause invocations needed to sort a list of 10, 20, and 30 elements are 87, 271, and 523, respectively.

### 2.2.3 Control

The order of goals in the clause determines the order in which they are solved. The order of clauses in the program determines the order in which they are tried. Besides the sequencing of goals and clauses, Prolog provides one other facility for specifying control information. This is the *cut* symbol, written "!". A cut is inserted in the program just like a goal, and its effect is as follows [14]:

"When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the "parent goal", i.e. that goal which matched the head of the clause containing the cut, and caused the clause to be activated. In other words, the cut operation commits the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded. The goals thus rendered "determinate" are the parent goal itself, any goals occurring before the cut in the clause containing the cut, and any subgoals which were executed during the execution of those preceding goals."

The cut is Prolog's *go to*. It is a low-level control primitive, and unrestricted use of it tends to result in an unstructured, incomprehensible code. Similar to the way *go to* can be used to implement higher level control constructs as the *while* and *repeat* loops in conventional programming languages, *cut* can be used to define higher level control predicates in Prolog.

One such construct is negation, Prolog's *not*. Procedurally, a goal *not*($A$) succeeds iff the goal $A$ fails. A fixpoint semantics to *not* is given by Apt and van Emden [7]. The *not* construct can be implemented using *cut*, as follows:

$$not(P) \leftarrow P,\ !,\ fail.$$
$$not(P).$$

where *fail* is a goal that fails (i.e. has no clauses defined for it). The *not* program uses the *meta-variable* feature of Prolog, that allows goals to be determined dynamically, at runtime; it is similar to passing procedure names as parameters in a conventional programming language. Tired of being accused that Prolog's *not* is not really not, the implementors of Prolog-10 switched to "\+" instead, hoping that no one would have any emotional attachments to such an obscure symbol; we follow this convention in Prolog code that is quoted directly from the actual systems developed.

Another, more conservative extension to Prolog is *or*, written ";", and is defined as follows:

$$(P ; Q) \leftarrow P.$$
$$(P ; Q) \leftarrow Q.$$

For example, the following program for *grandfather*

$$grandfather(X,Y) \leftarrow father(X,Z),\ (\ father(Z,Y)\ ;\ mother(Z,Y)\ ).$$

computes the same relation as:

$$grandfather(X,Y) \leftarrow father(X,Z),\ father(Z,Y).$$
$$grandfather(X,Y) \leftarrow father(X,Z),\ mother(Z,Y).$$

but more efficiently, since it does not recompute *father*($X$,$Z$) for solving *mother*($Z$,$Y$). The *or* construct can be viewed as a notational convention, since, if the program has no *cut*, it can always be removed by introducing a new predicate, as in:

$$grandfather(X,Y) \leftarrow father(X,Z),\ parent(Y,Z).$$

$$parent(X,Y) \leftarrow father(X,Y).$$
$$parent(X,Y) \leftarrow mother(X,Y).$$

Another useful control construct is Prolog's *if-then-else* "$\rightarrow$". The goal ($p \rightarrow q$ ; $r$) is executed as follows: "solve $p$, if successful, solve $q$, else solve $r$". The *if-then-else* construct can be defined as follows.

$$(P \rightarrow Q\ ;\ R) \leftarrow P,\ !,\ Q.$$
$$(P \rightarrow Q\ ;\ R) \leftarrow R.$$

The goal $P \rightarrow Q$, when occurring other than as one of the alternatives of a disjunction, is equivalent to $P \rightarrow Q$ ; *fail*.

### 2.2.4 Side-effects

Prolog is a functional language by nature, and many applications which typically require global data structures such as stacks and queues, can be implemented in it cleanly and efficiently without side-effects (see Program 17, page 124 for an example of a functional implementation of a queue). It is, however, essential sometimes for a program to modify the state of the system. Prolog has two predicates, *assert* and *retract*, whose execution results in side-effecting the database of clauses, i.e. the program being executed. *assert*($X$) adds the clause $X$ to the program (it has two variants, *asserta* that adds the asserted clause as the the first clause in the procedure, and *assertz* that adds it as the last). *retract*($X$) deletes a clause that unifies with $X$ from the database, if such a clause exists, and fails otherwise.

Side-effects are used by the systems developed in this thesis for two purposes only: to modify the program being synthesized or debugged, and to record user's answers to queries, so he is not asked the same query twice. All the rest of the computations are done functionally; system i/o excluded, of course.

### 2.2.5 Second order predicates

It is sometimes convenient to refer explicitly to the solutions to a goal. Prolog provides predicates that enables one to compute this set. The goal *bagof(X,P,S)* returns in *S* the nonempty list of all instances of *X* for which *P* is true , in the order they are found when solving *P*, and fails if no such solution exists. For example, one can compute all pairs of two lists using the following goal:

?– *bagof((X–Y),(member(X,[1,2,3]), member(Y,[a,b,c,d])),S).*

*S* = [1–*a*,1–*b*,1–*c*,1–*d*,2–*a*,2–*b*,2–*c*,2–*d*,3–*a*,3–*b*,3–*c*,3–*d*]

*setof* is similar to *bagof*, except that it returns an ordered list of solutions, ordered according to Prolog's canonical ordering of terms, with duplicates removed. Both predicates are implementable within Prolog, using side-effects and forced backtracking. The resulting code is not the most elegant, but no Prolog user ever needs to look at it. The implementation of *bagof* in Program 2 is a simplified version of its implementation in Prolog-10.

**Program 2: An implementation of *bagof***

```
bagof(X,P,Xs) ←
        asserta(item('$bag')), P, asserta(item(X)), fail ;
        reap([],Xs), nonempty(Xs).

reap(Xs,Ys) ←
        retract(item(X)), !,
        ( X=='$bag' → Xs=Ys ; reap([X|Xs],Ys) ).

nonempty([_|_]). ▯
```

The marker *'$bag'* is necessary for recursively nested *bagof*s to work properly; without it they may steal each other's solutions. The actual Prolog implementation of *bagof(X,P,S)* is more sophisticated; if *P* has

variables not occuring in *X*, Prolog indexes the instances of *X* on the different solutions of these free variables in *P*. Free variables in *P* have to be quantified existentially to override this indexing. For example, Prolog's *bagof* (not the one above) will return,

?– *bagof(X,(member(X,[1,2,3]), member((X–Y),[1–a,2–a,2–b,3–b])),S).*

*S* = [1,2],
*Y* = *a* ;

*S* = [2,3],
*Y* = *b*

but if we existentially quantify the variable *Y* in the example above we get

?– *bagof(X,Y↑(member(X,[1,2,3]),*
    *member((X–Y),[1–a,2–a,2–b,3–b])),S).*

*S* = [1,2,2,3]

### 2.2.6 Meta-programming

"I'd rather write programs that help me write programs than write programs."

— An anonymous meta-programmer

Considering the goals of this thesis, the most important aspect of Prolog is the ease with which Prolog programs can manipulate, reason about and execute other Prolog programs. Program 3 below shows an interpreter for pure Prolog, written as a pure Prolog program. It uses the system predicate *clause(P,Q)*, which functions as if the program is reresented as a list of unit clauses *clause(P,Q)*← for any clause *P*←*Q* in it. A unit clause *P*← is represented as *clause(P,true)*←, a conjunctive goal $A_1,A_2,...,A_{n-1},A_n$ is represented as $(A_1,(A_2,(...,(A_{n-1},A_n)...)))$.

The interpreter operates as follows. On the goal *true* is simply succeeds, using the first clause. It solves a conjunctive goal (*A*,*B*) using the second clause, by calling itself recursively on *A* and *B*. The "real" work is

**Program 3:** An interpreter for pure Prolog

```
solve(true).
solve((A,B)) ← solve(A), solve(B).
solve(A) ← clause(A,B), solve(B). ▯
```

done in the third clause: on a unit goal $A$ it invokes a clause $A \leftarrow B$, and recursively solves $B$. The interpreter that executes this interpreter performs the clause invocation and the unification when solving the goal *clause*$(A,B)$. Failure to solve the goal *solve*$(B)$ would force the interpreter that executes this interpreter to backtrack and provide new solutions to *clause*$(A,B)$.

This interpreter cannot use the system predicates such as arithmetic and input-output. We extend the interpreter using *system*$(X)$, a predicate that succeeds if $A$'s main functor is a system predicate. System predicates have no clauses defined for them, and they are solved directly by the Prolog system. Since *true* is a system predicate that succeeds, it need not be mentioned explicitly in the interpreter below.

```
solve((A,B)) ← solve(A), solve(B).
solve(A) ← system(A), A ; clause(A,B), solve(B).
```

The extended interpreter can, for example, execute the insertion sort program above.

```
?- solve(isort([2,1,3,6,4,2],X)).

X = [1,2,2,3,4,6]
```

# Chapter 3

# PROGRAM DIAGNOSIS

This chapter presents diagnosis algorithms, which are interactive algorithms that can identify a bug in a program that behaves incorrectly.

We distinguish three types of errors in program behavior: termination with an incorrect output; termination with a missing output; and nontermination. Each type of error receives the following uniform treatment. A property of procedures is defined, for which an error in the program implies that it contains a procedure having this property. This procedure needs to be modified to eliminate the error in the program.

Algorithms that diagnose erroneous procedures are developed. These algorithms are interactive, as they query the user for information about the intended behavior of the program. The computational- and query-complexity of the algorithms is analyzed. We then instantiate the definitions and algorithms to logic programs, develop Prolog implementations of them and demonstrate their performance.

From one point of view, the diagnosis algorithms are a quantitative improvement on the standard single-stepping trace technique, since they perform the same function of showing the programmer the behavior of the program, except that they save some human labor by showing only pieces of the computation that are relevant for diagnosing the bug. They seem to function rather well as a replacement, or enhancement, to the standard trace package of an advanced programming environment.

From another point of view, however, these algorithms open up the way for a much more automated form of debugging, since they do not necessarily rely on a human agent to answer their queries: stored answers to previous queries, a previous version of the program that is known to work, assertions concerning the input/output behavior of procedures — all can be used to answer the queries in place of the human programmer, and help locate the bug.

## 3.1 Assumptions about the programming language

Although the target programs of the diagnosis algorithms are logic programs, the methods they employ are general, and are applicable to a large class of programming languages. We therefore describe an abstract set of properties of a programming language that are sufficient for the diagnosis algorithms to apply, and develop the algorithms in this general setting. Describing these algorithms in a logic-independent context also allows their natural extension to full Prolog, as discussed in Section 3.6.

Our approach is geared towards languages in which the basic computation mechanism is a procedure (or function) call, but is insensitive to the inner workings of procedures. The diagnosis algorithms abstract away all the details of the computation, except the procedure calls performed, their inputs, and their outputs.

"Purified" versions of many existing and proposed programming languages satisfy our model, as elaborated below. 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 and no side-effects, Backus' applicative languages, Hope, and Harel's And/Or programs.

We describe our assumptions more formally. A *program* is a finite set of procedures. A *procedure* is defined by its name, its arity, and its "code". In this abstract setting we do not specify what that code is, but assume that the language has a computable interpreter, that maps this code into a class of legal behaviors for that procedure, which we describe via top level

traces [16].

A *top level trace* of a procedure $p$ on input $x$ that returns $y$ is a finite (possibly empty) ordered set of triples:

$$\{<p_1,x_1,y_1>, <p_2,x_2,y_2>, ..., <p_k,x_k,y_k>\}$$

Where the $p$'s are procedure names, and the $x$'s and $y$'s are vectors over some domain $D$, not containing the symbol $\perp$. The intended interpretation of such a top level trace is as follows: if the set is empty, it means that the procedure $p$, on input $x$, has a legal computation that returns $y$ without performing any procedure calls; otherwise it means that $p$, on input $x$, has a possible computation that calls $p_1$ on $x_1$, and if this call returns $y_1$, then $p$ calls $p_2$ on $x_2$, and if this call returns $y_2$ then... , then $p$ calls $p_k$ on $x_k$, and if this call returns $y_k$, then $p$ returns $y$.

The restriction that the top-level trace is ordered assumes a sequential interpreter. This restriction needs to be relaxed to handle concurrent programming languages, and we suspect that most of the techniques of the diagnosis algorithms will carry through. The programming language can be nondeterministic (with bounded nondeterminism), and in such a case for each triple $<p,x,y>$ there can be only finitely many legal top level traces.

We describe legal computations of a program via computation trees. Let $P$ be a program. A *partial computation tree* of $P$ is a rooted, ordered tree whose nodes are triples $<p,x,y>$, where $p$ is a procedure name, $x$ and $y$ are vectors over $D$, and for every internal node $<p,x,y>$ in $T$ with sons $S$, $S$ is a legal top level trace of $<p,x,y>$, where $p$ is a procedure in $P$. A tree $T$ is a *complete computation tree* of $P$ (computation tree for short) if it is a partial computation tree of $P$ in which all leaves have empty top level traces. Complete computation trees represent successful computations, and correspond to refutation trees in logic programs.

A tree $T$ is a *reachable computation tree* of $P$ if it is a partial computation tree of $P$ such that for every path in $T$, every subtree to the left of any node in the path, except the last node, are complete computation trees. Reachable trees represent partial computations that can be realized by the interpreter of the language. A tree $T$ is a *infinite reachable computation tree* of $P$ if it is a reachable computation tree of $P$ that contains an infinite path. Infinite reachable computation trees represent nonterminating computations.

We require that if a program $P$ has no infinite reachable computation tree rooted at $<p,x,y>$ then it has only finitely many such reachable computation trees. This implies that if the program is not diverging, we can explore all its legal computations in finite time.

We assume that the programming language has an interpreter, that operates as follows. When invoked with a procedure call, the interpreter performs some computations, including invoking itself recursively zero or more times with other procedure calls, and then returns some output. We assume that the computation trees of a program represent accurately the possible behaviors of this interpreter: given a program $P$, and a procedure call $<p,x>$, we assume that the interpreter responds as follows: it diverges if $P$ has a infinite reachable computation tree rooted at $<p,x,y>$, for some $y$; otherwise it nondeterministically outputs a $y$ for which $P$ has a complete computation tree rooted at $<p,x,y>$, if such a $y$ exists; otherwise it returns $\perp$.

We define semantics for our programming language, analogous to the model theoretic semantics of logic programs. An interpretation $M$ is a set of triples $<p,x,y>$, where $p$ is a name of a procedure with $n$ input variables and $m$ output variables, $x$ 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 $<p,x,y>$ is in $M$. We say that $y$ is a *correct output* of a procedure call $<p,x>$ in $M$ if $<p,x,y>$ is in $M$, or, in case $y=\perp$, if for no $y'$, $<p,x,y'>$ is in $M$. Otherwise, $y$ is said to be an *incorrect output* in $M$.

A program $P$ is *partially correct* in $M$ if the root of every complete computation tree of $P$ is in $M$. A program $P$ is *complete* in $M$ if every triple in $M$ is a root of a complete computation tree of $P$. A program $P$ is *everywhere terminating* if it has no infinite reachable computation tree, otherwise it is *diverging*. A program is *totally correct* if it is partially correct, complete, and everywhere terminating.

We point out some properties of the programming language, that follow from our definition. The functionality of the language follows from the "context-freeness" of the trees. A complete computation tree represents a legal computation of the program, independent of the context in which it is executed. Another property is the monotonicity of the set of legal computations of programs with respect to the subset relation: if a program $P_1$ is a subset of $P_2$, then the (partial, complete, infinite reachable) computation trees of $P_1$ are a subset of those of $P_2$.

We employ two computational complexity measures that are intended to reflect the machine resources consumed by an interpreter as above: *length*, the number of procedure calls performed during a computation, and *depth*, the maximal depth of procedure invocation (stack depth). The length of a computation is also the number of nodes in the computation tree, and the depth of the computation is the depth of this tree. Sometimes we consider also the *branching* of the computation, defined to be the maximal branching of the computation tree.

Our diagnosis algorithms typically involve simulating a computation, and performing some additional operations in between. We assume that a simulator of the interpreter can be constructed, that works in at most a constant overhead in the length or depth of the computation. Furthermore, we assume that such a simulator can be extend in natural ways: that we can extended it with a routine that performs some operation whenever a procedure call returns; that we can modify the interpreter so it calls another routine with a procedure call, instead of calling itself recursively; and that we can store results of procedure calls for further processing.

The following is an informal summary of the properties of the programming language:

- A program is a finite set of procedures.
- A procedure has a name, arity, and "code".
- The code of the procedure determines the set of its possible behaviors, which are described via top-level traces.
- The legal computations of a program are described via computation trees constructable from top-level traces of its procedures.

## 3.2 Diagnosing termination with incorrect output

### 3.2.1 Correctness

If a program is partially correct then every subprogram of it is also partially correct, as the computation trees of a subprogram are a subset of those of the program as a whole. The opposite, however, is not always true. For example, a subprogram $P'$ of a partially correct program $P$ can be modified in a way that maintains the partial correctness of $P'$, but violates the partial correctness of $P$, as in the following program for (unary) integer multiplication.

```
times(0,Y,0).
times(X,0,Z) ← plus(X,0,Z).
times(s(X),Y,Z) ← times(X,Y,U), plus(U,Y,Z).

plus(0,Y,Y).
plus(X,0,Z) ← plus(X,0,Z).
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```

This program is partially correct with respect to the standard interpretation of *times* and *plus*, as whenever the computation on *times*$(X,Y,Z)$ terminates (and this happens if $Y \neq 0$), $Z$ is $X$ times $Y$. However, we can modify *plus*$(X,Y,Z)$ to return in $Z$ the value of $X$ if $Y=0$. This modification preserves the partial correctness of the *plus* program, but violates the partial correctness of the program as a whole, since now *times*$(1,0,Z)$ will return $Z=1$ instead of diverging as before.

This example shows that the property of partial correctness is not local to procedures, hence we need a finer concept for this diagnosis task. The definition we propose is that a procedure $p$ will be called *correct* with respect to an interpretation $M$ if whenever all procedure calls performed by $p$ return an output correct in $M$, then $p$ returns an output correct in $M$.

**Definition 3.1:** A procedure $p$ *covers* $<p,x,y>$ with respect to $M$ if $<p,x,y>$ has a legal top level trace which is a subset of $M$.

A procedure $p$ is *correct* in $M$ if every triple $<p,x,y>$ covered by $p$ with respect to $M$ is in $M$. Otherwise $p$ is *incorrect* in $M$.

These definitions parallel the definitions for logic programs, given in Section 2.1.

If a procedure $p$ is incorrect in $M$, then it has a top level trace in $M$ for some triple $<p,x,y>$ not in $M$. Such a top level trace is called a *counterexample* to the correctness of $p$ in $M$.

We show below that if every procedure in a program is correct, then the program is partially correct; the opposite, however, is not always true. For example, any everywhere nonterminating program is partially correct, but the nonterminating program

$$f(X,Y) \leftarrow integer(2\times X),\ f(2\times X,Y).$$

is incorrect in the interpretation $M = \{<f, X, 1> \mid X \text{ is an integer}\}$, since $<f, 0.5, 1>$ is not in $M$, but its top level trace, $\{<f, 1, 1>\}$ is. Theorem 3.2 shows that if a program has a finite computation that returns an incorrect output then it contains an incorrect procedure.

**Theorem 3.2:** Let $P$ be a program and $M$ an interpretation. If $P$ is not partially correct with respect to $M$ then $P$ contains a procedure incorrect in $M$.

**Proof:** Let $p$ be a procedure in $P$ that on input $x$ returns an output $y \neq \perp$ incorrect in $M$. We examine the computation tree of $<p,x,y>$. Consider the first node $<q,u,v>$ in the post-order traversal of the tree which is not in $M$. Such a node exists, since the root is not in $M$. By the choice of this node, all its sons (if any) are in $M$, hence $q$ covers $<q,u,v>$. But since $<q,u,v>$ is not in $M$, it follows that $q$ is incorrect in $M$. []

## 3.2.2 A single-stepping algorithm for diagnosing incorrect procedures

The proof of Theorem 3.2 suggests a diagnosis algorithm for detecting an incorrect procedure in a program $P$ that is not partially correct, using a standard tracing technique: single step through the procedure calls of the computation. The first procedure to return an incorrect output is incorrect.

We formalize the algorithm and argue that it is correct. The algorithm uses a *ground oracle* for $M$, which is a device that, on input $<p,x,y>$, outputs *yes* if $<p,x,y>$ is in $M$ and *no* otherwise.

**Algorithm 1:** Tracing an incorrect procedure by single-stepping

*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 triple $<q,u,v>$ not in $M$ such that $q$ covers $<q,u,v>$.

*Algorithm*: Simulate the execution of $p$ on $x$ that returns $y$; whenever a procedure call $<q,u>$ returns an output $v$, check, using a ground oracle, whether $<q,u,v>$ is in $M$. If it is not, return $<q,u,v>$ and terminate. ▯

Algorithm 1 is correct since the order in which procedure calls return in a computation corresponds exactly to the post-order traversal of the computation tree. Consider the first node $<q,u,v>$ 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 $q$ is incorrect in $M$.

## 3.2.3 A Prolog implementation

We relate the above discussion to logic programs. A top level trace of a procedure corresponds to the body of a ground instance of a clause. Correctness of a program $P$ in $M$ is its model-theoretic truth in $M$: a clause is false in $M$ iff it covers a goal not in $M$, and a counterexample to

the correctness of $P$ is a false instance of a clause in $P$. In the case of logic programs, Algorithm 1 amounts to a search for a false instance of a clause in $P$, given a proof of a false conclusion from $P$. Program 4 implements Algorithm 1.

**Program 4:** Tracing an incorrect procedure by single-stepping

```
fp((A,B),X) ← !,
    fp(A,Xa),
    ( Xa=ok → fp(B,X) ; X=Xa ).
fp(A,X) ←
    system(A) → A, X=ok ;
    clause(A,B), fp(B,Xb),
    ( Xb≠ok → X=Xb ;
      query(forall,A,true) → X=ok ; X=(A←B) ). ▯
```

The program *fp* detects a false clause in a Prolog program that succeeds on a false goal. It is a simple extension to the Prolog interpreter shown in Program 3 above. $fp(A,X)$ computes the relation "$A$ is a solvable goal; if $A$ is false then $X$ is a false instance of a clause used in solving $A$, otherwise $X$=*ok*". The procedure *fp* contains two clauses. The first clause deals with conjunctive goals. It returns $(A' \leftarrow B')$ if the recursive call on any conjunct returns $(A' \leftarrow B')$, and return *ok* otherwise. The second clause deals with unit goals. If $A$ is a system predicate it executes it and returns *ok* if $A$ succeeds. Otherwise it returns $(A' \leftarrow B')$ if the recursive call of *fp* on the body of the invoked clause returned $(A' \leftarrow B')$. Otherwise it returns *ok* if the instantiated goal is tested and found true, and return the (ground instance) of the clause invoked if the result of the test is false.

A solved goal may still contain variables, which are interpreted to be universally quantified. The implementation of *query* can either instantiate the variables before querying the user, or perform a universal query. In our implementation *query(forall,A,V)* returns $V$=*true* if every instance of $A$ is true, or unifies $A$ with such a false instance and $V$ with *false* otherwise; the code is shown in Appendix II.

We examine the behavior of *fp* on the following buggy insertion sort program.

```
isort([X|Xs],Ys) ← isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs]) ← Y>X, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) ← X≤Y.
insert(X,[],[X]).
```

We first test *isort* on [2,1,3],

```
| ?- isort([2,1,3],X).

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

and get a wrong result. The tree of this computation is shown in Figure 5.

```
isort([2,1,3],[2,3,1])
        isort([1,3],[3,1])
                isort([3],[3])
                        isort([],[])
                        insert(3,[],[3])
                insert(1,[3],[3,1])
                        3>1
                        insert(1,[],[1])
        insert(2,[3,1],[2,3,1])
                2≤3
```

**Figure 5:** The computation tree of (incorrect) insertion sort

We apply *fp* to *isort*([2,1,3],[2,3,1]). The queries *fp* performs are answered by the user.

```
| ?- fp(isort([2,1,3],[2,3,1]),C).

query: isort([],[])? y.

query: insert(3,[],[3])? y.

query: isort([3],[3])? y.

query: insert(1,[],[1])? y.

query: insert(1,[3],[3,1])? n.

C = insert(1,[3],[3,1]) ← 3>1, insert(1,[],[1]))

yes
```

*fp* returned a false instance of the first clause of *insert*. Examining it shows that the arguments for the > test are exchanged. We fix that bug, and try *isort* again,

```
| ?- isort([2,1,3],X).

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

and it returns a correct output.

### 3.2.4 A lower bound on the number of queries

We evaluate diagnosis algorithms along two dimensions: their computational complexity, which reflects the machine resources they consume, and their query complexity, which reflects the number and type of oracle queries they perform during the computation. Typically, the acting oracle will be the user, therefore we put greater emphasis on optimizing the query complexity of the diagnosis algorithms, even at the expense of a reasonable increase in their computational complexity.

Length and depth of computations of the diagnosis algorithms are measured as a function of the complexity of the computation being diagnosed, as defined in Section 3.1. We ignore the cost of performing a query since we give a separate analysis of the query complexity for each diagnosis algorithm.

Using these measures, the worst-case length and depth of the single-stepping diagnosis algorithm, when diagnosing a computation of $p$ on $x$ that returns $y$, are linear in the length and depth, respectively, of the faulty computation. The maximum number of oracle queries it performs is bounded by the length of the computation.

Answering that many queries can become a tedious matter if the computation is long. We show that, in the worst case, the number of queries any diagnosis algorithm of this type requires is of order of the logarithm of the number of procedure calls in the computation. In the following section we develop an algorithm that achieves this performance.

The lower bound proof is based on an information-theoretic adversary argument, and its idea is simple: the most a query can tell us is the existence of a bug in a component of the computation tree. An "adversary bug" can choose to hide in the larger component of the tree. The best strategy against such a bug is to perform queries that split the tree into two roughly equal components. The result of such a query narrows the search space for a bug to one component, and a sequence of at most $log_2\ n$ such queries can detect the bug.

We assume that a diagnosis algorithm for incorrect procedures uses a ground oracle for $M$ and, when applied to a computation of any procedure $p$ on input $x$ that returned an output $y$ incorrect in $M$, it returns a triple $<q,u,v>$ not in $M$.

**Theorem 3.3:** Let $DA$ be a diagnosis algorithm for incorrect procedures. Then there is a program $P$ such that for any $n$ there is an interpretation $M$ and a triple $<p,x,y>$ with a computation tree of length less than or equal to $n$, for which $DA$ on $<p,x,y>$ performs at least $log_2\ n$ queries.

**Proof:** We show a particular program in which such an adversary strategy for "bug-hiding" can be implemented. Consider the program

$p(0).$
$p(s(X)) \leftarrow p(X).$

The program checks whether its input is a string composed of 0, and zero or more applications of the successor function $s$.

We define the interpretation $M$ to have all such strings of length $\leq k$ for some $k \geq 0$. For the diagnosis algorithm to find a counterexample to the program, it needs to find the exact $k$ for which $p(s^k(0))$ is in $M$, but $p(s^{k+1}(0))$ is not.

Since a positive answer to a query $p(s^i(0))$ only constrains us to choose $k>i$, and a negative answer to such query to choose $k \leq i$, it follows that for every input $X$ of size $n$ and every query strategy of $DA$ we can "hide" $k$ so that $DA$ would need at least $log_2\ n$ queries to find it. ▯

## 3.2.5 Divide-and-query: a query-optimal diagnosis algorithm

The lower-bound proof suggests an improvement over the single-stepping querying strategy: query the node $<q,u,v>$ in the computation tree that will divide the tree into two roughly equal components. If $<q,u,v>$ is in $M$, then omit the subtree rooted at that node and iterate; otherwise apply the algorithm recursively to that subtree.

We develop an algorithm that implements this querying strategy whose length and depth are linear in the length of depth of the original computation[2]. To do so we first describe a method to divide the tree.

Let $M'$ be a subset of $M$, and consider the computation tree of $p$ on $x$ that returned $y$. The *weight* of $<p,x,y>$ modulo $M'$ is defined as follows. If $<p,x,y>$ is in $M'$ its weight is 0. Otherwise, if $<p,x,y>$ is a leaf then its weight is 1. Otherwise, the weight of $<p,x,y>$ is 1 plus the sum of the weight modulo $M'$ of its sons.

Let $T$ be a computation tree whose weight modulo $M'$ is $w$. Define the *middle node* of the tree to be the leftmost heaviest node in the tree

[2]The development of this algorithm benefited from collaboration with David Plaisted.

whose weight modulo $M'$ is $\leq\lceil w/2 \rceil$. Given a computation tree with weight $w$ modulo $M'$, we can compute the middle node in length linear in $w$, by calling the following recursive procedure *fpm* with the root of the tree.

The procedure *fpm* computes the middle node of the tree and its weight. To do so it computes weight and identity of nodes as long as it is in the "lower half" of the tree, i.e. when traversing nodes whose weight is less than half the weight of the tree, and returns the heaviest node returned by a son and its weight as soon as it enters the "upper half" of the tree, i.e. traverses nodes whose weight is greater than half the weight of the tree. It operates as follows: on input $<p,x,y>$ and $W$, it searchs the computation tree of $<p,x,y>$ in post order, pruning nodes in $M'$; for each node $A$ searched it computes $Wa$, the weight modulo $M'$ of the node, and the weight and identity of the heaviest node $(B,Wb)$, returned by the recursive calls of *fpm* to $A$'s sons; if $Wa>\lceil W/2 \rceil$ then it returns $(B,Wb)$, else it returns $(A,Wa)$. The procedure *fpm* as described assumes a given computation tree; in practice the computation tree is not given but computed by an interpreter, and *fpm* is an augmentation to that interpreter, as in the Prolog implementation below.

The implementation of the divide-and-query algorithm uses this procedure; it is shown as Algorithm 2 below.

**Theorem 3.4:** Let $P$ be a program and $M$ an interpretation. If a procedure $p$ in $P$ has a computation on input $x$ of length $n$, depth $d$ and branching $b$ that returns an output $y\neq\perp$ incorrect in $M$, then the computation of Algorithm 2 applied to $<p,x,y>$ and $M'=\{\}$ has length $cn$, for some constant $c>0$, depth $d+1$, performs at most $b\log n$ queries, and returns a triple $<q,u,v>$ not in $M$ such that $q$ covers $<q,u,v>$ in $M$.

We first establish the correctness of the algorithm.

**Lemma 3.5:** Under the conditions of Theorem 3.4, if the algorithm terminates and returns $<p,x,y>$, then $p$ covers $<p,x,y>$ in $M$, but $<p,x,y>$ is not in $M$.

**Algorithm 2:** Tracing an incorrect procedure by divide-and-query

*Input*: A procedure $p$ in $P$ and an input $x$ such that $p$ on $x$ returns an output $y\neq\perp$ incorrect in $M$, and a (possibly empty) set of triples $M'\subseteq M$.

*Output*: A triple $<q,u,v>$ not in $M$ such that $q$ covers $<q,u,v>$ in $M$.

*Algorithm*: Simulate the execution of $p$ on $x$ that returns $y$, computing $w$, the weight modulo $M'$ of the computation tree. Then call a recursive procedure *fp* with $<p,x,y>$, $w$ and $M'$.

The procedure *fp*, on input $<p,x,y>$, $w$, and $M'$, operates as follows. If $w=1$ then *fp* returns $<p,x,y>$. Otherwise it applies the procedure *fpm*, defined above, which finds the heaviest node $<q,u,v>$ in the tree of $<p,x,y>$ whose weight $w_q$ modulo $M'$ is less than or equal to $\lceil w/2 \rceil$. It then queries the ground oracle whether $<q,u,v>$ is in $M$.

If the oracle answers *yes*, then *fp* calls itself recursively with $<p,x,y>$, $w-w_q$, and $M'\cup\{<q,u,v>\}$. If the oracle answers *no*, *fp* calls itself recursively with $<q,u,v>$, $w_q$, and $M'$. ▯

**Proof:** Observe that if *fp* is called with $<p,x,y>$ then $<p,x,y>$ is not in $M$. This is the input condition of the algorithm, and is preserved by the recursive calls of *fp*. The procedure *fp* returns a triple $<p,x,y>$ only if it was called with $w=1$, which implies that the weight of the sons of $<p,x,y>$ in the computation tree is 0, or, in other words, that all of its sons (if any) are in $M'$. ▯

We analyze the query- and computation-complexity of the algorithm.

**Lemma 3.6:** Under the conditions of Theorem 3.4, the computation of Algorithm 1 applied to $<p,x,y>$ and $M'=\{\}$ has length $O(n)$, depth $d+1$, and performs at most $b\log n$ queries.

**Proof:** We show that at each application of the procedure *fp* the size of the computation tree decreases by a factor of at least $1/2b$. If the oracle answers *no* to the query performed by *fp*, then *fp* calls itself recursively with the subtree rooted at the node queried; the size of this subtree is at most 1/2 of the size of the original computation tree, and the claim holds. If the oracle answers *yes*, then the node is added to $M'$. In the next computation, the weight of this node is 0, thus decreasing the weight of the computation tree by at least a factor of $1/2b$. Let $I(n)$ be the maximal number of iterations of the algorithm on a tree of weight $n$. The following recurrence relation bounds $I(n)$.

$$I(1) \leq 1$$
$$I(n) \leq 1 + I(n(2b-1)/2b)$$

We can verify by induction that $I(n)=b\log_2 n$ satisfies these inequalities. It satisfies the first inequality since $b\log_2 1$ equals 0 which is less than or equal to 1. The induction step is proved showing that the inequality

$$b\log_2 n \leq 1 + b\log_2(n(2b-1)/2b)$$

can be reduced to the inequality $\log_2(2b-1)/2b \geq -1$, which holds for $b \geq 1$.

The length of each iteration is linear in the size of the remaining computation tree. Hence the total length of the computation satisfies the following recurrence relation

$$L(1) \leq k$$
$$L(n) \leq kn + L(n(2b-1)/2b)$$

for some $k>0$, that is satisfied by the solution $L(n) \leq cn$, for some $c \geq 2b$. ▯

**Proof of Theorem 3.4:** The theorem follows from the last two lemmas. By Lemma 3.5, if the algorithm terminates then it returns a triple $<q,u,v>$ not in $M$ such that $q$ covers $<q,u,v>$ in $M$. By Lemma 3.5, the algorithm terminates, and its length and depth are as desired. ▯

We would like to point out a corollary of this theorem. If the length of the computations of the program being diagnosed is polynomial in the size of its input, then the number of queries performed by the divide-and-query diagnosis algorithm is of the order of the logarithm of the size of the input to the faulty computation, where the constant depends on the degree of the polynomial. The bugs that are harder to diagnose are those that manifest themselves only on large inputs. The query complexity of the divide-and-query algorithm suggests that the number of queries needed to diagnose such bugs would be feasible for programs that run in polynomial time.

## 3.2.6 A Prolog implementation of the divide-and-query algorithm

Program 5 implements the procedure *fpm* described above.

**Program 5:** An interpreter that computes the middle point of a computation

```
fpm(((A,B),Wab),M,W) ← !,
      fpm((A,Wa),(Ma,Wma),W), fpm((B,Wb),(Mb,Wmb),W),
      Wab is Wa+Wb,
      ( Wma>=Wmb → M=(Ma,Wma) ; M=(Mb,Wmb) ).
fpm((A,0),(true,0),W) ←
      system(A), !, A ; fact(A,true).
fpm((A,Wa),M,W) ←
      clause(A,B), fpm((B,Wb),Mb,W),
      Wa is Wb+1,
      ( Wa>(W+1)/2 → M=Mb ; M=((A←B),Wa) ). ▯
```

The first clause computes the heaviest node returned by the recursive calls on the sons, and the total weight of the node. The second clause prunes goals that are in $M'$ or are system predicates. The third clause solves unit goals by activating a clause, and also decides whether it is in the upper or lower half of the computation tree, and returns its output accordingly, as described above.

Program 6 is a direct implementation of Algorithm 2. The procedures *fp* is explained above. The procedure *false_solution* is augmented with an interface to an error handler, to be used in the systems developed below. The implementation assumes that $M'$ is represented as a set of clauses: for any goal $A$ in $M'$, there is a clause *fact(A,true)* in the database. Also, it assumes that results of system predicates are correct, hence saving some queries.

**Program 6: Tracing an incorrect procedure by divide-and-query**

```
false_solution(A) ←
      writelv(['Error: wrong solution ',A,'. diagnosing...']), nl,
      fpm((A,W),_,0), % finds W, the length of the computation
      fp(A,W,X) → handle_error('false clause',X) ;
      write('!Illegal call to fp'), nl.

fp(A,Wa,X) ←
      fpm((A,Wa),((P←Q),Wm),Wa),
      ( Wa=1 → X=(P←Q) ;
        query(forall,P,true) → Wa1 is Wa−Wm, fp(A,Wa1,X) ;
        fp(P,Wm,X) ). ▯
```

The difference between the implementation and the algorithm is that *fp* returns a false instance of a clause, rather than the false goal covered by that clause. This introduces only a minor complication to the implementation, and we find the result to be more informative to the human debugger.

Consider the following buggy insertion sort program.

```
isort([X|Xs],Ys) ← isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[X,Y|Ys]) ← X≤Y.
insert(X,[Y|Ys],[Y|Zs]) ← insert(X,Ys,Zs).
insert(X,[],[X]).
```

If we apply the single-stepping algorithm to diagnose *isort*([2,1,4,3,5,6],[6,4,5,2,3,1]) it will perform 16 queries, in a computation of length 17. In comparison, the divide-and-query algorithm needs only four queries in this case to find a false clause.

```
?– fp(isort([4,1,2,3,5,6],[1,2,3,5,4,6]),17,C).

Query: isort([2,3,5,6],[2,3,5,6])? y.

Query: insert(4,[2,3,5,6],[2,3,5,4,6])? n.

Query: insert(4,[5,6],[5,4,6])? n.

Query: insert(4,[6],[4,6])? y.

C = insert(4,[5,6],[5,4,6])←insert(4,[6],[4,6])
```

Theorem 3.4 ensures that the length and depth of the computations of the diagnosis program are linear in that of the faulty computation. We suspect that a similar claim cannot be made on the running time of the program, since the division strategy does not take into account the amount of backtracking that is needed to construct the different parts of the computation tree, but their final sizes only. However, an $O(n \log n)$ bound on the running time is easy to show: the number of iterations is bounded by the *log* of the length of the computation tree, which is bounded by $n$; and the running time of each iteration is bounded by the running time of the original faulty computation, which is $n$, since *fpm* is just a Prolog interpreter that performs some additional arithmetic operations and unifications, whose overhead is a constant function of the running time.

## 3.3 Diagnosing finite failure

For a deterministic programming language, if $y$ is a correct output for $p$ on $x$, but the computation of $p$ on $x$ terminates and returns an output different from $y$, then this output is incorrect, and the algorithms for the diagnosis of termination with incorrect output are applicable. For a nondeterministic language, however, it may happen that every computation of $p$ on $x$ terminates and returns an output correct in $M$, but no computation returns $y$. Such a program is said to *finitely fail* on

$<p,x,y>$. Finite failure needs a special treatment for a nondeterministic programming language. In this chapter we develop an algorithm for the diagnosis of finite failure.

### 3.3.1 Completeness

A program $P$ is said to be *complete* in $M$ if for every triple $<p,x,y>$ in $M$, there is a computation of $p$ on $x$ that returns $y$. If a program finitely fails on a triple in $M$, then it is not complete in $M$. Again, we define a property of procedures such that an incomplete program can be proved to contain a procedure with this property.

We say that a procedure $p$ is *complete* with respect to $M$ if for any $<p,x,y>$ in $M$, $p$ covers $<p,x,y>$ with respect to $M$; otherwise $p$ is said to be *incomplete*. Clearly, if a procedure $p$ is incomplete, say, by not covering $<p,x,y>$, then $p$ needs to be modified, as the only way (if any) in which $p$ applied to $x$ can return $y$ is by having some procedure call subordinate to $<p,x>$ return an output incorrect in $M$.

**Theorem 3.7:** Let $P$ be a program and $M$ an interpretation. If $P$ finitely fails on a triple $<p,x,y>$ in $M$, then $P$ contains an incomplete procedure.

**Proof:** We have to show that if there is a triple $<p,x,y>$ in $M$ for which every reachable computation tree of $P$ rooted at $<p,x,y>$ is finite, and there is no such complete computation tree, then $P$ contains an incomplete procedure. The proof is by induction on $d$, the maximal depth of any reachable computation tree rooted at $<p,x,y>$.

If $d=1$, then $p$ has no top level trace for $<p,x,y>$, hence it does not cover $<p,x,y>$, and since this triple is in $M$ then $p$ is incomplete.

Assume that the claim holds for $d-1$, where $d>1$ is the maximal depth of any reachable computation tree rooted at $<p,x,y>$. If no top level trace of $<p,x,y>$ is in $M$ then $p$ does not cover $<p,x,y>$, and $p$ satisfies the claim. Otherwise, consider the triples in the top level trace. For at least one such triple $<q,u,v>$, there is no computation tree; otherwise there is a complete computation tree for $<p,x,y>$, in contradiction to the assumption.

Also, the maximal depth of any reachable computation tree of $<q,u,v>$ is $d-1$, by the assumption that the maximal depth of any reachable computation tree rooted at $<p,x,y>$ is $d$. Hence the assumptions of the claim apply to $<q,u,v>$ and $d-1$, and by the inductive assumption $P$ contains an incomplete procedure. ▯

### 3.3.2 An algorithm that diagnoses incomplete procedures

As in the proof of Theorem 3.2, the proof of Theorem 3.7 also suggests an algorithm for detecting an incomplete procedure, which is described below. The algorithm uses existential queries to detect such a procedure. An *existential query* is a pair $<p,x>$; the answer to an existential query $<p,x>$ in an interpretation $M$ is the set $\{y \mid <p,x,y> \text{ is in } M\}$. By our assumption on interpretations, this set is finite.

The algorithm performs oracle computations using an oracle for $M$ that can answer existential queries. An *oracle computation* of $<p,x>$ is a computation in which every procedure call $<q,u>$ subordinate to $<p,x>$ is simulated by a call $<q,u>$ to an existential oracle for $M$, followed by a nondeterministic choice of some $v$ from the set returned by the oracle.

**Algorithm 3:** Tracing an incomplete procedure

*Input:* A triple $<p,x,y>$ in $M$ on which $p$ finitely fails.
*Output:* A triple $<q,u,v>$ in $M$ not covered by $q$.
*Algorithm:* The algorithm calls a recursive procedure *ip* with input $<p,x,y>$.
The procedure *ip* operates as follows. On input $<p,x,y>$ it tries to construct an oracle simulation of the procedure call $<p,x>$ that returns $y$, using existential queries, and while doing so it stores the top level trace that corresponds to that computation. If it fails, then *ip* returns $<p,x,y>$. If it succeeds, then it searches through the top level trace for a triple $<q,u,v>$ on which $P$ finitely fails, calls itself recursively with $<q,u,v>$, and returns the output of the recursive call. ▯

**Theorem 3.8:** Let $P$ be a program and $<p,x,y>$ a triple in $M$ on which $P$ finitely fails. Assume that any reachable computation tree of $<p,x,y>$ has length of at most $n$ and depth at most $d$, and that the maximal size of the union of the top level traces of any triple $<q,u,v>$ in any reachable computation tree rooted at $<p,x,z>$, for any $z$, is at most $b$.

Then the computation of Algorithm 3 applied to $<p,x,y>$ has length at most $dn+1$, depth at most $d+1$; it performs at most $b(d-1)+1$ queries, and returns a triple $<q,u,v>$ in $M$ not covered by $q$.

**Proof:** The proof that *ip* terminates follows from the following complexity analysis of the algorithm. It is easy to see that if *ip* terminates it returns a triple $<q,u,v>$ in $M$ such that $q$ does not cover $<q,u,v>$.

Assume that *ip* is called with $<p,x,y>$. Since $P$ has no infinite reachable computation tree rooted at $<p,x,z>$, for any $z$, it follows, by our assumption on the programming language, that it contains only finitely many reachable computation trees rooted at $<p,x,y>$; let $n$ be the maximal length, and $d$ the maximal depth of any of these trees.

We prove by induction on $d$ that the depth of the computation of *ip* is at most $d+1$, its length is at most $dn+1$, and the number of queries it performs is at most $b(d-1)$. If $d=1$, it means that there are no top level traces for $<p,x,y>$, hence *ip* returns immediately without performing any queries. Hence both the depth and the length of *ip*'s computation are $d+1=dn+1=2$, and the number of queries performed is $b(d-1)=b(1-1)=0$.

Assume that the claim holds for $d-1$, where $d>1$. In this case *ip* tries to construct an oracle simulation of the procedure call $<p,x>$ that returns $y$; to do this it has to perform at most $b$ existential queries, by the assumption that the sum of sizes of any top level traces of any triple in reachable computation trees of $<p,x,y>$ is $b$.

Following this step, *ip* either returns or calls itself recursively with some triple $<q,u,v>$ that finitely fails. By the assumption that the depth of the computation of $p$ on $x$ is $d$ and its length is $n$, the depth of any computation of a triple in the top level trace is at most $d-1$, and the maximal sum of their lengths is at most $n-1$.

By the inductive assumption, the depth of the computation of *ip* on $<q,u>$ is at most $d$ and its length is at most $(d-1)n+1$, therefore the depth of the computation of *ip* on $<p,x,y>$ is at most $d+1$, and its length is at most $(d-1)n+1+n=dn+1$.

By the inductive assumption the number of queries performed by *ip* on $<q,u>$ is at most $b(d-2)$, therefore the number of queries performed by *ip* on $<p,x>$ is $b(d-2)+b=b(d-1)$, and the claim is proved. ▯

## 3.3.3 A Prolog implementation

We relate the general discussion to logic programs, and describe a Prolog implementation of Algorithm 3. Finite failure in logic programs was studied by Apt, van Emden [7], and Clark [26]. A goal $A$ *immediately fails* in $P$ if there is no clause $A' \leftarrow B'$ in $P$ such that $A'$ unifies with $A$. A goal $A$ *finitely fails* in a program $P$ if all computations of $P$ on $A$ are finite, and each computation contains at least one goal that immediately fails. In the context of logic programs, Theorem 3.7 says that if a program $P$ finitely fails on a goal $A$ in $M$, there there is a goal $B$ in $M$ such that no clause in $P$ covers $B$. Program 7 can detect such goals. It is a direct implementation of Algorithm 3.

**Program 7:** Tracing an incomplete procedure

```
ip((A,B),X) ← !,
     ( A → ip(B,X) ; ip(A,X) ).
ip(A,X) ←
     clause(A,B), satisfiable(B) → ip(B,X) ; X=A.

satisfiable((A,B)) ← !,
     query(exists,A,true), satisfiable(B).
satisfiable(A) ←
     query(exists,A,true). ▯
```

The procedure *ip(A,X)* computes the relation "if $A$ is a finitely failing true goal, then X is an uncovered true goal". The uncovered true goal is found by tracing down the path of failing goals in the computation, on the assumption that the original goal to *ip* is true and finitely fails. Going down this path *ip* will encounter either an immediately failing goal, on which the goal *clause*(*A*,*B*), will fail, or an uncovered goal, on which the call *satisfiable*(*B*) will fail on any solution of *clause*(*A*,*B*); in either case *A* is uncovered, and is returned as output by *ip*.

The procedure *query*(*exists*,*A*,*V*) queries the user for all the true instances of *A*. It nondeterministically returns such an instance, with *V*=*true*, if such an instance exists (i.e. returns the first solution supplied by the user and backtracks if necessary), and returns *V*=*false* otherwise. Similar to the top level Prolog interpreter, the user who answers this query returns all the true instances of the goal, ending the sequence with *no*. If the system knows that a certain goal is determinate, it does not ask for more then one instance. Also, if the goal being queried is a system predicate, such as $X>Y$ and $X\leq Y$ in the example above, then *query* solves directly it rather then querying the user about it.

We demonstrate the behavior of *ip* on the following insertion sort program; in this session *isort* and *insert* are declared to be determinate, so only one answer to the existential query is necessary. Examples of diagnosing context free grammars are shown in Appendix I below, in which the programs being diagnosed are nondeterminate.

```
isort([X|Xs],Ys) ← isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs]) ← X>Y, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) ← X≤Y.
```

The program finitely fails, for example,

```
| ?- isort([3,2,1],X).

no
```

So we call *ip* on *isort*([3,2,1],[1,2,3]),

```
| ?- ip(isort([3,2,1],[1,2,3]),X).

query: isort([2,1],X)? y.
which X? [1,2]

query: insert(3,[1,2],[1,2,3])? y.

query: isort([1],X)? y.
which X? [1].

query: insert(2,[1],[1,2])? y.

query: isort([],X)? y.
which X? [].

query: insert(1,[],[1])? y.

X = insert(1,[],[1])

yes
```

And it finds that *insert*(1,[],[1]) is uncovered. We examine the two clauses for *insert*, and see that neither of them can cover this goal: their heads do not unify with it, since they expect a nonempty list in the second argument. We realize that we forgot the base clause *insert*(*X*,[],[*X*]).

An improvement of the query-complexity of *ip* can be obtained by interleaving the search for a top level trace that is in *M*, with the search for the failing goal, as mentioned above. In the best case, this improvement can be a factor of at most *b*, where *b* is the maximum number of goals in the body of a clause in the program being diagnosed. In the worst case, the query-complexity will remain the same. Program 8 contains this improvement, and is the program used in the systems developed in the following. It is also augmented with an interface to the error handler, similar to Program 6.

The procedure *ip1* interleaves the construction of the top level trace

**Program 8:** Tracing an incomplete procedure (improved)

```
missing_solution(A) ←
        writel(['error: missing solution ',A,'. diagnosing...']), nl,
        query(exists,A,true), \+solve(A,true) →
                ip(A,X), handle_error('uncovered atom',X);
        write('!Illegal call to ip'), nl.

ip(A,X) ←
        clause(A,B), ip1(B,X) → true ; X=A.

ip1((A,B),X) ← !,
        ( query(exists,A,true), ( A, ip1(B,X) ; \+A, ip(A,X) ) ).
ip1(A,X) ←
        query(exists,A,true), ( A → break(ip1(A,X)) ; ip(A,X) ). []
```

and the search for a finitely failing goal; as soon as it detects a true finitely failing goal it recursively invokes *ip*, without completing the oracle simulation. If it completes the oracle simulation without detecting such a goal it breaks, as this situation violates its input conditions. We apply the improved *ip* to the same insertion sort program,

```
| ?- ip(isort([3,2,1],[1,2,3]),X).

query: isort([2,1],X)? y.
which X? [1,2].

query: isort([1],X)? y.
which X? [1].

query: isort([],X)? y.
which X? [].

query: insert(1,[],[1])? y.
```

```
X = insert(1,[],[1])
```

and see that it needs 4 queries to detect the uncovered goal, compared to 6 queries needed by the previous program.

Theorem 3.8 bounds the length and depth of computations of *ip* as a function of the depth and length of computations of the faulty computation. A similar argument shows that the running time of *ip* is also bounded by the square of the running time of the faulty computation: the number of iterations of *ip* is bounded by the depth of the original, faulty computation, and the running time of each iteration is bounded by that of the original computation.

## 3.4 Diagnosing nontermination

One may ask: "how can we diagnose nonterminating computations, as we know they do not terminate only if we wait an infinite amount of time?" This problem does not prevent programmers from debugging such programs; a nonterminating program either exhausts the space allocated to running it (and on any existing machine this amount has some fixed upper bound), or the patience of the person waiting for its output (a quantity not proven bounded, but in the following assumed to be so).

But even if the computation exhausts one of these resources, it still does not mean that it is nonterminating; it may be that the program is not efficient enough, the person is not patient enough, or the computer is not big enough. Indeed, when the diagnosis algorithm described below is applied, it may fail to detect an error in a program that exhausted a resource, and in such a case it is up to the programmer to decide which of the three implied courses of action to take.

### 3.4.1 Termination

Our approach to the diagnosis of nonterminating programs employs a tool found useful in proving program termination [34]. Let $S$ be a nonempty set. A *well founded ordering* $\succ$ on $S$ is strict partial ordering on

$S$ that has no infinite descending sequences. That is, $\succ$ is a binary relation over $S$ which is transitive, asymmetric and irreflexive, such that for no infinite sequence $x_1$, $x_2$, ... of elements of $S$ do we have that $x_1 \succ x_2 \succ ...$

**Lemma 3.9:** A program $P$ is everywhere terminating iff there is a well founded ordering $\succ$ on the set of procedure calls such that for every computation of $P$ in which $<p,x>$ calls $<q,u>$ it is the case that $<p,x> \succ <q,u>$.

**Proof:** If there is such a well founded ordering then the depth of any computation of $P$ is finite and hence the computation terminates.

Assume that every computation of $P$ terminates. Let $d(<p,x>)$ be the maximal depth of any computation of $p$ on $x$. Define an ordering $\succ$ to be $<p,x> \succ <q,u>$ iff $d(<p,x>) > d(<q,u>)$. It is easy to see that the ordering thus defined is well founded. ▮

As in the previous two types of program misbehavior, we would like to define a property of procedures for which knowing that a program is diverging implies that it contains a procedure with that property. However, the mere fact that a procedure $p$ performed a call that violated the well-founded ordering does not mean that the code for $p$ is wrong. For example, consider the following buggy quicksort program.

```
qsort([X|Xs],Ys) ←
        partition(Xs,X,Xs1,Xs2), qsort(Xs1,Ys1), qsort(Xs2,Ys2),
        append(Ys1,[X|Ys2],Ys).
qsort([],[]).

partition([X|Xs],Y,Xs1,[X|Xs2]) ← Y < X, partition(Xs,Y,Xs1,Xs2).
partition([X|Xs],Y,[X|Xs1],Xs2) ← X ≤ Y, partition(Xs,Y,Xs1,Xs2).
partition([],X,[X],[]).

append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
append([],Xs,Xs).
```

The computation of this program on the goal *qsort*([1,2,1,3],$X$) does not terminate, as *qsort*([1,1],$Ys$) loops. The initial segment of the stack of the computation is

```
qsort([1,2,1,3],Ys)
qsort([1,1],Ys)
qsort([1,1],Ys)
qsort([1,1],Ys)
...
```

However, the problem does not lie in the *qsort* procedure, but in *partition*, since it may return an output list longer than its input list, as in

```
| ?- partition([1],1,Xs,Ys).

Xs = [1,1],
Ys = []
```

If we examine the code for *partition* (or diagnose *partition*([1],1,[1,1],[]) using *fp*) we find that the base clause is wrong, and should be *partition*([],$X$,[],[]). Hence the following definition.

**Definition 3.10:** Let $M$ be an interpretation and $\succ$ a well-founded ordering on procedure calls. A procedure $p$ is said to *diverge* with respect to $\succ$ and $M$ if it has a triple $<p,x,y>$ with a top level trace $S$ for which:

1. There is a triple $<q,u,v>$ in $S$ for which $<p,x> \not\succ <q,u>$
2. All triples in $S$ that precede $<q,u,v>$ are in $M$.

Note that knowing the well-founded ordering by itself is insufficient to diagnose an error in a nonterminating computation; one needs to know the intended interpretation as well, to verify that results of procedure calls performed before the call that violated the ordering were correct.

We say that a procedure *loops* if it calls itself with the same input it was called with. Note that any procedure that loops also diverges, provided procedure calls that precede the looping call returned a correct output, since for any well-founded ordering $\succ$, procedure $p$ and input $x$, $<p,x> \not\succ <p,x>$.

**Theorem 3.11:** Let $P$ be a program, $M$ an interpretation, and $\succ$ a well founded ordering over procedure calls. If $P$ is diverging then it contains a procedure incorrect in $M$, or a procedure that diverges with respect to $\succ$ and $M$.

**Proof:** Assume that some computation of $p$ on $x$ does not terminate. By our assumptions on the programming language stated in Section 3.1 there is a reachable computation tree with an infinite path of procedure calls. Such a path must contain two consecutive procedure calls $<p,x>$, $<q,u>$ such that $<p,x> \not\succ <q,u>$, since $\succ$ is well-founded. Consider all the procedure calls, if any, that $p$ on $x$ performed before calling $<q,u>$; if any of them returned an incorrect output, then $P$ is not partially correct, and by Theorem 3.2 $P$ contains an incorrect procedure. Otherwise, there is a top level trace of $p$ on $x$ that has a triple $<q,u,v>$, for some $v$, such that every triple that precedes $<q,u,v>$ in the trace is in $M$, and by definition $p$ diverges with respect to $\succ$ and $M$. ▯

### 3.4.2 An algorithm that diagnoses diverging procedures

Our approach to debugging nontermination is based on the assumption that even if the programmer cannot explicitly describe such a well-founded ordering, he has one such ordering in mind when writing the program, or, at the least, when presented with a looping or diverging computation can decide which of the procedure calls involved is illegal. The latter assumption is a minimal one, and must hold for a programmer to be capable of debugging his programs, with algorithmic aids or without.

Algorithm 4 below requires a ground oracle for $M$ and an oracle for $\succ$, which is a device that can answer queries of the form "is $<p,x> \succ <q,u>$?" for every procedure $p$ and $q$ in $P$. It is assumed that every computation of $p$ on $x$ has some fixed bound $d$ on its depth (not necessarily a uniform bound), which cannot be exceeded.

We have not specified how to implement the search for violation of $\succ$. One feasible approach, which is implemented in our system, is as follows. First search for a "looping segment" in the stack, that is, a segment of the form $<p,x>...<p,x>$. If such a segment is found, it must contain two

**Algorithm 4:** Tracing a diverging procedure

*Input*: A procedure $p$ in $P$, an input $x$ and an integer $d>0$ such that the depth of the computation of $p$ on $x$ exceeds $d$.

*Output*: A triple $<q,u,v>$ not in $M$ such that $q$ on $u$ returns $v$, or two procedure calls $<q,u>$, $<r,w>$ which violate $\succ$, or "*no divergence found*".

*Algorithm*: The algorithm simulates $p$ on $x$. When the depth of the computation exceeds $d$, it aborts the computation and returns its current stack of procedure calls. The algorithm then examines the stack for two consecutive procedure calls $<p,x>$, $<q,u>$ such that $<p,x> \not\succ <q,u>$. If it finds such procedure calls, it searches, using a ground oracle, for a procedure call performed by $p$ on $x$ before calling $<q,u>$ that returned an output $v$ incorrect in $M$. If such a procedure call is found, then the algorithm calls the procedure *fp* from Algorithm 2 with input $<q,u,v>$, and returns the output of *fp*. Otherwise, the algorithm returns $<p,x>$, $<q,u>$. If no violation of $\succ$ is found, the algorithm returns "*no divergence found*". ▯

consecutive procedure calls that violate $\succ$. This pair can be detected using the oracle for $\succ$. If no looping segment is found, then what is left is to search the entire stack. To detect such a pair we perform a linear search, which was found suitable for the examples we have tried. It is possible that by using a more sophisticated search technique, the query complexity of the algorithm may be improved.

### 3.4.3 A Prolog Implementation

We first describe a Prolog interpreter that accepts as input a goal and $A$ a depth-bound, and returns *true* and an instance of $A$ if it succeeds in solving $A$ without exceeding the given depth-bound, or the stack of goals of depth $d$ if an attempt to exceed this bound was encountered.

Consider the following insertion sort program.

**Program 9:** A depth-bounded interpreter

```
solve(true,D,true) ← !.
solve(A,0,(overflow,[])) ← !.
solve((A,B),D,S) ← !,
        solve(A,D,Sa),
        ( Sa=true → solve(B,D,Sb), S=Sb ; S=Sa ).
solve(A,D,Sa) ←
        system(A) → A, Sa=true ;
        D1 is D-1,
        clause(A,B), solve(B,D1,Sb),
        ( Sb=true → Sa=true ;
          Sb=(overflow,S) → Sa=(overflow,[A|S]). □
```

```
isort([X|Xs],Ys) ← isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[X,Y|Ys]) ← X≤Y.
insert(X,[Y|Ys],Zs) ← insert(X,Ys,Ws), insert(Y,Ws,Zs).
insert(X,[],[X]).
```

It does not terminate on input [2,1,3]. If we solve *isort([2,1,3],X)* using *solve*, with a depth bound of 6 we get:

```
| ?- solve(isort([2,1,3],X),6,S).

S = (overflow,[isort([2,1,3],X),isort([1,3],[1,3]),insert(1,[3],[1,3]),
        insert(3,[1],[1,3]),insert(1,[3],[1,3]),insert(3,[1],[1,3])]),
X = X
```

And we see that the program is looping. Since the diagnosis algorithm needs to check that results of calls performed "to the left" of the diverging call are correct before it concludes that a clause is diverging, it may be more efficient to store these results during the computation, and return them when a stack overflow occurs, instead of recomputing them. If so desired the last disjunct of the last clause of *solve* should be

$$Sb=(overflow,S) \rightarrow Sa=(overflow,[(A\leftarrow B)|S]).$$

Program 10 below requires this modification. It implements Algorithm 4, using the linear search technique.

**Program 10:** Tracing a stack overflow

```
stack_overflow(P,S) ←
        writel(['error: stack overflow on ',P,'. diagnosing...']), nl,
        ( find_loop(S,Sloop) → check_segment(Sloop) ;
          check_segment(S) ).

find_loop([(P←Q)|S],Sloop) ←
        looping_segment((P←Q),S,S1) → Sloop=[(P←Q)|S1] ;
        find_loop(S,Sloop).

looping_segment((P←Q),[(P1←Q1)|S],[(P1←Q1)|Sl]) ←
        same_goal(P,P1) → writel([P,' is looping.']), nl, Sl=[] ;
        looping_segment((P←Q),S,Sl).

check_segment([(P←Q),(P1←Q1)|S]) ←
        query(legal_call,(P,P1),true) →
           check_segment([(P1←Q1)|S]) ;
        false_subgoal(P,Q,P1,Q1) → false_solution(Q1) ;
        handle_error('diverging clause',(P←Q)).

false_subgoal(P,(Q1,Q2),P1,Q) ←
        Q1≠P1,
        ( query(forall,Q1,false) → Q=Q1 ; false_subgoal(P,Q2,P1,Q) ). □
```

The way *solve* and *stack_overflow* are hooked together is application dependent, and will vary between the different systems that use them. We show here part of a session with the diagnosis system, described in the next section, which uses these programs to diagnose the looping insertion sort.

```
@isort([2,1,3],X).
stack overflow. debugging isort([2,1,3],X)
insert(1,[3],X) is looping.

is (insert(1,[3],X),insert(3,[1],X)) a legal call? no.

query: insert(1,[],[1])? y.

(insert(1,[3],X)←insert(1,[],[1]),insert(3,[1],X)) is diverging.
```

We invoked the depth-bounded interpreter with the goal *isort*([2,1,3],*X*). The stack overflowed, and the stack diagnosis algorithm was applied. It found a looping segment in the stack, starting with the goal *insert*(1,[3],*X*). So it searches down this segment for a call that violates $\succ$. The violation is detected after one query. We answered negatively, since we know that the the size of the input list to *isort* should decrease as the sorting progresses. Before concluding that *isort* is diverging, it made sure that the calls performed before the diverging call, in this case *insert*(1,[1],*X*), returned a correct output, in this case *X*=[1]. Since we answered positively, the diagnosis algorithm concluded that *isort* is diverging, and provided an example for that.

## 3.5 A diagnosis system

The diagnosis system is composed of the diagnosis programs described in the previous sections, an interactive shell, and an error handling routine, which are shown in Program 11.

The diagnosis system is organized as follows. *pds* is the top level "read-solve" loop; it iterates, reading goals and solving them, until it reads *exit*. *solve_and_check* constructs the set of solutions to the input goal, and then checks them, using *check_solutions*. *check_solutions* examines the solutions; if it finds that a stack overflow occurred, or that the computation returned a solution known to be false, or failed to return a solution known to be true, then the appropriate diagnosis algorithm is

**Program 11: A diagnosis system**

```
pds ←
      nl, read('@',P), ( P=exit ; solve_and_check(P), pds ).

solve_and_check(P) ←
      bagof0((P,X),solve(P,X),S), check_solutions(P,S).

check_solutions(P,S) ←
      member((P1,(overflow,X)),S) → stack_overflow(P1,X) ;
      member((P1,true),S), fact(P1,false) → false_solution(P1) ;
      fact(P,true), \+member((P,true),S) → missing_solution(P) ;
      confirm_solutions(P,S).

confirm_solutions(P,[(P1,X)|S]) ←
      writel(['solution: ',P1, ';']),
      ( ( system(P1) ; fact(P1,true) ) → nl, confirm_solutions(P,S) ;
        confirm(' ok') → assert_fact(P1,true), confirm_solutions(P,S) ;
        assert_fact(P1,false), false_solution(P1) ).
confirm_solutions(P,[]) ←
      write('no (more) solutions.'),
      ( system(P) → nl ;
        confirm(' ok') → true ;
        ask_for_solution(P), assert_fact(P,true), missing_solution(P) ).

solve(P,X) ← see Program 9, page 61.
stack_overflow(P,X) ← see Program 10, page 62.
false_solution(P)← see Program 6, page 47.
missing_solution(P) ← see Program 8, page 55.

handle_error('false clause',X) ← !,
      writel(['error diagnosed: ',X,' is false.']), nl, plisting(X).
handle_error('uncovered atom',X) ← !,
      writel(['error diagnosed: ',X,' is uncovered.']), nl, plisting(X).
handle_error('diverging clause',X) ← !,
      writel(['error diagnosed: ',X,' is diverging.']), nl, plisting(X).
```

invoked. Otherwise, it calls *confirm_solutions*, that displays the solutions to the user, and queries him whether they are correct. The error handling procedure is trivial in this system: it just displays the results that the diagnosis algorithms return.

Below is an annotated session with this system, logging an attempt to debug a quicksort program. We challenge the reader to find the four bugs in the program below before following the rest of the diagnosis session.

We invoke the diagnosis system from Prolog, type in our initial version of quicksort, and try it out.

```
| ?- pds.
@[user].
| qsort([X|L],L0) ←
|         partition(L,X,L1,L2),
|         qsort(L1,L3), qsort(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) ← X ≤ Y, partition(L,Y,L1,L2).
| partition([],X,[X],[]).
|
| append([X|L1],L2,[X|L3]) ← append(L1,L2,L3).
| append([],L,L).
| ↑Z

user consulted   156 words     1.38 sec.

@qsort([2,1,3],X).
error: stack overflow on qsort([2,1,3],X). diagnosing...
qsort([2],X) is looping.
```

The computation overflowed the stack, and the *stack_overflow* diagnosis program is invoked; searching through the stack, it finds that *qsort*([2],*X*) calls itself recursively with the same input, i.e. is looping. Before concluding that the error is in *qsort* itself, the program checks that all calls made by *qsort* before the looping call returned correct answers.

```
query: partition([],2,[2],[])? n.
```

Indeed, it found that *partition* returned an incorrect output.

```
error: wrong solution partition([],2,[2],[]). diagnosing...
error diagnosed: (partition([],2,[2],[]) ← true) is false.
Listing of partition([],2,[2],[]):
    (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W))
    (partition([X|Y],Z,[X|V],W) ← X≤Z,partition(Y,Z,V,W))
    (partition([],X,[X],[]) ← true)
```

It then invoked the program for diagnosing incorrect solutions, which detected an error in the base clause of *partition*. We fix that that clause to be *partition*([],*X*,[],[]), and and try again.

```
@qsort([2,1,3],X).
no (more) solutions. ok? n.

query: qsort([2,1,3],X)? y.
which X? [1,2,3].
error: missing solution qsort([2,1,3],[1,2,3]). diagnosing...
```

This time the computation finitely failed; we tell the system that this is incorrect behavior, and supply the desired output. The *missing_solution* program is then invoked.

```
query: partition([1,3],2,X,Y)? y.
which (X,Y)? [1],[3].

query: qsort([1],X)? y.
which X? [1].

query: partition([],1,X,Y)? y.
which (X,Y)? [],[].
```

```
query: qsort([],X)? y.
which X? [].

error diagnosed: qsort([],[]) is uncovered.
Listing of qsort([],[]):
(qsort([X|Y],Z)←
        partition(Y,X,W,X1),qsort(W,Z1),qsort(X1,V1),
        append([X|Z1],V1,Z))
```

After performing some queries, the program found that *qsort* fails to sort the empty list. Examining the listing for *qsort*, we find that indeed we forgot to include the base clause *qsort*([],[]), in it. We fix that, and try *qsort* again.

```
@qsort([2,1,3],X).
error: missing solution qsort([2,1,3],[1,2,3]). diagnosing...

query: qsort([3],X)? y.
which X? [3].

query: append([2,1],[3],[1,2,3])? n.

error diagnosed: qsort([2,1,3],[1,2,3]) is uncovered.
Listing of qsort([2,1,3],[1,2,3]):
(qsort([X|Y],Z)←
        partition(Y,X,W,X1),qsort(W,Z1),qsort(X1,V1),
        append([X|Z1],V1,Z))
(qsort([],[])←true)
```

*qsort* failed again to return the desired output. The diagnosis algorithm found an uncovered atom, which is supposed to be covered by the main clause of *qsort*. If we examine it carefully, we see that the problem is that the call *append*([2,1],[3],[1,2,3]) is incorrect; we have inserted the partition element in front of the list of the smaller elements, rather than between this list and the list of the larger elements. We fix that clause to read:

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

and try *qsort* again.

```
@qsort([2,1,3],X).
solution: qsort([2,1,3],[2,1,3]); ok? n.
error: wrong solution qsort([2,1,3],[2,1,3]). diagnosing...
```

This time *qsort* returned a wrong solution; the system knowns that *qsort* is determinate, and knows that *qsort*([2,1,3],[1,2,3]) is correct, hence it could deduce that [2,1,3] is an incorrect output, and automatically invokes the appropriate diagnosis algorithms.

```
query: qsort([1,3],[1,3])? y.

query: partition([3],2,[],[3])? y.

error diagnosed:
        (partition([1,3],2,[],[1,3]) ← partition([3],2,[],[3])) is false.
Listing of partition([1,3],2,[],[1,3]):
   (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W))
   (partition([X|Y],Z,[X|V],W) ← X≤Z,partition(Y,Z,V,W))
   (partition([],X,[],[]) ← true)
```

An error is detected in the first clause of *partition*. We have forgot the < test in it. We include the test:

```
| partition([X|L],Y,L1,[X|L2]) ← X<Y, partition(L,Y,L1,L2).
```

And try again.

```
@qsort([2,1,3],X).
error: missing solution qsort([2,1,3],[1,2,3]). diagnosing...

error diagnosed: partition([3],2,[],[3]) is uncovered.
Listing of partition([3],2,[],[3]):
    (partition([X|Y],Z,U,[X|W]) ← X<Z,partition(Y,Z,U,W))
    (partition([X|Y],Z,[X|V],W) ← X≤Z,partition(Y,Z,V,W))
    (partition([],X,[],[]) ← true)
```

This time the program finitely fails, and the error is found to be in *partition* again. We examine the uncovered atom, and realize that the first clause is supposed to cover it; it fails because we reversed the arguments to the < test we just introduced.

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

So we fix it.

```
@qsort([2,1,3],X).
solution: qsort([2,1,3],[1,2,3]);
no (more) solutions. ok? y.

@qsort([2,1,4,3,66,2,477,4],X).
solution: qsort([2,1,4,3,66,2,477,4],[1,2,2,3,4,4,66,477]); ok? y.
no (more) solutions. ok? y.

@exit.
```

This time *qsort* behaves to our satisfaction, so we end the diagnosis session, and return to top-level Prolog.

We summarize the main points exemplified in this session:

1. Any program can be diagnosed, no matter how "buggy" it is.
2. There is no need to finish debugging "low level" procedures before "high level" procedures can be debugged (in contrast to programming by stepwise refinement), or vice versa; the program can be debugged as a whole.
3. False instances of clauses and uncovered atoms are useful clues as to how to correct bugs.

Several easy extensions to the system can make it more user-friendly, including invoking an in-core editor with the appropriate arguments by the error handler; automatic retry of a goal after an error has been detected and fixed; and a facility for the user to declare certain procedures as "correct", thus avoiding tracing their execution and querying their results.

## 3.6 Extending the diagnosis algorithms to full Prolog

The previous sections considered the application of the debugging algorithms to the "pure" part of Prolog only. Since the diagnosis algorithms were developed in an abstract setting, it should be fairly evident that they can handle any side-effect free extensions of pure Prolog. We do not know yet how to handle side-effects without resorting to a state-transition type semantics.

In the following we examine several such extensions, and point out how the diagnosis algorithms can handle them. We do not confront any conceptual problems in doing so; the major effort in modifying the diagnosis programs would be to augment every mini-interpreter they use to handle the extensions desired.

### 3.6.1 Negation

The way to diagnose negation was pointed out to me by Frank McCabe (personal communication, 1981). He observed that Algorithms 1 and 2 are dual, in a sense, and discovered that the following augmentation to *fp* and *ip* is sufficient to allow them to diagnose programs that contain negation.

*fp*(*not*(*A*),*X*) ← *ip*(*A*,*X*).

*ip*(*not*(*A*),*X*) ← *fp*(*A*,*X*).

A goal *not*(*A*) succeeds iff *A* finitely fails; hence if the goal *not*(*A*) erroneously succeeded, is means that *A* erroneously finitely failed. This justifies the clause *fp*(*not*(*A*),*X*) ← *ip*(*A*,*X*). Similarly, if *not*(*A*) erroneously failed, it follows that *A* erroneously succeeded; hence the clause *ip*(*not*(*A*),*X*) ← *fp*(*A*,*X*) is correct. The augmented *ip* and *fp* can return now either an instance of a clause or a goal; in the former, the clause is a false instance of a clause in *P*; in the latter the goal is a true goal uncovered by *P* in *M*.

We demonstrate the behavior of *fp* and *ip*, augmented with these two clauses. The version of *fp* used is the one in Program 4, and of *ip* is Program 7. The more sophisticated implementations of *fp* and *ip* in Programs 6 and 8 need more elaborate modifications to handle negation.

Consider the following program that finds an element in the symmetric difference of two lists.

*difference*(*X*,*Ys*,*Zs*) ← *member*(*X*,*Ys*), *not*(*member*(*X*,*Zs*)).
*difference*(*X*,*Ys*,*Zs*) ← *member*(*X*,*Zs*), *not*(*member*(*X*,*Ys*)).

*member*(*X*,[*Y*|*Xs*]).
*member*(*X*,[*Y*|*Ys*]) ← *member*(*X*,*Ys*).

We try it on a simple example:

| ?- difference(X,[1,2,4],[2,3]).

*no*

The program did not find any element in the difference. So we call the augmented *ip* with one of the possible solutions:

| ?- ip(difference(4,[1,2,4],[2,3]),C).

*query*: *member*(4,[1,2,4])? *y*.

*query*: *member*(4,[2,3])? *n*.

*C* = *member*(4,[2,3])←*true*

And a counterexample to the base clause of *member* is found. Although the program finitely failed, *ip* found a counterexample to a clause (by calling *fp*), rather than an uncovered goal. We modify the base clause of *member* to be *member*(*X*,[*X*|*Xs*]), try again:

| ?- difference(X,[1,2,4],[2,3]).

*X* = 1 ;

*X* = 4 ;

*X* = 3 ;

*no*

and get the correct result.

## 3.6.2 Control predicates

Control predicates with local scope, such as Prolog *if-then-else* construct "→". can be handled in a way similar to negation. Matters are more difficult with the *cut* predicate "!". Since its scope is the whole procedure (set of clauses with the same head predicate), one cannot assign the simple model theoretic semantics to individual clauses; the abstract semantics we have developed for procedures still hold, though, as it does not rely on properties of logic.

Since the abstract definition of the algorithms refer to procedures, rather then clauses, they are applicable to Prolog programs with cuts as well. The difference is in how the result of the diagnosis is interpreted. In a program with cuts, a false instance of a clause does not necessarily mean that the clause needs to be modified; it may mean that a cut is missing from some clause that preceded it, thus erroneously letting it work on a goal it was not supposed to. The same holds for uncovered goals; if a goal is uncovered, is does not necessarily mean that a clause needs to be added, or that the clause that was "supposed to" cover it needs to be modified. It may be that some clause *q* above the clause *p* that was supposed to work has a superfluous cut, which prevented *p* from being activated.

Maarten van Emden [96] suggested that there are two different uses of *cut*: One is to influence the flow of control; he calls a cut that serves this function a *red cut*. The other is to increase the efficiency of the program by preventing it from backtracking into useless paths; he calls such cuts *green cuts*. Our programming experience suggests that red cuts can almost always be subsumed by a correct use of the local constructs *if-then-else* and *not*.

If one restricts oneself to the use of green cuts, one can ignore them during the debugging process, unless the resulting program is too slow. The problem of how one distinguishes between green and red cuts remains, however.

### 3.6.3 Second order predicates

A solution to the goal *setof*($X$,$P$,$S$) can be wrong in two ways: $S$ may include a wrong solution to $P$ (an instance of $X$ for which $P$ is false), or $S$ may fail to include some correct solution to $P$ (an instance of $X$ for which $P$ is true). In the first case, the *false_solution* program should be invoked with the false instance of $P$; in the second, the *missing_solution* program should be invoked with the missing instance of $P$. Since *setof* explores all computation paths, we are guaranteed that in both cases the original computation on the wrong or missing solution terminates. If the *setof*($X$,$P$,$S$) goal does not terminate, this implies that the goal $P$ does not terminate, and the *stack_overflow* program is applicable.

The same approach can be applied to *bagof*, as long as it is perceived only as a more efficient version of *setof*, in which the multiplicity and

ordering of solutions is immaterial for correctness. If this is not the case then clauses can not be debugged individually, similarly to clauses with red *cuts*, since their order in a procedure and their multiplicity are relevant to this aspect of the behavior of *bagof*.

## 3.7 Mechanizing the oracle

In addition to optimizing the query-complexity of the diagnosis algorithms, one can alleviate some of the burden on the user by partially mechanizing the oracle. The simplest improvement is already incorporated in the implementations described above. User answers to queries are remembered within and between sessions, so that he is not asked the same query twice. Known positive facts also bias the construction of the heaviest path by the interpreter, improving the query-complexity of Algorithm 2 in another way.

Another immediate improvement can be achieved using the known negative facts. If, during a computation, a goal succeeds and returns an output already known to be incorrect, there is no need to start diagnosing the computation from the top; rather, the diagnosis algorithm should be invoked directly by the interpreter, as soon as it "traps" incorrect outputs.

An interpreter that monitors the computation, and invokes the appropriate diagnosis algorithm as soon as it detects an error is shown as Program 12.

The interpreter *msolve* is an enhancement of the depth-bounded interpreter *solve* in Program 9 above. Its first argument is the goal to be solved, the second is the depth bound. In its third argument it returns *true*, if the computation terminated correctly and no errors found, (*overflow*,$S$) if the stack $S$ overflowed, and *false* if an error occurred during the computation.

The procedure that monitors the correctness of solutions to goals is *result*. Its first clause traps stack overflow. Its second clause returns *false* if an error occurred in solving subordinate goals, or if a solution to the current goal is known to be false, or if there is a known solution to the

**Program 12: An interpreter that monitors errors**

```
msolve(A,0,(overflow,[])) ← !.
msolve((A,B),D,S) ← !,
      msolve(A,D,Sa),
      ( Sa=true → msolve(B,D,Sb), S=Sb ; S=Sa ).
msolve(A,D,Sa) ←
      system(A) → A, Sa=true ;
      D1 is D−1,
      setof0((A,B,Sb), (clause(A,B), msolve(B,D1,Sb)),R),
      result(R,A,Sa).

result(R,A,(overflow,[(A←B)|St])) ←
      member((A,B,(overflow,St)),R), !.
result(R,A,false) ←
      member((A,_,false),R), ! ;
      member((A,B,true),R), fact(A,false) , !,
         false_solution(A) ;
      fact(A,true), \+(member((A,_,true),R)), !,
         missing_solution(A).
result(R,A,true) ←
      member((A,_,true),R). ∎
```

current goal that was not found by the program. Finally, its third clause returns a solution to the goal, if no errors were detected by the first two clauses.

An interpreter that monitors errors is a bit slower than the standard interpreter, depending on how hard it is to check whether a result is known to be incorrect; but in "debugging mode" such an overhead is acceptable, and may save some human labor.

There are situations besides interactive debugging in which a monitoring interpreter may prove useful. For example in a production system in which the correctness of the output of some procedure in the

system is critical. In such a system, one can keep the "debugging mode" always turned on; that is, whenever such a procedure returns an output, it will be checked against the information available about the intended behavior of that procedure, and some kind of complaining mechanism can be established when the result is found incorrect. This use of redundant information is similar to its use in error correcting codes, and has already been suggested by Hewitt [43].

Answers to queries is one way to inform a debugging system of the intended behavior of the program, but not necessarily the most convenient or concise one. For example, the constraint

*isort(X,Y) → ordered(Y)*

is readable, easy to verify, and may save the user from answering some queries if it is known by the system. It is easy to incorporate constraints and partial specifications in the current scheme, by making the *query* procedure first consult the available information about the intended input/output behavior of the program, and ask the user only if it fails to answer the query using this information.

We found two other types of information useful: whether a procedure is *determinate*, i.e. whether it has at most one solution to any goal in a given domain, and whether it is *total*, i.e. has at least one solution for any goal in a domain. If a procedure has two solutions for a determinate goal, the diagnosis system can conclude that at least one of them is false, and after performing at most one query can invoke *false_solution*. If a procedure is total, then the interpreter can trap termination with missing output as soon as it occurs in such a procedure, and apply *missing_solution* starting at this point in the computation.

The following two clauses, added to the *result* procedure in Program 12 above, will enable it to make use of such information.

```
result([],A,false) ←
        attribute(A,total), !,
           writel(['Error trapped: no solution to ',A]), nl,
           query(exists,A,true), missing_solution(A).
result([A1,A2|R],A,false) ←
        attribute(A,determinate), !,
           writel(['Error trapped: too many solutions to ',A]), nl,
           member((A,_,_),[A1,A2|R]), query(forall,A,false), !,
              false_solution(A).
```

# Chapter 4

# INDUCTIVE PROGRAM SYNTHESIS

In this chapter we apply the diagnosis algorithms to the problem of synthesizing a program from examples of its behavior. Inductive inference was the testbed in which the diagnosis algorithms were developed, and it is the application of the algorithms with which we have the most experience.

These algorithms enable the development of an incremental inductive inference algorithm, since they can pinpoint accurately where the problem with the program being synthesized lies, and thus allow local modifications to the program to effectively correct the bug.

We survey concepts of inductive inference, and show the problem of program synthesis from examples to be a special case of program debugging. We then develop an incremental inductive inference algorithm that synthesizes logic programs from examples of their input/output behavior, and study its behavior in abstract terms and via examples of the behavior of the Model Inference System, its implementation.

## 4.1 Concepts and methods of inductive inference

### 4.1.1 Identification in the limit

One way to debug a program is to augment it with a table of patches, and add an input/output entry to the table for every input for which the program behaves incorrectly. Patching constitutes a rudimentary form of learning. It seems that patching is not satisfactory as a general approach to program debugging, as it works only if the program behaves incorrectly on a finite number of inputs only. One way to justify this claim on theoretical grounds uses the concept of *identification in the limit*, introduced by Gold [36, 37], and is defined as follows.

A *presentation* of a program $P$ is a (possibly infinite) sequence of input/output samples of $P$ in which every input in the domain of $P$ eventually appears. Assume that a debugging algorithm is given an initial program and a presentation of some target program. The debugging algorithm reads the samples one at a time, and performs modifications to the initial program as it pleases. The debugging algorithm is said to *identify the target program in the limit* if eventually there comes a time when the modifications it performs results in a program with the same input/output behavior as the target program, and it does not modify this program afterwards.

Note that within this definition, a debugging algorithm based on patching alone will not identify a program in the limit if its initial program behaves incorrectly on an infinite number of inputs. A debugging algorithm that has a fixed initial program (say, the empty program) is called an *inductive inference algorithm*. An inductive inference algorithm may be supplied with some initial information on the target program, such as program schemas; this information can be viewed as a restriction on the class of possible target programs.

### 4.1.2 Enumerative inductive inference algorithms

Gold has shown that there is no general-purpose inductive inference algorithm, that is, there is no algorithm that, given some fixed initial program, will identify any program in the limit. This implies that there is no general-purpose debugging algorithm as well. Gold showed, however,

that if we restrict the target programs to be any recursively enumerable class of programs which are everywhere terminating (i.e. compute total functions), then there exists an algorithm that can identify them in the limit. The technique he suggested, called *identification by enumeration*, is very general, and operates as follows.

Let $P_1$, $P_2$, $P_3$,... be some effective enumeration of a class of programs with the property that for every $i$ and every input $x$, the computation of $P_i$ on $x$ terminates. Call the initial program of the algorithm $P_0$, and assume that when receiving the $n^{th}$ input/output sample $<x_n,y_n>$, the conjecture of the algorithm is $P_j$, for some $j>0$. After receiving the $n^{th}$ sample, the algorithm simulates $P_j$ on $x_n$. If the result is $y_n$, the algorithm proceeds to read the next sample. Otherwise, the algorithm searches for the next program $P_k$ that follows $P_j$ on the list, with the property that the result of simulating $P_k$ on $x_i$ is $y_i$ for every $i$, $1 \leq i \leq n$.

Assume that the inductive inference algorithm thus defined is supplied with a presentation of some program $P$ on the list. Let $P_n$ be the first program on the list with the same input/output behavior as $P$. For any program $P_i$, $i<n$, there is some sample on which $P_i$ behaves incorrectly, hence the algorithm will eventually reject this conjecture, and therefore the algorithm will eventually try $P_n$. But since $P_n$ has the same behavior as $P$, the algorithm will never abandon this correct conjecture. Hence it identifies $P$ in the limit.

Gold has shown that no inference method needs fewer input/output samples to discover the target program than identification by enumeration, in the following sense. Define the *convergence point* of an inductive inference algorithm $I$ to be the sequential number of the first sample it reads after which it does not modify its conjecture. Then $I$ is *uniformly more data efficient* than $I'$ if the following two conditions hold: for any presentation of a program $P$ which $I'$ identifies, $I$ also identifies $P$ and its convergence point does not exceed the convergence point of $I'$; and there exists some presentation of some program such that the convergence point of $I'$ on this presentation exceeds the convergence point of $I$. Gold's result is that no inductive inference method is uniformly more data efficient then identification by enumeration.

Even if the target program computes a total function, it is not always easy to restrict the search space of a debugging algorithm to terminating programs. As every experienced programmer knows, even if the initial program and the target program are terminating, it happens that in the process of debugging one constructs intermediate programs which do not terminate on some input. Blum and Blum [13] suggested an approach to this problem, which is a variant of identification by enumeration. The idea is to specify in advance a complexity bound for the target program, and reject intermediate programs that happen not to run under this bound. Let $h(x,y)$ be our complexity bound. Then the new algorithm can be obtained from the identification by enumeration algorithm by restricting any simulation of $P$ on $x$ that should return $y$ to take no more than $h(x,y)$ computation steps. The resulting algorithm follows.

Again, call the initial program of the algorithm $P_0$, and assume that when receiving the $n^{th}$ input/output sample $<x_n,y_n>$, the conjecture of the algorithm is $P_j$, for some j>0. After receiving the $n^{th}$ sample, the algorithm simulates $P_j$ on $x_n$ for no more than $h(x_n,y_n)$ computation steps. If the result is $y_n$, the algorithm proceeds to read the next sample. Otherwise, the algorithm searches for the next program $P_k$ that follows $P_j$ on the list, with the property that the result of simulating $P_k$ on $x_i$ for no more than $h(x_i,y_i)$ computation steps is $y_i$ for every $i$, $1 \leq i \leq n$.

The Blums showed that this algorithm is the most powerful among inductive inference algorithm that are reliable on the partial recursive functions. A debugging algorithm is called *reliable* over a class of programs if, whenever given a presentation of a program in that class it will never converge on a buggy program. An algorithm is not required to identify a class of programs to be called reliable over that class, but simply not to pretend that it has identified a target program in that class by converging on another, buggy program. The Blums showed that for any inductive inference algorithm $I$ reliable on the class of partial recursive functions there exists a recursive function $h$, uniform in $I$, such that if $I$ identifies some program $P$ in the limit then there exists a program $P'$ with the same input/output behavior as $P$, and if $P'$ applied to $x$ returns $y$ then it does so in at most $h(x,y)$ steps.

This and other results of the Blums mark an upper bound on the power of any practical debugging algorithm that uses identification in the limit as its criterion of success. Case and Smith [23] showed that if one weakens this criteria by allowing errors in the final program, one can obtain more powerful algorithms.

The practical drawback of identification by enumeration, however, is not lack of power but inefficiency. We survey some of the approaches that have been taken to construct more practical inductive inference algorithms. Angluin and Smith [6] provide a deeper survey of the theoretical work in inductive inference, as well as of some of the more practical methods described below.

### 4.1.3 Speeding up inductive inference algorithms

One approach to obtain a more efficient inductive inference algorithm is to try to speed up the enumerative algorithm, by taking advantage of the structure of the hypotheses space. Wharton [98] used structural properties of context free grammars to speed up inference by enumeration. He found a set of tests that a grammar for a given sample should pass. If a grammar fails to pass such a test, then any superset of this grammar also fails this test. Wharton describes a method that detects a subset of a grammar that fails a test, and leaps in the enumeration to the next grammar that does not include this subset. He gives empirical evidence to the utility of such a procedure. A comparison of the performance of the Model Inference System with Wharton's grammatical inference system is given in Section 4.6.

Another approach to speeding up enumerative inference algorithms was studied by Biermann et al. [10, 11, 12]. They have considered the problem of speeding up the enumerative algorithm by using program traces, rather then input/output samples. They showed that high performance gains can be achieved, but the resulting algorithms were still impractical for synthesizing complex programs.

One source of the inefficiency of the identification by enumeration algorithm that it is not incremental. The algorithm does not try to modify a refuted hypothesis, but abandons it altogether and looks for the next hypothesis in an arbitrarily ordered list. One way to define the desired property is as follows.

**Definition 4.1:** We say that an inductive inference algorithm is *incremental* if is satisfies the following two conditions:

1. Its conjectures are sets (of grammar-rules, logical axioms, state-transitions, etc.).
2. For every $i \leq j \leq k$, and $p$, if $p$ is in $P_i$ and is not in $P_j$, then it is not in $P_k$ either.

In other words, an incremental algorithm never includes in its conjecture an element that was once removed from it.

It seems that being incremental is a key property to any practical inductive inference or debugging algorithm. The incremental algorithms described below also make use of the structure of the hypotheses space to prune the search.

Knobe and Knobe [53] suggested a method for the incremental construction of a context free grammar from samples of strings in some unknown context free language $L$. Their method is based on searching the space of possible grammars from general to specific. At any given moment their algorithm has some partial grammar. The algorithm reads the samples one at a time, and if it finds that it cannot generate some string in $L$, it looks for the most general production (under a built-in criterion of generality) whose addition to the grammar will enable the derivation of the string to go through. It then generates some strings using the new grammar, using a probabilistic technique, and queries the user, or teacher, whether they are legal or not. If no illegal string is generated, it assumes that the production added is correct. Although their algorithm is incremental, it is fairly order dependent, as they do not have a general strategy for debugging incorrect grammars, but use a heuristic that resembles patching.

Mitchell [60] developed an incremental algorithm for concept learning. The hypothesis space, which is the set of patterns in a pattern language, is partially ordered according to generality. A pattern $M_1$ is *as general as* $M_2$ if $M_1$ matches all the instances than $M_2$ matches. The algorithm reads

instances and non-instances of the target pattern, and maintains two sets of patterns, the set $G$ of most general patterns than match all positive instances but no negative instances of the patterns, and a set $S$ of most specific patterns than match all positive instances and no negative instance. The set of patterns that lie between the sets $G$ and $S$ is called the *version space*. The algorithm converges when the version space is a singleton. It was applied in the context of Meta-DENDRAL [17], a system that inductively synthesizes rules for determining the molecular structure of the samples of organic chemicals from empirical data about them.

In [82, 84] we describe an incremental algorithm that infers universally quantified first order theories from ground (variable-free) facts, based on a search strategy similar to Mitchell's [60]. The logical axioms are partially ordered by the *refinement relation*, which is reminiscent of the subsumption relation, studied by Plotkin in the context of machine learning [67, 68, 69]. The set of most general axioms which are not longer than some parameter $d$ and have not refuted so far by the facts is maintained as the conjecture of the algorithm. If these axioms are ever discovered to imply a negative fact (a ground sentence known to be false) then an error detecting algorithm, called the *contradiction backtracing algorithm* is invoked, leading to the detection of at least one false axiom in the conjecture. The diagnosis algorithms developed in Chapter 3 grew out of the experience of implementing the contradiction backtracing algorithm to debug Prolog programs. If the axioms are discovered not to imply some positive fact, then the parameter $d$ is incremented. Since the question of whether a set of universally quantified sentences imply a ground sentence is undecidable, a complexity bound is used to limit the resources allocated to this check, in much the same way as in the Blums' enumerative algorithms. The inductive synthesis algorithm and system developed below are the result of specializing this algorithm to logic programs.

A different direction towards provably efficient inductive inference algorithms was pursued by Angluin [2, 3] and Crespi-Reghizzi [29, 30], among others. The idea is to look at inductive inference problems in a restricted domain, and devise specialized, efficient algorithms for them. Crespi-Reghizzi et al. [29, 30] describe efficient algorithms for finding parse trees from an unknown grammar, given samples of unlabeled parses of strings. Angluin describes in [2] a polynomial algorithm for inferring patterns, combined of constant and variable symbols, and in [3] a polynomial algorithm for inferring a class of languages called the k-reversible languages, which are a subset of the regular languages.

The problem of finding the minimal finite automaton compatible with a given sample of strings, marked as *in* and *out* of some unknown regular language was shown to be *NP*-complete by Gold [38]. Angluin [4] showed that if $n$, the number of states of the canonical acceptor for the language is known, and the subset of the sample marked *in* exercises all transitions in that acceptor, then there is an algorithm that using queries can find the acceptor in time polynomial in $n$ and the size of the given sample.

Other approaches to inductive inference and concept learning were studied by Michalski [32, 59], Langley [55], Young, Plotkin and Lintz [103], Winston [100], and Brazdil [15], among others. A comparative survey of some of this work was done by Bundy and Silver [20].

### 4.1.4 Synthesis of Lisp programs from examples

The target language in most of the work done on program synthesis from examples is Lisp. A survey of this work is given by Smith [89]. Hardy [41] and Shaw, Swartout and Green [87] describe heuristic methods for synthesizing Lisp programs from a given example and a built-in program scheme. The system of Shaw et al. is interactive, and may query the user further about the target program, in order to distinguish between plausible alternatives. The two systems constructed a program on the basis of one example.

A more systematic approach was taken by Summers [92, 93]. Summers's approach is based on finding recurrence relations between successive input/output pairs. One interesting property of his method is that it can introduce auxiliary variables when needed. His method was studied and extended by Angluin [5] Guiho, Jouannaud and Kodratoff [50, 51, 52], among others. Most of this work concentrated on developing special purpose algorithms that detect recurrence relations between the input/output samples, and hence results in non incremental

algorithms.

Under the assumptions made by Summers, the input/output samples given to his algorithm effectively convey the information in a trace. Siklossy and Sikes [88] studied the synthesis of robot programs from traces of the robot actions. Biermann [9] applied his method for synthesis from traces to Lisp. Again, the enumerative character of his approach limited the practicality of the system.

A comparison of the performance of the Model Inference System with Summers's and Biermann's systems is given in Section 4.6.

## 4.2 An algorithm for inductive program synthesis

The inductive synthesis algorithm uses the diagnosis algorithms as subroutines, and behaves as follows: it reads in samples of the behavior of the target program, one at a time. If the program is found to behave incorrectly on some input, it invokes the appropriate diagnosis algorithm that detects a bug in the program, and modifies the program according to the result of the diagnosis.

Although the algorithm can be described in general terms, we find that the strategy it uses to prune the search relies heavily on properties of logic, therefore they may not generalize to other programming languages the same way the diagnosis algorithms do. Hence we describe the algorithm in logic programming terms.

Algorithm 5 below uses the following notions. A *fact* about an interpretation $M$ is a pair $<A,V>$ where $A$ is a variable-free goal and $V$ is *true* if $A$ is in $M$, *false* otherwise. Let $S$ be a set of facts. We say that a goal $A$ is *true in* $S$ if $<A,true>$ is in $S$, and that $A$ is *false in* $S$ if $S$ has a fact $<A,false>$. A behavior of a program is said to be *totally correct* with respect to $S$ if it succeeds on any goal true in $S$, and finitely fails on any goal false in $S$. We refer to the set of facts that have been read by the algorithm at a particular point in time, together with the facts queried by the diagnosis algorithms at that time, as the set of *known facts*.

Algorithm 5 is actually an algorithm schema, as its component that

**Algorithm 5: Inductive program synthesis**

*Given*: A (possibly infinite) ordered set of clauses $L$,
an oracle for an interpretation $M$,
an oracle for a well-founded ordering $\succ$ on the domain of $L$,
and a definition of $X$, the parameterized interpretation.

*Input*: A (possibly infinite) list of facts about $M$.

*Output*: A sequence of programs $P_1$, $P_2$,... in $L$ each of which
is totally correct with respect to the known facts.

*Algorithm*:

```
set $P$ to be the empty program.
let the set of marked clauses be empty.
repeat
  read the next fact.
  repeat
    if the program $P$ fails on a goal known to be true
      then find a true goal $A$ uncovered by $P$ using Algorithm 3;
      search for an unmarked clause $p$ in $L$ that covers $A$ in $X$;
      add $p$ to $P$.
    if the program $P$ succeeds on a goal known to be false
      then detect a false clause $p$ in $P$ using Algorithm 2;
      remove $p$ from $P$ and mark it.
  until the program $P$ is totally correct
  with respect to the known facts.
output $P$.
until no facts left to read.

if the depth of a computation of $P$ on some goal $A$ exceeds $h(A)$,
  then apply Algorithm 4, the stack overflow diagnosis algorithm,
  which either detects a clause $p$ in $P$ that is diverging with respect
  to $\succ$ and $M$, or a clause $p$ in $P$ that is false in $M$;
  remove $p$ from $P$, mark it, and restart the computation on $A$. ▯
```

searches for a covering clause has an interpretation $X$ as a parameter. To instantiate this scheme one has to substitute some concrete interpretation for $X$. Different choices of $X$ and their effects on the behavior of Algorithm 5 are explored in Section 4.4. We first investigate aspects of the algorithm that are independent of this choice; for the sake of concreteness, the reader may assume during this discussion that $X=M$.

## 4.2.1 Limiting properties of the algorithm

As suggested in the discussion of identification in the limit above, to get a full grasp of the behavior of an inductive inference algorithm one has to study its limiting properties. In this section we investigate the limiting behavior of Algorithm 5 on presentations of interpretations.

**Definition 4.2:** A *presentation* $S$ of an interpretation $M$ for a language $L$ is an infinite sequence of facts about $M$ such that for any variable-free goal $A$ in the Herbrand base of $L$, if $A$ is true in $M$ then $<A,true>$ is in $S$, otherwise $<A,false>$ is in $S$.

For this investigation to be fruitful we cannot assume a fixed bound on the stack space, lest we restrict ourselves to finite classes of interpretations only. Therefore we assume that the depth-bound on computations in Algorithm 5 varies as a function of the goal being solved. More precisely, we assume a given computable function $h$ from goals to integers, and require that for every goal $A$ the depth of any computation on $A$ should not exceed $h(A)$. We say that a program $P$ is *h-easy* iff for any goal $A$ in $H(P)$, the depth of any computation of $P$ on $A$ is at most $h(A)$.

We associate with any depth-bound $h$ a well-founded ordering $\succ$, to be used by the algorithm that diagnoses stack overflow. For any two goals $A$ and $B$, we say that $A \succ B$ iff $h(A)>h(B)$. It is easy to see that if the depth of a computation of $P$ on $A$ exceeds $h(A)$ then the computation also violates $\succ$, as defined in page 58.

**Definition 4.3:** Let $L$ be an ordered set of clauses, $M$ an interpretation and $\succ$ a well-founded ordering on the domain of $L$. Assume that Algorithm 5 is given a presentation of $M$.

We say that the algorithm *converges* on this presentation if eventually there comes a time when it outputs some program $P$ and does not output a different program afterwards. We say that the algorithm *identifies $M$ in the limit* if it converges to a program that is totally correct in $M$.

For each of the search strategies described in Section 4.4 below we prove an *identification in the limit theorem,* which defines conditions under which Algorithm 5, equipped with that search strategy, identifies a target program in the limit. However, the reliability of the algorithm can be proved in a more general setting.

**Theorem 4.4:** Assume that Algorithm 5, applied to a presentation of an interpretation $M$, eventually reads in every fact, and converges to some program $P$. Then $P$ is totally correct in $M$.

To prove the theorem, we show that if the algorithm eventually reads in all the facts, then the following facts hold:

1. The algorithm does not converge to an incorrect program.
2. The algorithm does not converge to an incomplete program.
3. The algorithm does not converge to a program that is not $h$-easy.

Together these facts imply the theorem.

**Lemma 4.5:** Under the assumption of Theorem 4.4, every false clause included in $P$ by Algorithm 5 eventually gets marked.

**Proof:** Assume that $p=A\leftarrow B_1,B_2,...,B_n$ is a false clause that is included at some stage in $P$. Since it is false there is a substitution $\theta$ such that $A\theta$ is not in $M$, but $B_i\theta$, $1\leq i\leq n$ are in $M$. Consider the first time in which the algorithm leaves the inner *repeat* loop when it knows that $A\theta$ is false and all the $B_i\theta$ are true; such a time must come by the assumption that the algorithm eventually reads in every fact. By that time $P$ succeeds on all the

$B_i\theta$ and fails on $A\theta$, which implies that $P$ does not include $p$; but since $p$ was in $P$, it follows that $p$ got marked. []

**Lemma 4.6:** Under the assumption of Theorem 4.4, every program Algorithm 5 converges to is $h$-easy.

**Proof:** Assume by way of contradiction that the algorithm converges to a program $P$ that is not $h$-easy. If $P$ is not correct in $M$, then a false clause in it eventually will get marked, by Lemma 4.5.

If $P$ is correct in $M$ but not $h$-easy, then there is a fact $<A,V>$ in the presentation of $M$ and a computation of $P$ on $A$ whose depth exceeds $h(A)$. By assumption the algorithm eventually reads the fact $<A,V>$. When trying to solve $A$, a stack overflow would occur, the stack overflow diagnosis algorithm would be invoked, and a diverging clause in $P$ would get marked. Both cases contradict the assumption that the algorithm converges to $P$. []

**Lemma 4.7:** Under the assumptions of Theorem 4.4, every program Algorithm 5 converges to is complete in $M$.

**Proof:** Assume to the contrary that the algorithm converges to a program $P$ which is incomplete for $M$. Let $A$ be a goal in $M$ that is not in $M(P)$. Consider the first time the algorithm outputs $P$ after reading in the fact $<A,true>$; such a time must come since the algorithm eventually reads in every fact. Since $P$ does not solve $A$, there can be two cases:

1. The depth of a computation of $P$ on $A$ exceeds $h(A)$. In this case the stack overflow diagnosis algorithm is invoked, resulting in removing a clause from $P$.
2. $P$ finitely fails on $A$ without exceeding the depth bound $h(A)$. In this case the algorithm for diagnosing finite failure is invoked, resulting in finding a goal $B$ in $M$ uncovered by $P$, followed by adding to $P$ a clause that covers $B$.

In both cases $P$ is modified, in contradiction to the assumption that the algorithm converges to $P$. []

Together, the last three lemmas prove Theorem 4.4. The following lemma provides a sufficient condition for Theorem 4.4 to show that Algorithm 5 identifies an interpretation in the limit. We define $L_n$ to be the first $n$ clauses of $L$.

**Lemma 4.8:** Assume that Algorithm 5 is applied to a presentation of an interpretation $M$, and there is an $n>0$ such that whenever the algorithm searches for an unmarked clause in $L$ that covers a goal in $X$, it finds such a clause in $L_n$. Then the algorithm identifies $M$ in the limit.

**Proof:** To prove the lemma, we show that under its assumptions

1. The algorithm converges.
2. The algorithm eventually reads in every fact.

These two facts, together with Theorem 4.4, imply that the algorithm identifies $M$ in the limit.

By the assumption of the lemma, every program it outputs is a subset of $L_n$, hence it can output only finitely many different programs. Since the algorithm is incremental, it never returns to a program it once abandoned. Hence the algorithm converges.

To show that the algorithm eventually reads in every fact, we show that the inner *repeat* loop terminates. Each iteration executes the program on finitely many goals, possibly invokes a diagnosis algorithm, and possibly searches for an unmarked clause that covers a goal. The execution of the program on a goal terminates since it uses a depth bounded interpreter. The diagnosis algorithms were proved to terminate in Chapter 3. The search for a covering clause terminates by the assumption of the lemma.

There are only finitely many iterations since each iteration either marks a clause or adds a clause to $P$. Both can happen at most $n$ times by the assumption that only clauses in $L_n$ are included in the program. ▯

When describing the different search strategies in Section 4.4, we specify conditions under which the assumption of Lemma 4.8 holds, and by doing so prove an identification in the limit theorem for these strategies.

### 4.2.2 Complexity of the algorithm

We analyze the length of computations of Algorithm 5, as a function of the sum of sizes of facts it has seen. We say that the a program $P$ works in length $l(n)$ if the length of any partial computation tree of any goal $A$ of size $n$ in $H(P)$ is at most $l(n)$. We say that the algorithm works in length $l(n)$ if the length of its computation until it requests the next fact is at most $l(n)$, where $n$ is the sum of sizes of facts known at the time of the request.

We show that the length of computations of Algorithm 5 is dominated by two factors: the running time of the programs it synthesizes, and the number of candidate clauses it tries; if they are both polynomial, in the sense explained below, then the algorithm works in polynomial length.

**Definition 4.9:** Let $L$ be a language and $M$ an interpretation of $L$. Then $L$ is said to have a candidate space of $p(n)$ with respect to $M$ if for any goal $A$ in $M$ of size $n$, the number of clauses in $L$ that cover $A$ in $M$ is at most $p(n)$.

**Theorem 4.10:** Assume that Algorithm 5 is equipped with a search algorithm over $L$ with a search space of $n^{k1}$, for some $k1>0$, and that any intermediate program constructed by Algorithm 5 works in length $n^{k2}$, for some $k2>0$. Then Algorithm 5 works in length of at most $cn^{2k1+k2}$, for some $c>0$.

**Proof:** In the worst case, every clause in the search space for any positive fact will be tried (i.e. added to $P$). We apportion the different activities of the algorithm to the clauses tried.

Testing a program against the current set of facts is apportioned to the last clause added or removed from the program; thus for each clause, we charge executing the current program against each fact at most twice.

Invoking Algorithm 2 is charged to the clause the algorithm detects to be false; such a clause is then marked, and is never included in $P$ afterwards, hence this can happen once for each clause. By Theorem 3.4, the length of computations of the diagnosis algorithm is linear in the length of the computation it diagnoses, which is at most $n^{k1}$ by assumption.

When Algorithm 3 is invoked on a goal $A$, it finds an uncovered goal, for which the search algorithm suggests a clause that covers it; we charge running Algorithm 3 to the clause found; it is at most the square of the length of the longest computation tree of $A$. By the assumption that programs in $L$ work in polynomial length, stack overflow will not occur, hence Algorithm 4 will not be invoked.

Summing up the length of computations apportioned to a clause, we get an upper bound of $4n^{2k1}$. Summing up over the clauses searched we get an upper bound of $4n^{2k1+k2}$. ▯

Note that if all the programs generated by the algorithm are deterministic, then Algorithm 2 can be used instead of Algorithm 3 to diagnose termination with undefined output, and a bound of $O(n^{k1+k2})$ can be obtained. Since length of computations of deterministic programs is also their running time, we obtain in the deterministic case a polynomial bound on the running time of the algorithm.

The analysis treats the algorithm that searches for a candidate clause as a black box; a separate discussion of the complexity of the search algorithm we use is done in Section 4.5. That section also provides an example of a language for which such polynomial bounds hold.

## 4.3 The Model Inference System

One day, a merchant came to the studio of an artist and asked him to paint a picture of a fish. The artist agreed. The merchant came back one week later inquiring about his picture. The artist told him that the picture was not yet ready. The merchant came back the following week, but discovered the picture was still not ready. After coming back another week later and finding the picture not ready the merchant decided that he would never get his picture.

Three years later the merchant met the artist in the market and asked what had become of the picture of the fish. The artist told him that he had finished the picture just that morning and that he would show it to the merchant if he came to the studio. They arrived at the studio and on the easel was a blank piece of paper. The artist took a brush, made three quick strokes, and had a beautiful picture of a fish. Why, the merchant asked, had it taken the artist so long to produce the picture if it only took him a moment to make? The artist walked over to a door and opened it. Out poured ten thousand torn-up pictures of fish.

— a Zen parable

We describe a Prolog implementation of the inductive synthesis algorithm, called for historical reasons the Model Inference System [82, 85]. Originally, the system was conceived of as an implementation of the general inductive inference algorithm described in [82, 84], which infers first order theories from facts. It was a later realization that the inductive inference algorithm, restricted to infer Horn-clause theories, actually synthesizes Prolog programs from examples; this realization led to the focus of the current research.

The version of the Model Inference System described here is simpler and cleaner than previous ones, but apparently comparable to them in power and efficiency. Similar to the diagnosis system, it is constructed by augmenting the diagnosis programs with an interactive shell and an error handler. Algorithm 5 is diffused between these three components, hopefully in a natural way. The Model Inference System, excluding the diagnosis component and the search component, is shown as Program 13 below.

The procedure ***mis*** implements the outermost ***repeat*** loop of Algorithm 5. In addition to facts, it accepts as input the atom ***check***, on which it checks the consistency of the current set of facts with the current program. ***check_fact(X)*** checks that the current program performs correctly on all facts (X,V), and implements the inner ***repeat*** loop and the two ***if*** statements. ***solve(P)*** solves the goal $P$ using the depth bounded interpreter, and handles the exception of stack overflow, by invoking the ***stack_overflow*** diagnosis program. ***handle_error*** handles errors found by the diagnosis algorithms: if a clause is found false or diverging then it is retracted; if a goal is found to be uncovered, the program that searches for a covering clause is invoked, and the clause found is added to the current program. Marking a clause is done implicitly, by recording the facts that refute a clause or the ones that imply it is diverging.

**Program 18: The Model Inference System**

```
mis ←
        nl, ask_for('Next fact',Fact),
        ( Fact=check → check_fact(_) ;
          Fact=(P,V), (V=true ; V=false) → assert_fact(P,V), check_fact(P) ;
          write('! Illegal input'), nl ),
        mis.

check_fact(P) ←
        write('Checking fact(s)...'), ttyflush,
        ( fact(P,true), \+solve(P) →
            nl, missing_solution(P), check_fact(_) ;
          fact(P,false), solve(P) →
            nl, false_solution(P), check_fact(_) ;
          write('no error found.'), nl ).

solve(P) ←
        solve(P,X),
        ( X=(overflow,S) → stack_overflow(P,S), solve(P) ; true ).

solve(P,X) ← see Program 9, page 61.
false_solution(P)← see Program 6, page 47.
missing_solution(P) ← see Program 8, page 55.
stack_overflow(P,X) ← see Program 10, page 62.

handle_error('false clause',X) ←
        writel(['error diagnosed: ',X,' is false.']), nl,
        retract(X), plisting(X).
handle_error('uncovered atom',X) ←
        writel(['error diagnosed: ',X,' is uncovered.']), nl,
        search_for_cover(X,C), assert(C), plisting(X).
handle_error('diverging clause',X) ←
        writel(['error diagnosed: ',X,' is diverging.']), nl,
        retract(X), plisting(X).

search_for_cover(X,C) ← see Program 17, page 124.
```

We demonstrate the performance of the system on a simple example, making it infer a program for concatenating an element to a list. The purpose of the example is to illustrate the control flow of the Model Inference System. User input is always preceded by a question mark, all the rest is the system's output, some of which the reader may recognize from the session with the diagnosis system.

```
| ?- mis.

Next fact? concat(a,[b],[b,a]),true.
Checking fact(s)...
Error: missing solution concat(a,[b],[b,a]). diagnosing...
Error diagnosed: concat(a,[b],[b,a]) is uncovered.

Searching for a cover to concat(a,[b],[b,a])...
Declare concat(X,Y,Z)? declare(concat(+x,+[x],-[x]),[total,determinate]).
Procedures called by concat(X,Y,Z)? concat(_,_,_).
```

An error is found and diagnosed; the diagnosis was easy, since the program for *concat* is empty. The search algorithm is then invoked. It uses the adaptive search strategy, discussed below. The system needs some information to create the candidate clauses. We declared the type and the input/output mode of *concat*, told the system that it is a total and determinate procedure, and that the only procedure it calls is itself.

```
Checking: (concat(X,[Y|Z],[Y,X|Z])←true)
Found clause: (concat(X,[Y|Z],[Y,X|Z])←true)
   after searching 19 clauses.
Listing of concat(X,Y,Z):
   (concat(X,[Y|Z],[Y,X|Z])←true).

Checking fact(s)...no error found.
```

The system found a clause that covers the facts. The clause returns in its output list the input list with the input element inserted its second position. The clause is correct for input lists of length one only. We give it a more complex example.

```
Next fact? concat(a,[c,b],[c,b,a]),true.
Checking fact(s)...
Error: missing solution concat(a,[c,b],[c,b,a]). diagnosing...
Error diagnosed: concat(a,[c,b],[c,b,a]) is uncovered.

Searching for a cover to concat(a,[c,b],[c,b,a])...
Checking: (concat(X,[Y|Z],[Y|V])←concat(X,Z,V))
Found clause: (concat(X,[Y|Z],[Y|V])←concat(X,Z,V))
   after searching 31 clauses.
Listing of concat(X,Y,Z):
   (concat(X,[Y|Z],[Y,X|Z])←true).
   (concat(X,[Y|Z],[Y|V])←concat(X,Z,V)).

Checking fact(s)...no error found.
```

The initial information concerning type mode and called procedures restricts the search space for clauses, hence the system needed to search only 31 clauses to find a correct recursive clause for *concat*. We are not sure whether the program is correct or not, so we break to Prolog and test it.

```
Next fact? break.

[ Break (level 1) ]
| ?- concat(a,[c,b],X).

X = [c,a,b] ;

X = [c,b,a] ;

[1] no
| ?- ↑Z
[ End Break (level 1) ]
```

We found that the program is still incorrect, so we provide the system with the appropriate fact.

```
Next fact? concat(a,[c,b],[c,a,b]),false.
Checking fact(s)...
Error: wrong solution concat(a,[c,b],[c,a,b]). diagnosing...
Error diagnosed: (concat(a,[c,b],[c,a,b])←true) is false.
Listing of concat(X,Y,Z):
   (concat(X,[Y|Z],[Y|V])←concat(X,Z,V)).
```

A counterexample to the base clause was found, and the clause was removed.

```
Checking fact(s)...
Error: missing solution concat(a,[b],[b,a]). diagnosing...

Query: concat(a,[],[a])? y.
Error diagnosed: concat(a,[],[a]) is uncovered.
```

After the wrong base clause is removed, the program failed on one of the examples; the diagnosis algorithm was applied, and diagnosed the error: a base clause for *concat* is missing.

```
Searching for a cover to concat(a,[],[a])...
Checking: (concat(X,Y,[X|Y])←true)
Found clause: (concat(X,Y,[X|Y])←true)
   after searching 7 clauses.
Listing of concat(X,Y,Z):
   (concat(X,[Y|Z],[Y|V])←concat(X,Z,V)).
   (concat(X,Y,[X|Y])←true).

Checking fact(s)...
Error: wrong solution concat(a,[c,b],[c,a,b]). diagnosing...
```

The system found an unrefuted clause that covers *concat(a,[],[a])*, but after incorporating it into the program, it now succeeds on a goal known to be false. The diagnosis algorithm for wrong solutions is then applied.

*Query*: *concat*(*a*,[*b*],[*a*,*b*])? n.
*Error diagnosed*: (*concat*(*a*,[*b*],[*a*,*b*])←*true*) *is false*.
*Listing of concat*(*X*,*Y*,*Z*):
  (*concat*(*X*,[*Y*|*Z*],[*Y*|*V*])←*concat*(*X*,*Z*,*V*)).

*Checking fact*(*s*)...
*Error*: *missing solution concat*(*a*,[*b*],[*b*,*a*]). *diagnosing*...
*Error diagnosed*: *concat*(*a*,[],[*a*]) *is uncovered*.

The base clause of *concat* was discovered to be false. After its removal, the program finittely fails on *concat*(*a*,[*b*],[*b*,*a*]). The diagnosis algorithm for missing solution is applied, and it detects that *concat*(*a*,[],[*a*]) is uncovered. No user query was performed during the diagnosis, since *concat*(*a*,[],[*a*]) is already known to be true.

*Searching for a cover to concat*(*a*,[],[*a*])...
*Checking*: (*concat*(*X*,*Y*,[*X*|*Y*])←*true*)
*Refuted*: (*concat*(*a*,[*b*],[*a*,*b*])←*true*)
*Checking*: (*concat*(*X*,[],[*X*])←*true*)
*Found clause*: (*concat*(*X*,[],[*X*])←*true*)
  *after searching* 9 *clauses*.
*Listing of concat*(*X*,*Y*,*Z*):
  (*concat*(*X*,[*Y*|*Z*],[*Y*|*V*])←*concat*(*X*,*Z*,*V*)).
  (*concat*(*X*,[],[*X*])←*true*).

*Checking fact*(*s*)...*no error found*.

and after one more iteration the system found the correct base clause. It needed the following facts:

*concat*(*a*,[*b*],[*b*,*a*]), *true*.
*concat*(*a*,[*c*,*b*],[*c*,*b*,*a*]), *true*.
*concat*(*a*,[*c*,*b*],[*c*,*a*,*b*]), *false*.

*concat*(*a*,[],[*a*]), *true*.
*concat*(*a*,[*b*],[*a*,*b*]), *false*.

The first three were supplied by the user, the last two were asked for by the diagnosis algorithms. The whole session took 8 CPU seconds on the DECsystem 2060.

More examples of the behavior of the system are given in Appendix I.

## 4.4 Search strategies

We describe three search strategies for Algorithm 5, that result from different choices of $X$, the parameterized interpretation of the search algorithm, and investigate the effect of their incorporation on the behavior of Algorithm 5.

Recall the setting. The search algorithm assumes an ordered set of clauses $L$, some of them are marked; it is given a variable-free goal $A$, and has to find an unmarked clause $p$ in $L$ that covers $A$ in $X$. A clause $A' \leftarrow B_1, B_2, \ldots, B_n$ covers a goal $A$ in an interpretation $M$ if there is a substitution $\theta$ such that $A\theta = A'\theta$, and $B_i\theta$ are in $M$, for $1 \leq i \leq n$.

### 4.4.1 An eager search strategy

The most obvious choice for $X$ is $M$, the interpretation for which Algorithm 5 is trying to synthesize a program. We call the resulting search strategy *eager search*. The implication of this choice is that for each clause encountered by the search algorithm whose head unifies with the given goal, existential queries will be posed to $M$'s oracle, to find whether the clause does cover this goal. This strategy is the most powerful of the three, but also the most wasteful in terms of human (oracle) resources, as explained below.

The advantage of the eager search strategy is that it is order independent. The following example will give some intuitive basis for this claim. Assume that the algorithm is supplied with the fact *<member(a,[b,a]),true>*, that should be solved using the clause *member(X,[Y|Z])←member(X,Z)*, but was not yet supplied with the fact *<member(a,[a]),true>*. The eager search algorithm will eventually encounter the above clause when searching for a cover to *member(a,[b,a])*. It will then unify the head of the clause with the goal, and test whether the resulting instance of the body of this clause, which is *member(a,[a])*, is satisfiable in *M*, using existential queries. In other words, it will construct an instance of a rule even if no such instance is contained in the known facts.

The disadvantage of the eager strategy is that many of the queries it performs during the search for a covering clause are superfluous; since oracle queries are typically answered by the user, they are the operation we would like to optimize the most. The search strategies described later are more economical in this sense, as they do not initiate oracle queries.

We demonstrate the performance of the system, incorporated with an eager search strategy, and then analyze its effect on Algorithm 5.

```
| ?- mis.

Next fact? call(set(search_strategy,eager)).
Next fact? member(a,[b,a]),true.
Checking fact(s)...
Error: missing solution member(a,[b,a]). diagnosing...
Error diagnosed: member(a,[b,a]) is uncovered.

Searching for a cover to member(a,[b,a])...
Declare member(X,Y)? declare(member(+x,+[x]),[determinate]).
Procedures called by member(X,Y)? member(_,_).

Checking: (member(X,[Y|Z])←true)
Found clause: (member(X,[Y|Z])←true)
   after searching 2 clauses.
```

```
Listing of member(X,Y):
   (member(X,[Y|Z])←true).
Checking fact(s)...no error found.
```

The clause found by the system on the basis of the fact *<member(a,[a]), true>* says that any element is a member of any nonempty list. We give the system a counterexample to that.

```
Next fact? member(a,[b,c]),false.
Checking fact(s)...
Error: wrong solution member(a,[b,c]). diagnosing...
Error diagnosed: (member(a,[b,c])←true) is false.
Listing of member(X,Y):
```

The next fact we supplied caused the system to discard that false clause.

```
Checking fact(s)...
Error: missing solution member(a,[b,a]). diagnosing...
Error diagnosed: member(a,[b,a]) is uncovered.

Searching for a cover to member(a,[b,a])...
Checking: (member(X,[Y|Z])←true)
Refuted: (member(a,[b,c])←true)

Query: member(a,[a])? y.

Query: member(b,[a])? n.
```

These two queries were performed by the eager search algorithm. The first was used to determine whether the clause *member(X,[Y|Z])←member(X,Z)* covers *member(a,[b,a])*, and the answer is positive; the second was to determine whether *member(X,[Y|Z])←member(Y,Z)* covers it, and the answer is negative. The reason for this search order will be clarified when the pruning strategy is discussed, in Section 4.5 below.

```
Checking: (member(X,[Y,Z|U])←true)
Refuted: (member(a,[b,c])←true)
Checking: (member(X,[Y|Z])←member(X,Z))
Found clause: (member(X,[Y|Z])←member(X,Z))
   after searching 4 clauses.
Listing of member(X,Y):
   (member(X,[Y|Z])←member(X,Z)).
```

The algorithm found the recursive clause for *member*, even though no instance of it was supplied by the user initially.

```
Checking fact(s)...
Error: missing solution member(a,[b,a]). diagnosing...

Query: member(a,[])? n.
Error diagnosed: member(a,[a]) is uncovered.
```

The fact <*member*(*a*,[*a*]), *true*> is known by the system; this saved the user from answering this query to the *missing_solution* diagnosis program.

```
Searching for a cover to member(a,[a])...
Checking: (member(X,[Y|Z])←true)
Refuted: (member(a,[b,c])←true)
Checking: (member(X,[Y])←true)
Refuted: (member(b,[a])←true)
Checking: (member(X,[X|Z])←true)
Found clause: (member(X,[X|Z])←true)
   after searching 4 clauses.
Listing of member(X,Y):
   (member(X,[Y|Z])←member(X,Z)).
   (member(X,[X|Z])←true).

Checking fact(s)...no error found.
```

And the synthesis of *member* is completed. The facts needed were

```
member(a,[b,a]), true.
member(a,[b,c]), false.

member(a,[a]), true.
member(b,[a]), false.

member(a,[]), false.
```

The first two were supplied by the user, the next two were asked for by the search algorithm, and the last one was given as an answer to a diagnosis query.

We prove that Algorithm 5, equipped with an eager search strategy, will identify an interpretation ***M*** in the limit, independent of the order of presentation of ***M***.

> **Theorem 4.11:** Let ***L*** be an ordered set of clauses, ***M*** an interpretation, and *h* a depth-bound function. Assume that ***L*** has an *h*-easy correct and complete program for ***M***, and that Algorithm 5, equipped with an eager search strategy, is given a presentation of ***M***. Then the algorithm identifies ***M*** in the limit.

> **Lemma 4.12:** Under the assumptions of Theorem 4.11, there is an $n>0$ such that whenever the search algorithm is invoked it returns a clause in $L_n$.

**Proof:** Let $n>0$ be the minimal index such that $L_n$ contains an *h*-easy correct and complete program, say $P_0$. Since $P_0$ is correct in ***M***, no clause in $P_0$ will ever get marked by Algorithm 2. Since $P_0$ is *h*-easy, no clause in ***P*** will ever get marked by Algorithm 4. These claims follows from the correctness of the diagnosis algorithms.

Since $P_0$ is complete for ***M***, it covers ***M***, hence any goal *A* in ***M*** has a clause ***p*** in $P_0$ that covers it in ***M***. It follows that whenever the search algorithm searches for a cover for *A*, it will either find ***p*** or a clause that precedes ***p*** in ***L***. []

**Proof of Theorem 4.11:** By Lemma 4.12, there is an *n* such that the algorithm always finds a covering clause in $L_n$, when searching for one.

Hence the assumption of Lemma 4.8, Page 91, is satisfied, and its conclusion is that the algorithm identifies $M$ in the limit.

The original implementation of the Model Inference System [82, 85] incorporated an eager search strategy.

An implementation of the eager *covers* test is shown as Program 14. *verify*($X$) tests whether $X$ is solvable, without actually instantiating $X$ to a solution. It is implemented via the hack *verify*($X$)← *not*(*not*($X$)).

**Program 14:** The eager *covers* test

```
covers(eager,(P←Q),P1) ←
        verify(( P=P1, satisfiable(Q) )).

satisfiable((P,Q)) ← !,
        query(exists,P,true), satisfiable(Q).
satisfiable(P) ←
        query(exists,P,true). ▮
```

### 4.4.2 A lazy search strategy

The next choice of the parameterized interpretation $X$ we investigate results in a ***lazy search strategy***. In this strategy $X$ is defined to be the set of goals known to be true.

The advantage of lazy search is that it does not bother the user with queries, since, by its definition, existential queries are answered with respect to the known facts. The disadvantage of this strategy is its order dependence. For example, this strategy would not discover the rule *member*($X$,[$Y$|$Z$])←*member*($X$,$Z$) unless the known facts contain an instance of that rule, i.e., there is a substitution $\theta$ such that both member(X,[Y|Z])$\theta$ and member(X,Z)$\theta$ are known to be true.

Two facts like <*member*(*a*,[*b*,*a*]),*true*>, and <*member*(*c*,[*c*]),*true*> will not do in this case. Thus an adversary ordering of facts, such as

<*member*(*a*,[*a*]),*true*>,<*member*(*b*,[*a*,*b*]),*true*>,<*member*(*c*,[*a*,*b*,*c*]),*true*>,..

can force the programs synthesized by the system to be arbitrarily large. This behavior is shown in the following session with the Model Inference System, equipped with a lazy search strategy.

After giving the system the facts <*member*(*a*,[*a*]), *true*>, and <*member*(*a*,[*x*]), *false*>, it came up with the axiom (*member*($X$,[$X$|$Z$])←*true*), which says that $X$ is a member of any list whose first element is $X$. From there the session progressed as follows.

```
Next fact? member(b,[a,b]),true.
Checking fact(s)...
Error: missing solution member(b,[a,b]). diagnosing...
Error diagnosed: member(b,[a,b]) is uncovered.

Searching for a cover to member(b,[a,b])...

Found clause: (member(X,[Y,Z|U])←true)
   after searching 3 clauses.
Listing of member(X,Y):
   (member(X,[X|Z])←true).
   (member(X,[Y,Z|U])←true).

Checking fact(s)...no error found.
```

The second clause found by the system says that $X$ is a member of any list that has at least two elements. We provide the system with a counterexample to that axiom.

```
Next fact? member(b,[a,x]),false.
Checking fact(s)...
Error: wrong solution member(b,[a,x]). diagnosing...
Error diagnosed: (member(b,[a,x])←true) is false.
Listing of member(X,Y):
   (member(X,[X|Z])←true).
```

```
Checking fact(s)...
Error: missing solution member(b,[a,b]). diagnosing...
Error diagnosed: member(b,[a,b]) is uncovered.

Searching for a cover to member(b,[a,b])...

Found clause: (member(X,[Y,X|U])←true)
   after searching 5 clauses.
Listing of member(X,Y):
   (member(X,[X|Z])←true).
   (member(X,[Y,X|U])←true).
Checking fact(s)...no error found.
```

The new clause found by the system says that $X$ is a member of any list of which $X$ is the second element. Here the difference between the eager and lazy search strategies becomes apparent: although the clause *member(X,[Y|Z])←member(X,Z)* precedes *member(X,[Y,X|U])←true* in the enumeration, the search algorithm did not suggest it when searching for a cover to *member(b,[a,b])* since it does not know that *member(b,[b])* is true. Continuing with this ordering of facts, we can force the algorithm to augment the program with arbitrarily complex base cases (unit clauses), that say "$X$ is a member of $Y$ if $X$ is the $n^{th}$ element of $Y$", for larger and larger $n$'s, before it finds a recursive clause. Instead, we provide it with a fact for which it can find a recursive rule.

```
Next fact? member(b,[x,a,b]),true.
Checking fact(s)...
Error: missing solution member(b,[x,a,b]). diagnosing...
Error diagnosed: member(b,[x,a,b]) is uncovered.

Searching for a cover to member(b,[x,a,b])...

Found clause: (member(X,[Y|Z])←member(X,Z))
   after searching 4 clauses.
Listing of member(X,Y):
   (member(X,[X|Z])←true).
   (member(X,[Y,X|U])←true).
   (member(X,[Y|Z])←member(X,Z)).

Checking fact(s)...no error found.
```

The system found the recursive clause for *member* when searching for a cover to *member(b,[x,a,b])*. The lazy *covers* test succeeded since the system knew from previous facts that *member(b,[a,b])* is true.

Even though we can force the *member* program constructed by the system to be arbitrarily complex, it will still identify *member* in the limit, since eventually we would have to supply it with the facts it needs for finding a recursive rule. This is not always the case, however; the following example shows that adversary orderings of facts may prevent identification in the limit by Algorithm 5 with a lazy search strategy, even though eager search strategy would allow identification.

Assume that $L$ contains the following clauses, constructed from the predicate symbols $p$ and $q$, the one place functions $a$ and $b$, and the constant *nil*.

$p(a^n(X)) \leftarrow q(X)$, for all $n \geq 0$.
$p(a^m(b^n(X)))$, for all $m,n \geq 0$.
$q(b^n(X)) \leftarrow q(X)$, for all $n \geq 0$.
$q(nil)$.

Assume that the target interpretation is $\{q(b^n(nil)) \mid n \text{ is even}\} \cup \{p(a(b^n(nil))) \mid n \text{ is even}\}$. A simple program for this interpretation is

$p(a(X)) \leftarrow q(X)$.
$q(b(b(X))) \leftarrow q(X)$.
$q(nil)$.

However, if we order the facts so that $<p(a(b^n(nil))),true>$ precedes $<q(b^n(nil)),true>$ for any even $n$, the lazy search algorithm will never discover a recursive clause for $p$, but only unit clauses. This implies that Algorithm 5 equipped with a lazy search strategy would not converge on this presentation, hence will not identify the interpretation in the limit.

**Definition 4.13:** Let $\succ$ a well-founded ordering on goals. A sequence of facts $S$ is said to *conform* with $\succ$ if for every two facts $<A,true>$, $<B,true>$, if $<A,true>$ precedes $<B,true>$ in $S$ then it is not the case that $B \succ A$.

We say that $S$ *eventually conforms* with $\succ$ if it can be made to conform with $\succ$ by reordering some finite initial segment of it.

Consider again the example above, and assume that the presentation eventually gets ordered according to the well-founded ordering implied by $L$. It is easy to see that the lazy search algorithm will find the recursive clause for $p$ once it sees $<q(b^n(nil)),true>$ before seeing $<p(a(b^n)(nil)),true>$, for a sufficiently large $n$. But until this happens, the unit clauses it adds to the program can be arbitrarily large. The following theorem generalizes this behavior.

**Theorem 4.14:** Let $M$ be an interpretation and $L$ an ordered set of clauses, such that $L$ contains the unit clause $A \leftarrow$ for any goal $A$ in $M$. Let $h$ be a depth-bound for which $L$ contains an $h$-easy correct and complete program for $M$.

Assume that Algorithm 5, equipped with a lazy search strategy, is given a presentation of $M$ that eventually conforms with $\succ$, where $\succ$ is the well-founded ordering associated with $h$. Then the algorithm identifies $M$ in the limit.

**Proof:** Similar to the proof of Theorem 4.11, we need to show that there is an $n>0$ such that the search for a covering clause always succeeds within $L_n$.

Assume that the presentation has a finite (possibly empty) unordered initial segment $S_0$, and that $L_k$ has an $h$-easy correct and complete program for $M$. For each fact $<A,true>$ the search algorithm will not go past the unit clause $A \leftarrow$: this clause covers $A$ in any interpretation, and it will not get marked since it is true in $M$ and not diverging. Let $m$ be the largest index of any unit clause $A \leftarrow$ for which there is a fact $<A,true>$ in $S_0$. By the time the algorithm reads in the first fact not in $S_0$, it has included in $P$ only clauses in $L_m$.

The facts that follow $S_0$ are ordered according to $\succ$, which implies that for every fact $<A,true>$ that follows $S_0$ in the presentation, all goals $B$ in $M$ for which $A \succ B$ are already known to be true. Assume that the algorithm is searching for a cover for such a goal $A$, and that it encounters a true, nondiverging clause $p=A' \leftarrow B_1,B_2,\ldots,B_n$ that covers $A$ in $M$. Then the true instances of the $B$'s with which $p$ covers $A$ are all smaller than $A$ with respect to $\succ$, hence they are known to be true, and the search algorithm selects $p$. Hence, by an argument similar to that of Lemma 4.12, the algorithm would not include in $P$ at that stage clauses with an index larger then $k$. Hence $n=max\{k, m\}$ satisfies the assumption of Lemma 4.8, and identification in the limit follows. ▯

As evident from the proof the theorem, if the presentation is strictly ordered, then the assumption that $L$ contains unit clauses that correspond to the positive facts is unnecessary; hence the following Corollary.

**Corollary 4.15:** Let $M$ be an interpretation, $L$ an ordered set of clauses, and $h$ a depth-bound such that $L$ contains an $h$-easy correct and complete program for $M$.

Assume that Algorithm 5, equipped with a lazy search strategy, is given a presentation of $M$ that (strictly) conforms with $\succ$. Then the algorithm identifies $M$ in the limit.

An implementation of the lazy *covers* test is shown as Program 15.

**Program 15:** The lazy *covers* test

```
covers(lazy,((P←Q),P1) ←
        verify(( P=P1, fact_satisfiable(Q) )).

fact_satisfiable((P,Q)) ←!,
        fact_satisfiable(P), fact_satisfiable(Q).
fact_satisfiable(P) ←
        system(P) → P ; fact(P,true). ▯
```

### 4.4.3 An adaptive search strategy

Our third strategy, called ***adaptive search***, is a compromise between the two strategies described above. In this strategy the parameterized interpretation $X$ increases both as a function of the facts the algorithm has seen so far, and as a function of the program it has developed. The interpretation we choose for $X$ is $\{A \mid A \text{ is in } M(P \cup S)\}$, were $S$ is the set of goals known to be true, and $P$ is the program developed by the algorithm. This choice implies that to test whether a clause $A \leftarrow B$ covers a goal $A'$, we unify $A$ with $A'$ and try to solve the resulting $B$ using the program $P$, ugmented with unit clauses $C\leftarrow$ for each known fact $<C,true>$.

The advantage of adaptive search over eager search is that, similar to lazy search, it does not query the user during the search; the advantage of it over lazy search is that it is less order dependent. The following session with the Model Inference System, equipped with an adaptive search procedure, demonstrates this.

After giving the system the facts

```
member(a,[a]), true.
member(b,[a,b]), true.
member(a,[x]), false.
member(b,[a,x]), false.
```

it came up with the program

```
member(X,[X|Z])←true.
member(X,[Y,Z|U])←true.
```

as it did in the previous example. The session continued from this point as follows:

```
Next fact? member(b,[a,x]),false.
Checking fact(s)...
Error: wrong solution member(b,[a,x]). diagnosing...
Error diagnosed: (member(b,[a,x])←true) is false.
Listing of member(X,Y):
    (member(X,[X|Z])←true).
```

The system discarded of the false clause.

```
Checking fact(s)...
Error: missing solution member(b,[a,b]). diagnosing...
Error diagnosed: member(b,[a,b]) is uncovered.

Searching for a cover to member(b,[a,b])...

Found clause: (member(X,[Y|Z])←member(X,Z))
    after searching 4 clauses.
Listing of member(X,Y):
    (member(X,[X|Z])←true).
    (member(X,[Y|Z])←member(X,Z)).

Checking fact(s)...no error found.
```

The system found that *member(X,[Y|Z])←member(X,Z)* covers *member(b,[a,b])*, even though it was not supplied with the fact *member(b,[b])*, since it could solve the latter goal using the part of the program it already constructed: the clause *member(X,[X|Z])*.

We find adaptive search the most useful of the three in synthesizing the more complex programs. In the following theorem adaptive search is claimed to be as powerful as lazy search.

> **Theorem 4.16:** Under the assumptions of Theorem 4.14, if Algorithm 5 is equipped with an adaptive, rather than lazy, search strategy, then it identifies $M$ in the limit.

**Proof:** The difference between adaptive search and lazy search is that a clause may cover a goal according to adaptive search, but fail to cover it according to lazy search. Since whenever lazy search finds a covering clause, adaptive search also finds the same clause or a clause with a smaller index in $L$, this behavior does not invalidate the bound on the index of clauses included in $P$ as argued for in the proof of Theorem 4.14. Hence Lemma 4.8 applies. ▯

By the same argument, Corollary 4.15 holds also for adaptive search.

One possible drawback of adaptive search is that it may result in the program having "dead-code", i.e. clauses that do not cover any goal in $M$. This may happen since the adaptive search may erroneously conclude that a clause covers some goal, due to $P$ being incorrect. This cannot happen with eager or lazy search, since they do not depend on the correctness of the program being synthesized. We do not have yet enough experience to determine the practical consequences of this drawback.

A simplified implementation of the adaptive *covers* test is shown as Program 16. The actual implementation, shown in Appendix II, is complicated by the need to handle the possibility of a stack-overflow in the computation of *fact_solve*.

**Program 16:** The adaptive *covers* test

```
covers(adaptive,((P←Q),P1) ←
        verify(( P=P1, fact_solve(Q) )).

fact_solve((A,B)) ← !,
        fact_solve(A), fact_solve(B).
fact_solve(A) ←
        system(A) → A ;
        fact(A,true) ;
        clause(A,B), fact_solve(B). ∎
```

## 4.5 A pruning strategy

When detecting an uncovered goal, Algorithm 5 searches $L$ for a clause that covers this goal. Typically, the size of that search space is exponential in the size of the target clause; hence performing linear search would forbid any practical application of the algorithm.

In this section we develop ways to prune the search for a covering clause. We investigate structural and semantic properties of clauses, and develop a pruning strategy based on these properties. The pruning strategy organizes the search space of clauses according to their logical power, in a structure called a refinement graph. We study refinement graphs, give concrete examples of them, and provide evidence to the utility of the pruning strategy.

### 4.5.1 The refinement graph

A *refinement graph* is a directed, acyclic graph in which nodes are definite clauses and arcs correspond to refinement operations, defined below.

Let $L$ be a set of definite clauses and $\rho$ a mapping from $L$ to finite subsets of $L$. We define $<_\rho$ to be the binary relation over $L$ for which $p<_\rho q$ iff there is a finite sequence of clauses $p_1,p_2,\ldots,p_n$ such that $p_1=p$, $p_n=q$, and $p_{i+1}$ is in $\rho(p_i)$, for $0\leq i<n$. We say that $p\leq_\rho q$ iff $p<_\rho q$ or $p=q$. Note that we do not distinguish between clauses that are variants, i.e. differ only in the choice of variable names, and interpret $p=q$ accordingly.

The mapping $\rho$ is said to be a *refinement operator* over $L$ iff the following two conditions hold:

1. The relation $<_\rho$ is a well-founded ordering over $L$.
2. For every interpretation $M$ and goal $A$, if $q$ covers $A$ in $M$ and $p<_\rho q$ then $p$ covers $A$ in $M$.

It is easy to see that the union of two refinement operators is a refinement operator.

We define $L(\rho)$ to be the set of clauses $p$ for which $\square \leq_\rho p$, and say that $\rho$ *generates* $L$ if $L$ is a subset of $L(\rho)$.

The refinement operators we use employ two syntactic operations on clauses, which satisfy the covering condition:

1. Instantiate a clause.
2. Add a goal to the condition of a clause.

In [82] we showed that there is a refinement operator based on these two operations that generates all clauses over a finite set of predicate and function symbols. The existence of such a general refinement operator is mostly of theoretical interest; we find restricted refinement operators, tuned to a particular application, to be more useful. Examples of some concrete refinement operators are shown below.

### 4.5.2 Examples of refinement operators

We assume a given fixed set of predicate and function symbols, and define the following refinement operator $\rho_1$ with respect to these sets.

Let $p$ be a clause. Then $q$ is in $\rho_1(p)$ iff one of the following holds:

1. $p=\square$ and $q=a(X_1,X_2,...,X_n)$, for some $n$-place predicate symbol $a$ of $L$, $n\geq 0$, and $X_1,X_2,...,X_n$ are $n$ distinct variables.
2. $q$ is obtained by unifying two variables in $p$.
3. $q$ is obtained by instantiating a variable $X$ in $p$ to a term $t(X_1,X_2,...,X_n)$, where $t$ is an $n$-place function symbol and $X_1,X_2,...,X_n$ are variable symbols not occuring in $p$.
4. $p$ is a definite clause whose head is $A$, and $q$ is obtained by adding to $p$'s body a goal $B$, whose size is less then or equal to the the size of $A$, and every variable in $B$ occurs in $A$.

In [86] we showed that the class of logic programs generated by this refinement operator have the following property:

1. For any program $P$ in $L(\rho_1)$ there is an alternating Turing machine [24] $T$ that works in linear space, and accepts $A$ iff $A$ is in $M(P)$.
2. For any alternating Turing machine $T$ that works in linear space there is a program $P$ in $L(\rho_1)$ such that the goals in $M(P)$ are exactly those that represent (under some natural, fixed encoding) configurations of $M$ that lead to acceptance.

We also showed that if we restrict $\rho_1$ to add at most one goal to the body of a clause, the resulting class of programs satisfies the above claims for Nondeterministic Linear Space.

Many relations can be expressed naturally by logic programs within $L(\rho_1)$. Examples are list membership, subsequence, subset, concatenation,

and binary tree isomorphism.

$member(X,[X|Y])$.
$member(X,[Y|Z]) \leftarrow member(X,Z)$.

$subsequence([],X)$.
$subsequence([A|X],[A|Y]) \leftarrow subsequence(X,Y)$.
$subsequence(X,[A|Y]) \leftarrow subsequence(X,Y)$.

$subset([],X)$.
$subset([X|Xs],Ys) \leftarrow member(X,Ys), subset(Xs,Ys)$.

$append([],X,X)$
$append([A|X],Y,[A|Z]) \leftarrow append(X,Y,Z)$

$isomorphic(X,X)$.
$isomorphic(t(X1,X2),t(Y1,Y2)) \leftarrow isomorphic(X1,Y1), isomorphic(X2,Y2)$.
$isomorphic(t(X1,X2),t(Y1,Y2)) \leftarrow isomorphic(X1,Y2), isomorphic(X2,Y1)$.

Figure 6 shows a portion of the refinement graph for the predicate *member* and the terms [] and $[X|Y]$.

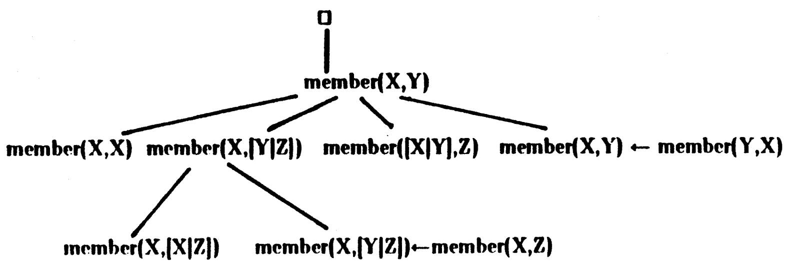

**Figure 6:** Part of the refinement graph for *member*

The second refinement operator we describe, $\rho_2$, also generates a class of logic programs with a characterizable expressive power: the equivalent of context-free grammars [1]. There is a one-to-one mapping between context-free grammars and definite clauses of a special form, called definite clause grammars. These grammars and their extensions were studied by Colmerauer [27], Warren and Pereira [65], among others. In this mapping every nonterminal is mapped into a binary predicate, and every terminal to an atom. For example, the following context-free grammar

$s \Rightarrow np,\ vp.$
$np \Rightarrow det,\ n.$
$np \Rightarrow det,\ n,\ [that],\ vp.$
$np \Rightarrow pname.$
$vp \Rightarrow tv,\ np.$
$vp \Rightarrow itv.$
$det \Rightarrow [every].$
$det \Rightarrow [a].$

is mapped into the logic program:

$s(X,Y) \leftarrow np(X,U),\ vp(U,Y).$
$np(X,Y) \leftarrow det(X,U),\ n(U,Y).$
$np(X,Y) \leftarrow det(X,U),\ n(U,[that|W]),\ vp(W,Y).$
$np(X,Y) \leftarrow pname(X,Y).$
$vp(X,Y) \leftarrow tv(X,U),\ np(U,Y).$
$vp(X,Y) \leftarrow itv(X,Y).$
$det([every|X],X).$
$det([a|X],X).$

The semantics of a predicate $n(X,Y)$ that corresponds to the nonterminal $n$ is "the difference between the string $X$ and the string $Y$ is of syntactic category $n$". Procedurally, $n(X,Y)$ accepts a string $X$ as input, chops off it a substring of category $n$, and returns the rest of the string as output in $Y$. The logic program above can be executed as a recursive-descent recognizer for the language defined by this grammar.

It can be shown [27] that if a context-free grammar $G$ is mapped into

a logic program $P$ under this transformation, then for each string $x$ and each nonterminal $n$, $n$ generates $s$ in $G$ iff $P$ succeeds on $n(x,y)$ in $P$, for any two strings $x$ and $y$ whose difference is $s$.

We define a refinement operator $\rho_2$ that generates the definite clause equivalent of context-free grammar rules over a given set $N$ of nonterminal symbols and a set $T$ of terminal symbols. Let $p$ be a clause. Then $q$ is in $\rho_2(p)$ iff one of the following holds:

1. $p=\square$ and $q=n(X,Y)$, where $n$ is in $N$.
2. $Y$ is a variable that occurs once in $p$ in the second argument of some predicate, and $q$ is $p\{Y \rightarrow [t|Z]\}$, where $t$ is in $T$ and $Z$ is a variable symbol not occuring in $p$.
3. $Y$ is a variable that occurs once in $p$ in the second argument of some predicate, $n$ is in $N$, and $q$ is obtained from $p$ by adding to its condition the atom $n(Y,Z)$, where $Z$ is a variable symbol not occuring in $p$.
4. $Y$ is a variable that occurs once in $p$ in the second argument of some predicate, $X$ is the variable symbol that occurs in the second argument of the predicate in the head of $p$, and $q=p\{Y \rightarrow X\}$.

A session with the Model Inference System equipped with this refinement operator is shown in Appendix I below.

Theorem 4.10 above says that if $L$ has a polynomial candidate space for covering clauses with respect to $M$, and every program in $L$ runs in polynomial time, then the algorithm runs in polynomial time. We restrict $\rho_2$, the refinement operator that generates definite clause grammars, so the language it generates satisfy the two assumptions.

To satisfy the polynomial runing time assumption, we restrict the refinement operator to generate only simple LL1 grammars [1]. A *simple LL1 grammar* is a grammar with no $\epsilon$-productions, and in which every right hand side of a production begins with a distinct terminal symbol. The logic program equivalent of an LL1 grammar runs in time linear in the size of the input goal. For a polynomial bound to hold on the size of the candidate space, we restrict the grammars production to have at most $k$ nonterminals on their right hand side, for some fixed $k$.

The definition of $\rho_3$ is obtained from that of $\rho_2$ above by modifying its first clause to read:

1. $p=\square$ and $q=n([t|X],Y)$, where $n$ is in $N$ and $t$ is in $T$.

And by adding to its third clause the restriction that the goal is added to the body of the clause only if it has less than $k$ goals in it.

Let $G$ be a context-free grammar. We say that grammar rule $R$ generates $s$ in $G$ if there is a derivation of $s$ in $G$ that starts with $R$. Let $M$ be an interpretation that corresponds to the language generated by $G$. From the correspondence between context-free grammars and definite clause grammars it follows that a definite grammar clause $p$ covers a goal $n(X,Y)$ in $M$ iff the grammar rule that corresponds to $p$ generates the string that is the difference between $X$ and $Y$ in $G$. Hence we argue about the size of the candidate space in terms of grammar rules.

**Lemma 4.17:** Let $G$ be a grammar over the nonterminal set $N$ and terminal set $T$, and $s$ a string in the language of $G$ of length $n$. Then for any $k$, there are at most $p(n)$ grammar rules with $k$ nonterminals that generate $s$ in $G$, where $p$ is a polynomial that depends on $k$ and $G$.

**Proof:** For a grammar rule $R$ to generate $s$ in $G$, the terminals in the rule must be a subsequence of $s$, and every contiguous block of terminals in $s$ that are missing from $R$ must be generated by at least one nonterminal in $R$. Since there at most $k$ nonterminals in $R$, there are at most $|T|^k$ choices for the nonterminals, and at most $kn^2$ choices for the contiguous blocks of terminals in $s$ they represent. Hence there are at most $|T|^k kn^2$ such grammar rules over $T$ and $N$. ▯

We have applied the Model Inference System to infer an LL1 grammar for the statements of Wirth's PLO programming language [102], without incorporating the restriction on the number of nonterminals. We have made the assumption that expressions are in prefix notation, otherwise the language does not have a simple LL1 grammar. The refinement operator was given the terminal set {*begin*, *end*, *if*, *then*, *while*, *do*, *call*, *odd*, ':=', =, ';', +, −} and the nonterminal set {*statement*, *statementlist*, *condition*, *expression*, *ident*, *number*, *comparator*}. We have supplied the

system with the following initial set of facts:

*expression*([*a*],[]), *true*.
*expression*([1],[]), *true*.
*expression*([+,*a*,1],[]), *true*.
*expression*([−,*a*,1],[]), *true*.
*condition*([=,*a*,+,*a*,1],[]), *true*.
*statement*([*a*,:=,+,*a*,1],[]), *true*.
*statement*([*call*,*a*],[]), *true*.
*statement*([*while*,=,*a*,1,*do*,*a*,:=,+,*a*,1],[]), *true*.
*statement*([*if*,=,*a*,1,*then*,*a*,:=,+,*a*,1],[]), *true*.
*statementlist*([;,*if*,=,*a*,1,*then*,*a*,:=,+,*a*,1,*end*],[]), *true*.
*statement*([*begin*,*call*,*a*,;,*if*,=,*a*,1,*then*,*a*,:=,+,*a*,1,*end*],[]), *true*.

After working for 110 CPU seconds[3], the system came up with the following grammar rules. The number preceding a clause is the number of clauses searched before finding it.

(8) (*statement*(X,Y):−*ident*(X,[:=|U]),*expression*(U,Y))
(4) (*statement*([*call*|X],Y):−*expression*(X,Y))
(14) (*statement*([*while*|X],Y):−*condition*(X,[*do*|U]),*statement*(U,Y))
(14) (*statement*([*if*|X],Y):−*condition*(X,[*then*|U]),*statement*(U,Y))
(8) (*statement*([*begin*|X],Y):−*statement*(X,U),*statementlist*(U,Y))
(7) (*statementlist*([;|X],Y):−*statement*(X,[*end*|Y]))
(8) (*condition*([=|X],Y):−*expression*(X,U),*expression*(U,Y))
(2) (*expression*(X,Y):−*ident*(X,Y))
(2) (*expression*(X,Y):−*number*(X,Y))
(8) (*expression*([+|X],Y):−*expression*(X,U),*expression*(U,Y))
(8) (*expression*([−|X],Y):−*expression*(X,U),*expression*(U,Y))

Which contains two errors: the rule for a procedure call is overgeneralized, as PLO has only parameterless procedure calls, and the rule for *statementlist* is too constrained, as it implies that a statement list has

[3]Most of the system was not compiled during this experiment, so the timing should not be considered as optimal.

only one statement. From this point the session continued as follows. We first gave the system two additional facts:

*statement([call,+,a,1],[]),false.*
*statement([begin,call,x,´;´,a,:=,+,a,b,´;´,call,y,end],[]),true.*

It queried for one more:

*query*: *statement([call,x,;,a,:=,+,a,b,;,call,y,end],X)? y.*
*which X? [;,a,:=,+,a,b,;,call,y,end].*
*which X? no.*

And corrected the two errors (in 20 CPU seconds). The new grammar rules are:

(5) *(statement([call|X],Y):−ident(X,Y))*
(9) *(statementlist([;|X],Y):−statement(X,U),statementlist(U,Y))*

The system spent most of its time pruning the refinement graph. The branching factor of the unpruned graph is the number of terminal symbols for the first refinement, which in this grammar is 13, and the number of terminals plus the number of nonterminals for all other refinements, which in the case of this grammar is 13+7=20, plus one refinement that closes the clause (i.e. unifies the output variable in the head of the clause with the free output variable in the body of the clause). A closed clause has no subsequent refinements. The more complex grammar rules, such as the ones for the *while* and *if* statements, were found at depth 5 in the refinement graph after searching 14 clauses. In the worst case, the unpruned breadth-first search for these clauses could have gone through 13+ 13×21 + 13×20×21 13×20×20×21 + 13×20×20×20×21 = 2,298,946 clauses to search the graph at that depth.

### 4.5.3 Searching the refinement graph

The pruning strategy we develop is based on searching the refinement graph in breadth-first order. It uses the property of refinement graphs that if a clause does not cover a goal in some interpretation $M$, then any refinement of this clause does not cover this goal in $M$ also. Hence

whenever a clause fails to cover a goal, the subgraph rooted at this clause is pruned. Algorithm 6 below describes this search strategy.

**Algorithm 6:** A pruning breadth-first search of the refinement graph

*Given*: An oracle for some interpretation $X$ and
a program for a refinement operator $\rho$.

*Input*: A goal $A$ and a set of marked clauses containing $\Box$.

*Output*: An unmarked clause $p$ in $L(\rho)$ that covers $A$ in $X$,
if one such exists, or *no clause found.*

*Algorithm*:

```
set Q to [□].
repeat
    remove a clause p from Q;
    compute R, the set of refinements of p that cover A in X,
        using the refinement operator ρ and the oracle for X;
    add R to the end of Q.
until the queue Q is empty or contains an unmarked clause.
if Q is empty
    then return no clause found
    else return the first unmarked clause in Q. []
```

Lemma 4.18 describes the conditions under which this search algorithm will find a clause as desired.

**Lemma 4.18:** Let $\rho$ be a refinement operator, $X$ an interpretation, $R$ a set of marked clauses, and $A$ a goal. If Algorithm 6 is applied to a goal $A$ for which $L$ has an unmarked clause that covers $A$ in $X$, then it will terminate and return such a clause.

**Proof:** It is clear that if the algorithm terminates and returns a clause then the clause satisfies the required properties. Assume that the algorithm fails to return such a clause, even though $L(\rho)$ contains one.

Let $p$ be an unmarked clause in $L(\rho)$ that covers $A$ in $M$, whose depth in the refinement graph is minimal, and $p_1,p_2,\ldots,p_n$ be a sequence of clauses such that $p_1=\Box$, $p_n=p$, and $p_{i+1}$ is in $\rho(p_i)$, $1\leq i\leq n$.

The algorithm can fail to return a clause in two ways: it can terminate and return *no clause found*, or it can fail to terminate. For either of these to happen, $p_n$ should not be added to the queue. This implies that the algorithm has to remove $p_i$ from the queue, for some $i\leq n$, without adding $p_{i+1}$ to it. This can happen only if $p_{i+1}$ does not cover $A$. However, by the assumption that $p_n$ covers $A$ and the definition of a refinement operator it follows that $p_{i+1}$ covers $A$, a contradiction. ▯

The oracle for $X$ appears in Algorithm 6 as a parameter. The different search strategies discussed in Section 4.4 above instantiate this parameter to different interpretations. The pruning strategy, however, is insensitive to the interpretation chosen.

### 4.5.4 An implementation of the pruning search algorithm

Program 17 below implements Algorithm 6. It doesn't start from the empty clause but from the most general term that corresponds to the given goal.

The procedure *search_for_cover(Qhead,Qtail,P,C)* represents the queue as a lazy list. The *remove* operation on the queue removes the first element from *Qhead*, the head of the list. The *add* operation on the queue instantiates *Qtail*, the unspecified tail of the list. The call *bagof0(Y,(refinement(X,Y), covers(Y,P)),Qnew)* returns a list *Qnew* of all instances of *Y* for which there is an *X* that solves the conjunctive goal *(refinement(X,Y), covers(Y,P))*. The call to *covers* is the one that prunes the search. The actual implementation of the search algorithm is slightly complicated by the need to generate and handle special data-structures that are associated with each clause and are used by refinement operator.

**Program 17:** A pruning breadth-first search of the refinement graph

```
search_for_cover(P,C) ←
        nl, writel(['Searching for a cover to ',P,'...']), nl,
        mgt(P,P1),
        search_for_cover([(P1←true)|Xs],Xs,P,C).

search_for_cover(Qhead,Qtail,P,C) ←
        Qhead==Qtail,
        writel(['Can't find a cover for ',P,'. queue is empty']),
        !, fail.
search_for_cover([X|Qhead],Qtail,P,C) ←
        bagof0(Y,( refinement(X,Y), covers(Y,P) ),Qnew),
        check_refinements(Qnew,Qhead,Qtail,P,C).

check_refinements(Qnew,Qhead,Qtail,P,C) ←
        member(C,Qnew), good_clause(C), !.
check_refinements(Qnew,Qhead,Qtail,P,C) ←
        append(Qnew,Qnewtail,Qtail),
        search_for_cover(Qhead,Qnewtail,P,C).
```

*covers(X,P)* ← *X* covers *P*. See Programs for the *covers* test, Section 4.4.

*good_clause(X)* ← *X* is an unmarked clause. ▯

## 4.6 Comparison with other inductive synthesis systems

In [84] we compared the behavior of an older version of the Model Inference System with the systems of Summers [92] and Biermann [9]. We summarize the main points found, and add a comparison with Wharton's system for grammatical inference [98]

We have equipped the Model Inference System with a special-purpose refinement operator, that generates a class of logic program that were sufficiently general to include all of Summer's examples. The following program for packing a list of lists into one list is in that class.

```
    pack([X|Y],Z) ← pack(Y,V),append(X,V,Z).
    pack([],[]).
```

We have found that we could infer most of Summer's examples in less than one CPU minute. For example, that version inferred the program for *pack* in 9 CPU seconds, and from 29 facts, most of them negative. We have applied the current Model Inference System to the same example. The result is shown below:

```
  | ?- mis.

  Next fact? pack([],[]),true.
  Checking fact(s)...
  Error: missing solution pack([],[]). diagnosing...
  Error diagnosed: pack([],[]) is uncovered.

  Searching for a cover to pack([],[])...
  Declare pack(X,Y)? declare(pack(+[[x]],-[x]),[total,determinate]).
  Procedures called by pack(X,Y)? pack(_,_),append(_,_,_).
```

We have declared *pack* to be a total, determinate procedure whose input is a list of lists, and its output is a list; we also declared that *pack* can call *append*, and also call itself recursively.

```
  Found clause: (pack(X,[]) ← true)
     after searching 3 clauses.
  Listing of pack(X,Y):
     (pack(X,[]) ← true).
  Checking fact(s)...no error found.
```

After searching for not too long, the system found a program that is consistent with all it knows so far: on any input, *pack* would return now the empty list.

```
  Next fact? pack([[a]],[a]),true.
  Checking fact(s)...
  Error: missing solution pack([[a]],[a]). diagnosing...
  Error diagnosed: pack([[a]],[a]) is uncovered.
```

The new example cannot be handled by the current program; so a search for a new clause begins.

```
  Searching for a cover to pack([[a]],[a])...
  Found clause: (pack([X|Y],X) ← true)
     after searching 8 clauses.
  Listing of pack(X,Y):
     (pack(X,[]) ← true).
     (pack([X|Y],X) ← true).
```

The clause found would return the first element of any nonempty list.

```
  Checking fact(s)...no error found.

  Next fact? pack([[a],[b]],[a,b]),true.
  Checking fact(s)...
  Error: missing solution pack([[a],[b]],[a,b]). diagnosing...
  Error diagnosed: pack([[a],[b]],[a,b]) is uncovered.

  Searching for a cover to pack([[a],[b]],[a,b])...
  Found clause: (pack([X,Y|Z],U) ← append(X,Y,U))
     after searching 85 clauses.
  Listing of pack(X,Y):
     (pack(X,[]) ← true).
     (pack([X|Y],X) ← true).
     (pack([X,Y|Z],U) ← append(X,Y,U)).

  Checking fact(s)...no error found.
```

The program being constructed still seems to be off the track, although it is consistent with the known facts. The new clause returns the concatenation of its first two elements; this clause behaves correctly for lists with no more than two elements in them.

> *Next fact?* $pack([[a],[b],[c]],[a,b]),false.$
> *Checking fact(s)...*
> *Error: wrong solution* $pack([[a],[b],[c]],[a,b])$. *diagnosing...*
> *Error diagnosed*: $(pack([[a],[b],[c]],[a,b]) \leftarrow append([a],[b],[a,b]))$ *is false.*
> *Listing of* $pack(X,Y)$:
> $(pack(X,[]) \leftarrow true).$
> $(pack([X|Y],X) \leftarrow true).$

At an earlier stage, we declared *append* to be a system predicate, which means that its program is considered to be correct. Therefore the diagnosis algorithm performed no user queries to detect this false clause.

> *Checking fact(s)...*
> *Error: missing solution* $pack([[a],[b]],[a,b])$. *diagnosing...*
> *Error diagnosed*: $pack([[a],[b]],[a,b])$ *is uncovered.*
>
> *Searching for a cover to* $pack([[a],[b]],[a,b])$...
> *Refuted*: $(pack([[a],[b],[c]],[a,b]) \leftarrow append([a],[b],[a,b]))$
> *Found clause*: $(pack([X|Y],Z) \leftarrow pack(Y,V),append(X,V,Z))$
> *after searching* 110 *clauses.*
> *Listing of* $pack(X,Y)$:
> $(pack(X,[]) \leftarrow true).$
> $(pack([X|Y],X) \leftarrow true).$
> $(pack([X|Y],Z) \leftarrow pack(Y,V),append(X,V,Z)).$

After removing the previous clause, $pack([[a],[b]],[a,b])$ became uncovered. After searching through 110 clauses, the correct recursive definition of *pack* was found. The base clauses still need to be fixed.

> *Checking fact(s)...*
> *Error: wrong solution* $pack([[a],[b],[c]],[a,b])$. *diagnosing...*

> *Query*: $pack([[b],[c]],[b])$? *n.*
> *Error diagnosed*: $(pack([[b],[c]],[b]) \leftarrow true)$ *is false.*
> *Listing of* $pack(X,Y)$:
> $(pack(X,[]) \leftarrow true).$
> $(pack([X|Y],Z) \leftarrow pack(Y,V),append(X,V,Z)).$

The first error was detected by executing the new program against the known facts, and was diagnosed using one user query. The same is true for the following error.

> *Checking fact(s)...*
> *Error: wrong solution* $pack([[a],[b],[c]],[a,b])$. *diagnosing...*
>
> *Query*: $pack([[c]],[])$? *n.*
> *Error diagnosed*: $(pack([[c]],[]) \leftarrow true)$ *is false.*
> *Listing of* $pack(X,Y)$:
> $(pack([X|Y],Z) \leftarrow pack(Y,V),append(X,V,Z)).$
>
> *Checking fact(s)...*
> *Error: missing solution* $pack([],[])$. *diagnosing...*
> *Error diagnosed*: $pack([],[])$ *is uncovered.*
>
> *Searching for a cover to* $pack([],[])$...
> *Found clause*: $(pack([],[]) \leftarrow true)$
> *after searching* 4 *clauses.*
> *Listing of* $pack(X,Y)$:
> $(pack([X|Y],Z) \leftarrow pack(Y,V),append(X,V,Z)).$
> $(pack([],[]) \leftarrow true).$
>
> *Checking fact(s)...no error found.*

And the correct base clause is found. The session took 19 seconds CPU time, and the system needed the following facts.

```
pack([],[]), true.
pack([[a]],[a]), true.
pack([[a],[b]],[a,b]), true.
pack([[a],[b],[c]],[a,b]), false.
pack([[b],[c]],[b]), false.
pack([[c]],[]), false.
```

There are two reasons for the slowdown compared with the old version of the Model Inference System: in this session we have used a more general refinement operator, and most of the system is not compiled, due to the Prolog compiler's inability to handle →. The reason for the improvement in the number of facts needed is the use of the adaptive search strategy in the last example, where the old Model Inference System could use eager search only.

We have also compared the Model Inference System to Biermann's system for the synthesis of regular Lisp programs from examples [9]. Biermann's system is strongly influenced by Summers's method, although is has an enumerative component which Summer's system did not. Again, we have developed a special refinement operator that handled all the examples in Biermann's paper. An example of a program in that class is ***heads***, which constructs a list of all first elements of lists in a list of Lisp atoms and lists.

```
heads([],[]).
heads([[X|Y]|Z],[X|W]) ← heads(Z,W).
heads([X|Y],Z) ← atom(X), heads(Y,Z).
```

We have been able to synthesize all of Biermann's programs; the time range was between 2 and 38 seconds, and the number of facts needed was between 6 and 25; again, eager search was used. Biermann's system needed between a fraction of a second and half an hour CPU time on a PDP-11 for the same examples. The hardest example for both systems was ***heads*** above.

Wharton's system [98] infers context-free grammars from examples. One of the more complex grammars his system inferred is the following grammar for arithmetic expressions:

$S \Rightarrow A$
$S \Rightarrow S+A$
$S \Rightarrow S-A$
$S \Rightarrow -A$
$A \Rightarrow a$
$A \Rightarrow (S)$

His system took 17 CPU minutes on an IBM 360 to find this grammar. It was given the following sample of positive strings:

$a,\ -a,\ a+a,\ a-a,\ (a),\ -a+a,\ -a-a,\ -(a),\ (-a)$

and the fact that no other strings of length less than or equal to four are in the language.

We have applied the Model Inference System, equipped with an adaptive search strategy and the refinement operator $\rho_2$, to the same inference task. After supplying it with the same positive facts, but with no negative facts, it came up with the following clauses:

```
s([a|X],X).
s([-|X],Y) ← s(X,Y).
s([a,+|X],Y) ← s(X,Y).
s([a|X],Y) ← s(X,Y).
s([(|X],Y) ← s(X,[)|Y]).
```

We added three negative facts:

```
s([a,a],[]), false.
s([-,-,a],[]), false.
s([,a,a],[]), false.
```

And the system came up with the program:

```
s([a|X],X).
s([a,+|X],Y) ← s(X,Y).
s(['('|X],Y) ← s(X,[')'|Y]).
s([a,−|X],Y) ← s(X,Y).
s([−,a|X],X) ← true.
s([−|X],Y) ← s(X,[+|U]),s(U,Y).
s([−,'('|X],Y) ← s(X,[')'|Y]).
s([−|X],Y) ← s(X,[−|U]),s(U,Y).
```

Which corresponds to the grammar:

$S \Rightarrow a$
$S \Rightarrow a + S.$
$S \Rightarrow ( S ).$
$S \Rightarrow a - S.$
$S \Rightarrow - a.$
$S \Rightarrow - S + S.$
$S \Rightarrow - ( S ).$
$S \Rightarrow - S - S.$

The whole session took 20 CPU seconds. During the session the system made two queries:

```
s([(,a,)],[+|X])?
s([a,−,a],[+|X])?
```

The answer to which is *false*. Both answers can be found mechanically, using the knowledge that for any nonterminal *n*, a necessary condition for $n(X,Y)$ to be true is that is has an instance in which $Y$ is a suffix of $X$.

# Chapter 5

# PROGRAM DEBUGGING

## 5.1 The bug-correction problem

Theoretically, the approach to program synthesis from examples developed in Chapter 4 is applicable to program debugging as well. To upgrade Algorithm 5 to a debugging algorithm one need only parameterize the initial program to be a program written by a human programmer, rather than the empty program. Algorithm 5 will then debug that program using the test-data supplied by the programmer.

It can be shown that the limiting behavior of the inductive synthesis algorithm is independent of the initial program. This property can be viewed as an advantage, since it implies that the inductive synthesis algorithm can correct arbitrary bugs in the initial program, but it also shows the weakness of the algorithm as a debugging tool: it loses much of the information in the initial program through the debugging process. The correct part of the initial program is retained by the algorithm, but this is not true of its incorrect part: if a clause is refuted, or is discovered to be diverging, it is removed from the program. When a clause is found missing, then the search for a new clause starts from scratch, without using the clause just removed as a "hint" for the search process.

In other words, the inductive synthesis algorithm is incremental at the program level, but not at the clause level. Our approach to debuging attempts to correct this drawback. To do so we follow the Program Mutation Project [18], and make the *competent programmer assumption.* We assume that the initial program was written with the intention of being correct, and that if it is not correct, then a close variant of it is. The debugging algorithm we develop tries to find such a variant; if it fails, it depends on the programmer to make the necessary change.

Formally, we make two assumptions. The first one, common in the field of program testing (e.g. [16, 44, 99]), is that the class of common errors we try to correct induces an equivalence relation on the set of candidate clauses. For example, the following two clauses

*append*([*X*|*Xs*],*Ys*,[*Z*|*Zs*]) ← *append*(*Xs*,*Ys*,*Zs*)

*append*([*X*|*Xs*],*Ys*,[*X*|*Zs*]) ← *append*(*Xs*,*Ys*,*Zs*)

are equivalent under the class of errors of misspelled variable names, discussed below.

The correction algorithm we develop searches the equivalence class that the current clause is in; if it finds in this class a clause that behaves correctly, the algorithm proposes this clause as a correction; otherwise it returns control to the user. The user can either try to correct the wrong clause manually (i.e. edit it), and by doing so successfully he necessarily ends up in a different equivalence class; or he can propose another equivalence class to be searched.

The second assumption we make is that each component of the program appears there for some reason. This assumption implies that if a piece of code is found to be incorrect, we cannot just throw it away; rather, we have to find the reason it is in the program, and find an alternative code that will satisfy that reason after the incorrect code has been discarded.

**Definition 5.1:** Let $P$ be the program to be debugged, and $M$ an interpretation. A *reason-set* for a clause $p$ in $P$ is a subset of $M$ which is intended to be covered by $p$ in $M$.

For example, the two goals {*append*([*a*,*b*],[*c*,*d*],[*a*,*b*,*c*,*d*]), *append*([*b*],[*c*,*d*],[*b*,*c*,*d*])} can can serve as a reason set for

*append*([*X*|*Xs*],*Ys*,[*X*|*Zs*]) ← *append*(*Xs*,*Ys*,*Zs*)

and {*append*([],[*c*,*d*],[*c*,*d*])} can serve as a reason set for *append*([],*Xs*,*Xs*).

We argue that under the above assumptions, the problem of debugging a program can be reduced to the following *bug-correction problem*:

*Given a reason-set $R$, a clause $p$, an interpretation $M$, and a set of marked clauses, find an unmarked clause $p'$ equivalent to $p$ that covers $R$ in $M$.*

We consider the two possible situations that occur in interactive debugging, and show how they can be described as instances of the bug correction problem.

If the error in the program is an incorrect solution or a diverging computation, then the appropriate diagnosis algorithm will detect a false or a diverging clause; such a clause needs to be removed from the program. Since, by our assumption, $p$ has some reason set $R$ associated with it, we do not just discard of it; rather, we try to find an equivalent clause to $p$ that covers $R$, but is neither refuted nor diverging.

If the error in the program is a missing solution, then the diagnosis algorithm detects a goal uncovered by the program. We then ask the user which clause in the program was "supposed to" cover this goal. Given such a clause $p$ with a reason set $R$, we try to find an equivalent clause to $p$ that covers $R \cup \{A\}$.

Thus the three types of possible errors in a program — termination with incorrect output, termination with missing output and nontermination — require a solution to the bug-correction problem. Given an algorithm that solves this problem, we can develop an interactive debugging algorithm such as Algorithm 7 below.

The top-level control structure of Algorithm 7 is very similar to the inductive program synthesis algorithm, Algorithm 5. Since both algorithms are incremental, i.e. they do not add to the program a clause they once removed from it, both can exploit the correct component of the initial program equally well. They differ in their use of the incorrect part of the of the program; Algorithm 7 views it as a hint for the search for a correction, but Algorithm 5 simply discards it.

**Algorithm 7: Interactive debugging**

*Given*: An equivalence relation $\approx$ on clauses,
an oracle for an interpretation $M$,
and an oracle for a well-founded ordering $\succ$ on goals.
*Input*: A program to be debugged, with a reason set associated with
each of its clauses, and a (possibly infinite) list of facts about $M$.
*Output*: A sequence of programs $P_1$, $P_2$, ..., each of which
is totally correct with respect to the known facts.
*Algorithm*:

```
let the set of marked clauses be the empty set.
read $P$, the program to be debugged, with a reason-set associated with
      each of its clauses.
repeat
   read the next fact.
   repeat
      if the program $P$ fails on a goal known to be true
      then find a true goal $A$ uncovered by $P$ using Algorithm 3,
         and mark it;
         ask which clause $p$ in $P$ with reason-set $R$ was supposed to cover
         modify $p$ to cover $R\cup\{A\}$ using the bug-correction algorithm.
      if the program $P$ succeeds on a goal known to be false
      then detect a false clause $p$ in $P$ using Algorithm 2;
         modify $p$ to an unmarked clause that cover $p$'s reason-set
         using the bug correction algorithm.
   until the program $P$ is correct with respect to the known facts.
   output $P$.
until no facts left to read.
if during a computation of $P$ a stack overflow occurs,
then apply Algorithm 4, the stack overflow diagnosis algorithm,
   which either detects a clause $p$ that is diverging with
   respect to $\succ$ and $M$, or a clause $p$ that is false in $M$;
   mark $p$ and modify it to an unmarked clause that cover $p$'s reason-set
   using the bug correction algorithm, and restart the computation. ▯
```

The task of the bug-correction algorithm is to search the equivalence class of $p$. Typically, the size of the equivalence class is exponential in the size of its members. In Section 5.2 below we consider restricted types of errors, and efficient ways to search equivalence classes they induce; in doing so we use the *refinement graph*, introduced in Section 4.5. The outcome is an algorithm that solves the bug-correction problem, given that the class of errors can be described via a refinement operator.

Algorithm 7 illustrates one possible application for a bug-correction algorithm. Its control-structure is rather atypical for an interactive debugger, as it requires the user to explicitly supply the output for inputs tried, and it automatically tries all previous input/output samples whenever a modification is made to the program. We find the control-structure used in the diagnosis system, described in Section 3.5, to be more natural. This control-structure is used by the interactive debugging system, described in Section 5.3.

## 5.2 A bug correction algorithm

We define common classes of bugs via refinement operators. The refinement operator is used in turn to define an equivalence class of clauses, to be searched by the bug-correction algorithm. The incorrect clause that needs to be modified serves as a hint for this search, by providing it with a starting point.

### 5.2.1 Describing errors via refinement operators

A refinement operator $\rho$ induces an equivalence relation $\approx_\rho$ on $L(\rho)$, as follows:

**Definition 5.2:** Let $\rho$ be a refinement operator. We say that $p\approx_\rho q$ iff either $p\leq_\rho q$ or $q\leq_\rho p$, or there is an $r$ such that $p\approx_\rho r$ and $r\approx_\rho q$.

We give examples of classes of typical errors and their associated refinement operators. The complexity of the refinement operators developed increases with the complexity of the errors that can be corrected using them, and since the union of two refinement operators is a refinement operator, one can easily combine two classes of errors, and treat them simultaneously.

The first class of errors we consider is misspelled variable names. We define a refinement operator $\rho_v$ for which $p \approx_{\rho_v} q$ iff $p$ can be obtained from $q$ by changing the names of some occurences of variables.

**Definition 5.3:** Let $p$ be a clause. Then $q$ is in $\rho_v(p)$ iff $q = p\{X \rightarrow Y\}$, where $X$ and $Y$ are variables that occur in $p$.

**Lemma 5.4:** If the clause $p$ can be obtained from the clause $q$ by changing some occurences of variables in $q$, then $p \approx_{\rho_v} q$.

**Proof:** Consider the clause $r$ obtained from $p$ by replacing each occurence of of a variable in $p$ with a new distinct variable name. Clearly $r \leq_{\rho_v} p$ and $r \leq_{\rho_v} q$. []

If we allow misspelled terms also, then the resulting class of errors is the *structure preserving errors* studied by Brooks [16] in the context of program testing.

Another class of errors we consider is errors in arithmetic test predicates ($<$, $\leq$, and $=$), such as a replacement of a $<$ test with a $\leq$ test, missing tests, superfluous tests, etc. The equivalence class induced by these errors is $\approx_{\rho_t}$, where $\rho_t$ is defined as follows:

**Definition 5.5:** Let $p$ be a clause. Then $q$ is in $\rho_t(p)$ iff $q$ can be obtained from $p$ by adding to $p$'s body the goal $t(X,Y)$, where $t$ is an arithmetic test predicate.

**Lemma 5.6:** If the clause $p$ can be obtained from the clause $q$ by adding and/or removing arithmetic test predicates, then $p \approx_{\rho_t} q$.

**Proof:** Consider $r$, the clause obtained from $p$ by removing from it all arithmetic test predicates. Clearly $r \leq_{\rho_t} p$ and $r \leq_{\rho_t} q$. []

In addition to these examples, any of the refinement operators discussed in the previous chapter can be used for bug correction as well, although they might be too general to be useful. For example, the

equivalence class induced by $\rho_2$, the refinement operator that generates definite clause grammars, is $L(\rho_2)$ itself, which means that a bug-correction algorithm based on this refinement operator can fix arbitrary bugs in grammar rules. In the current context, such generality may be viewed as a drawback rather than an advantage.

### 5.2.2 Searching the equivalence class

We describe an algorithm that solves the bug correction problem, for the case where the equivalence class is induced by refinement operators. Given a reason set and a clause, the algorithm searches for a refinement of the clause that covers the reason set. If it fails, then it *derefines* it, and iterates. The algorithm terminates when the clause can not be derefined. Intuitively, the derefinement operation generalizes the clause so it will cover the goals, by removing from it the components that prevent it from covering; a concrete derefinement procedure has to be developed for each refinement operator separately. The property it has to satisfy is as follows.

**Definition 5.7:** Let $\rho$ be a refinement operator and $p$ a clause. A *derefinement operator* for $\rho$ is a mapping from $\rho(L(\rho))$ to $L(\rho)$, such that if $q$ is a derefinement of $p$ then $q <_{\rho} p$.

The purpose of a derefinement operator is to climb up the refinement graph, where the particular direction and length of each step are not very important. Theoretically, we could have used the inverse of the refinement operator for that purpose. However, the inverse of the refinement operator returns a set of clauses rather than a single clause, and for typical refinement operators computing it is unnecessarily expensive.

Consider, for example, the refinement operator $\rho_t$. A derefinement operator for it is simple to define: remove some arithmetic test from $p$.

A derefinement operation for definite clause grammars, generated by $\rho_2$ (defined in Section 4.5) is also simple to define: remove the last goal from the body of $p$, if the body is not empty; otherwise, if the first argument of its head is not a variable, then replace it with a variable. A derefinement procedure for $\rho_v$ can be defined in a similar way.

Given a derefinement procedure, the bug-correction algorithm operates as shown below, in Algorithm 8.

**Algorithm 8: A bug-correction algorithm**

*Given*: A refinement operator $\rho$ and a derefinement procedure for it, an interpretation $M$, and a set of marked clauses.

*Input*: A set of goals $R$ in $M$ and a clause $p$.

*Output*: An unmarked clause $q$ that covers $R$ in $M$ and is equivalent to $p$ or *no clause found*

*Algorithm*:

```
repeat
        search for an unmarked refinement p that covers R in M
           using Algorithm 6.
        if a covering clause is found then return it,
        else if p has a derefinement then set p to it,
        else return no clause found
until a clause is found or p has no derefinement. ▯
```

The bug-correction algorithm is not guaranteed to find a clause as desired, even if such a clause exists; the reason is that the derefinement procedure may not climb high enough or in the right direction in the refinement graph. The exact conditions under which this algorithm finds a clause as desired are characterized by the following theorem.

**Theorem 5.8:** Assume that Algorithm 8 is applied to a set of goals $R$ and a clause $p$. If the set of clauses $q$ such that $q \approx_\rho p$ and $q$ covers $R$ in $M$ is finite, and if for every such $q$ there is a clause $r$, $r \leq_\rho q$ that can be obtained from $p$ by a sequence of derefinement operations, then Algorithm 8 will find an unmarked clause $q$ that covers $R$ in $M$.

**Proof:** By the assumption of the finiteness of the candidate space, every call to Algorithm 6 terminates. Let $r$ be the clause as in the assumption of the theorem. If no clause is found earlier by the algorithm, it will eventually derefine $p$ until it is equal to $r$. Since there is an unmarked clause $q$ that covers $R$ in $M$, for which $r \leq_\rho q$, such a clause will be found by Algorithm 6. ▯

## 5.3 An interactive debugging system

Similar to the previous systems, the interactive debugging system is composed of the diagnosis algorithms, an interactive shell and an error handler. The interactive shell is essentially identical to the one in the diagnosis system described in Section 3.5; the only difference is that the current system uses the monitoring interpreter that traps errors in the course of execution, described in Section 3.7. The error handling component is a somewhat rudimentary program editor, with an option for "automatic error correction". This option invokes Algorithm 8, the bug-correction algorithm, described in Section 5.2.

Since the system does not fit on one page (it requires a page and a half), we do not show it here. The code appears in Appendix II.

### 5.3.1 Debugging quicksort

Below we show a session with the system in which we interactively debug a quicksort program.

```
| ?- pds.
```

```
@[user].
Solving user...
| qsort([X|L],L0) ←
|        partition(L,X,L1,L2),
|        qsort(L1,L3), qsort(L2,L4),
|        append(L3,[X|L4],L0).
|
| ←declare(qsort(+[x],−[x]),[determinate,total]).
| ←declare_called(qsort(X,Y),
|        [qsort(_,_),partition(_,_,_,_),append(_,_,_)]).
|
| partition([X|L],Y,L1,[X|L2]) ←  partition(L,Y,L1,L2), X<Y.
| partition([X|L],Y,[X|L1],L2) ←  partition(L,Y,L1,L2), X≤Y.
| partition([],X,[],[]).
|
| ← declare(partition(+[x],+x,−[x],−[x]),[determinate,total]).
| ← declare_called(partition(X,Y,Z,U),
|        [partition(_,_,_,_),_<_,_≤_]).
|
| append([X|L1],L2,L3) ←  append([X|L1],L2,[X|L3]).
| append([],L,L).
|
| ← declare(append(+[x],+[x],−[x]),[determinate,total]).
| ← declare_called(append(X,Y,Z),[append(_,_,_)]).
|
| ↑Z

user consulted   172 words     1.00 sec.
```

We now have typed in our initial *qsort* program, some type, mode, and dependency declarations, and proceed to test it.

```
@qsort([2,1,3],X).
Solving qsort([2,1,3],X)...
```

```
Error trapped: no solution to partition([3],2,X,Y)
Error: missing solution partition([3],2,X,Y). diagnosing...
```

The monitoring interpreter trapped an error while solving the *partition* goal, and invoked the *missing_solution* diagnosis program.

```
Query: partition([3],2,X,Y)? y.
Which (X,Y)? [],[3].

Query: partition([],2,[],[])? y.
Error diagnosed: partition([3],2,[],[3]) is uncovered.
add (y) or (m)odify a clause? m.
which? partition([X|L],Y,L1,[X|L2]) ←  partition(L,Y,L1,L2), X<Y.
```

We have the option to add a new clause to the program, or modify a clause in the program to cover the uncovered goal. We chose to modify the clause we thought should have covered that goal. Since it does not cover it, it is first derefined to a covering clause.

```
Derefining (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),X<Z)
Found clause: (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W))
    after searching 1 clauses.
ok? n.
```

We do not like this clause, since we think it should contain an arithmetic test; so the system continues the search.

```
Found clause: (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),Z<X)
    after searching 5 clauses.
ok? y.
Listing of partition(X,Y,Z,U):
    (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z).
    (partition([],X,[],[]) ← true).
    (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),Z<X).
```

A clause is found to our satisfaction. The new clause can be obtained from the original one by exchanging the arguments of the < test. This modification requires one derefinement — which removes the erroneous test — and one refinement — which adds the correct test.

The system then automatically tries to solve the original goal, since an error was encountered during its execution.

```
Solving qsort([2,1,3],X)...
Error trapped: no solution to qsort([],X)
Error: missing solution qsort([],X). diagnosing...

Query: qsort([],X)? y.
Which X? [].
Error diagnosed: qsort([],[]) is uncovered.
add (y) or (m)odify a clause? y.
which? qsort([],[]).
Listing of qsort(X,Y):
    (qsort([X|Y],Z) ←
        partition(Y,X,W,X1),qsort(W,Z1),qsort(X1,V1),
        append(Z1,[X|V1],Z)).
    (qsort([],[]) ← true).
```

The second error the system finds is easy to correct: the base clause for *qsort* is missing, so we add it manually. The algorithm tries to solve the original goal again.

```
Solving qsort([2,1,3],X)...
Error: stack overflow on qsort([2,1,3],X). diagnosing...
append([1],[2,3],X) is looping.
Error diagnosed:
    (append([1],[2,3],X) ← append([1],[2,3],[1|X])) is diverging.
```

This time a stack overflow occurred; the error was diagnosed with no user intervention, since the stack overflow diagnosis algorithm found two duplicate goals on the stack. We request the system to modify the diverging clause. The system, in turn, prompted us for a reason for that

clause, so it would know how to modify it.

```
retract (y), (m)odify, or (r)eplace it? m.
What is a reason for (append([X|Y],Z,U) ← append([X|Y],Z,[X|U]))?
    append([1,2],[3,4],[1,2,3,4]).
Derefining (append([X|Y],Z,U) ← append([X|Y],Z,[X|U]))
Refining: (append([X|Y],Z,U) ← true)

Query: append([2],[3,4],[1,2,3,4])? n.

Query: append([3,4],[2],[1,2,3,4])? n.

Query: append([3,4],[],[1,2,3,4])? n.

Query: append([],[3,4],[1,2,3,4])? n.

Query: append([2],[4],[1,2,3,4])? n.

Query: append([4],[2],[1,2,3,4])? n.

Query: append([2],[3,4],[2,3,4])? y.
Found clause: (append([X|Y],Z,[X|V]) ← append(Y,Z,V))
  after searching 85 clauses.
ok? y.
Listing of append(X,Y,Z):
  (append([],X,X) ← true).
  (append([X|Y],Z,[X|V]) ← append(Y,Z,V)).
```

Since the system uses eager search strategy (see Section 4.4) we had to answer a few queries during the search process, but eventually a correct recursive definition of *append* was found. The system tries the original goal again.

```
Solving qsort([2,1,3],X)...
solution: qsort([2,1,3],[1,2,3]); ok? y.
```

```
    no (more) solutions. ok? y.

    @qsort([2,1,4,3],X).
    Solving qsort([2,1,4,3],X)...
    solution: qsort([2,1,4,3],[1,2,3,4]); ok? y.
    no (more) solutions. ok? y.
```

The program finally returns a correct output on the initial goal. We tried it on another goal, and it solved it correctly also. The *qsort* program constructed is now correct (this claim does not follow from the previous two; it is an independent observation). However, to demonstrate other aspects of the debugging system we proceed to "debug" it into a quicksort program that removes duplicates. The design decision we make is that duplicates are to be removed by *partition*, and we answer queries accordingly.

```
    @qsort([2,1,2,3],X).
    Solving qsort([2,1,2,3],X)...
    solution: qsort([2,1,2,3],[1,2,2,3]); ok? n.
    Error: wrong solution qsort([2,1,2,3],[1,2,2,3]). diagnosing...

    Query: qsort([1,2],[1,2])? y.

    Query: qsort([3],[3])? y.

    Query: append([1,2],[2,3],[1,2,2,3])? y.

    Query: partition([1,2,3],2,[1,2],[3])? n.

    Query: partition([2,3],2,[2],[3])? n.
    Error diagnosed:
          (partition([2,3],2,[2],[3]) ← partition([3],2,[],[3]),2≤2) is false.
          retract (y), (m)odify, or (r)eplace it? m.
    What is a reason for
          (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z)?
          partition([1],2,[1],[]).
```

```
    Refining: (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z)
    Found clause:
       (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z,X<Z)
       after searching 2 clauses.
    ok? y.
    Listing of partition(X,Y,Z,U):
       (partition([],X,[],[]) ← true).
       (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),Z<X).
       (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z,X<Z).
```

The system found that the test in the false clause was too weak, and it strengthens ≤ to <; however, since it has no built in knowledge about the semantics of ≤ and <, it cannot conclude that once the < test is added, the ≤ test becomes superfluous. It is conceivable that a post-processing stage can be added in which such an optimization can be incorporated.

```
    Solving qsort([2,1,2,3],X)...
    Error trapped: no solution to partition([2,3],2,X,Y)
    Error: missing solution partition([2,3],2,X,Y). diagnosing...

    Query: partition([2,3],2,X,Y)? y.
    Which (X,Y)? [],[3].
    Error diagnosed: partition([2,3],2,[],[3]) is uncovered.
    add (y) or (m)odify a clause? m.
    which? any.
```

The original *partition* program had no special clause for the case where the first element on the partitioned list and the partitioning element are equal. So we send the search algorithm off to look for such a clause.

```
    Refining: (partition(X,Y,Z,U) ← true)
    Checking: (partition([X|Y],Z,[],Y) ← true)
    Refuted: (partition([3],2,[],[]) ← true)
    Found clause: (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V))
       after searching 39 clauses.
```

```
ok? y.
Listing of partition(X,Y,Z,U):
    (partition([],X,[],[]) ← true).
    (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),Z<X).
    (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z,X<Z).
    (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V)).

Error: wrong solution partition([3],2,[],[]). diagnosing...
Error diagnosed: (partition([3],2,[],[]) ← partition([],2,[],[])) is false.
```

The clause found by the algorithm turns out to be false. So we ask the algorithm to modify it.

```
retract (y), (m)odify, or (r)eplace it? m.
What is a reason for (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V))?
        partition([2],2,[],[]).
Refining: (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V))
Found clause: (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V),X≤Z)
    after searching 3 clauses.
ok? y.
Listing of partition(X,Y,Z,U):
    (partition([],X,[],[]) ← true).
    (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),Z<X).
    (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z,X<Z).
    (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V),X≤Z).

Solving qsort([2,1,2,3],X)...
Error trapped: too many solutions to partition([1,2,3],2,X,Y)

Query: partition([1,2,3],2,[],[3])? n.
Error: wrong solution partition([1,2,3],2,[],[3]). diagnosing...
Error diagnosed:
        (partition([1,2,3],2,[],[3]) ← partition([2,3],2,[],[3]),1≤2) is false.
retract (y), (m)odify, or (r)eplace it? m.
What is a reason for
```

```
        (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V),X≤Z)?
        partition([2],2,[],[]).
Refining: (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V),X≤Z)
Found clause: (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V),X≤Z,Z≤X)
    after searching 2 clauses.
ok? y.
```

And after two more modifications, a correct clause is found. Due to the peculiarities of the refinement operator we use, and our not specifying = as one of the procedures called by *partition*, the restriction that $X$ and $Z$ need to be equal in that clause is expressed via the double $\leq$ test. Again, one can conceive of an optimization that replaces this test with an equality test.

```
Listing of partition(X,Y,Z,U):
    (partition([],X,[],[]) ← true).
    (partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W),Z<X).
    (partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W),X≤Z,X<Z).
    (partition([X|Y],Z,U,V) ← partition(Y,Z,U,V),X≤Z,Z≤X).

Solving qsort([2,1,2,3],X)...
solution: qsort([2,1,2,3],[1,2,3]); ok? y.
no (more) solutions. ok? y.
```

The final program found by the algorithm is shown below:

```
qsort([X|Y],Z) ←
        partition(Y,X,W,X1), qsort(W,Z1), qsort(X1,V1),
        append(Z1,[X|V1],Z).
qsort([],[]).

partition([],X,[],[]).
partition([X|Y],Z,U,[X|W]) ← partition(Y,Z,U,W), Z<X.
partition([X|Y],Z,[X|V],W) ← partition(Y,Z,V,W), X≤Z, X<Z.
partition([X|Y],Z,U,V) ← partition(Y,Z,U,V), X≤Z, Z≤X.

append([],X,X).
append([X|Y],Z,[X|V]) ← append(Y,Z,V).
```

It sorts and removes duplicates correctly, although the arithmetic tests in it are not the most elegant. The session took 41 CPU seconds, and 20 facts.

It may be interesting to note that synthesizing quicksort from examples seems to be beyond the current power of the Model Inference System; it can synthesize the program for *append* and *partition*, but the search space for the recursive clause of *qsort* is too large for the current implementation. The reason for that is not so much the size of the clause, but the fact that the procedures it calls are total, which prevents effective pruning of the search space: if a clause covers a goal and we add to its body a goal that is total, that clause still covers the goal. Since the branching factor for the refinement graph for quicksort can be up to 50 at the depth in which the clause is searched, the ineffectiveness of the pruning strategy results in a search space beyond the capabilities of the current implementation.

# Chapter 6

# CONCLUSIONS

One role of theory is to solve in an abstract setting problems that arise from practical experience, and provide a way for these solutions to apply back to the original, concrete problems. Since the major limits we have encountered so far in Computer Science are the limits of our imagination, I see the value of theory not only in its applicability to current problems, but also in its ability to project into the future, to point out new directions of development, and to help evaluate current trends.

In these remarks I would like to examine the results of this thesis in light of this methodology, and draw from them implications concerning the merit of different trends in programming languages and programming methodology.

## 6.1 Algorithmic debugging

Program debugging is a messy problem; so much so that the goal of many was to develop methodologies and tools that would eliminate the need to face it altogether. I see the main contribution of this thesis as showing that this problem is amenable to theoretical treatment, and that this treatment yields useful, practical results.

A word of caution is due. Because of the theoretical nature of this work, and the fact that it evolved mostly in the context of inductive inference, I have not implemented the debugging algorithms for full Prolog, and as a consequences did not use them extensively in real programming tasks.

However, I have reasons to think that it is valid to extrapolate from the sterile environment of inductive inference to the world of real debugging. The main reason is that the diagnosis algorithms are, in some sense, just an abstraction and elaboration of what programmers do intuitively. For example, since I invented the diagnosis technique for finite failure, I have been using its underlying mechanism within the standard Prolog debugger. Even though the algorithm is not implemented within the debugger, it can be simulated manually by a sequence of commands that the debugger understands. I found the systematic use of it, even in such a crude form, to be a good diagnosis technique. The divide-and-query algorithm can not be easily simulated manually, but I suspect that the same conclusion holds.

The diagnosis algorithm for stack overflow can also be viewed in this way. When a stack overflow occurs, all the programmer really wants to do is check the stack to see what's going on. The diagnosis algorithm enables the programmer to do so, but in a structured way, focusing his attention on the suspicious component of the stack, i.e., the part of the stack that contains a looping procedure call.

A simple implementation of the first two diagnosis algorithms is available in micro-Prolog [57]; their incorporation in other Prolog systems and their adaptation to Pascal is also being investigated [73, 74].

## 6.2 Incremental inductive inference

The second main result of the thesis is the development of a general, incremental inductive inference algorithm, that is simple enough to be analyzed theoretically, and is amenable to an implementation that compares favorably with other inductive inference systems. We attribute this success to the choice of logic as the target computational model.

The most important aspect of the inductive synthesis algorithm is that it is incremental, which enables gradual, evolutionary construction of the hypothesis. It is easier to achieve this incremental behavior in logic, since the semantics of a logical axiom is independent of the context in which this axiom occurs. If an axiom is discovered to be false, then it is false no matter what theory it is included in, and hence should be discarded and never tried again. The same cannot be said of Pascal statements, Lisp condition—action pairs, or Turing machine transitions.

Even though the algorithm is incremental, it has a search component. For any reasonable computational model, there are exponentially many programs of any given length; the same is true of logic. However, the pruning strategy we have developed is able to cope somewhat with this complexity. Its strategy critically depends on the intimate relationship between the syntax and semantics of logic, a relation rarely found in other computational models.

## 6.3 Prolog as a research tool

I think that using Prolog as the research tool played an invaluable role in achieving these results; in the following I describe the way this research has developed to help the reader appreciate this fact. The original problem I was concerned with was that of scientific discovery: I was intrigued by Popper's ideas on the role of refutations in scientific progress [71, 72], and wanted to test their applicability to computerized inductive inference. The first step was the development of the Contradiction Backtracing Algorithm [82, 84], which formalizes the notion of crucial experiments in science. This algorithm can detect a false axiom in a theory with a false conclusion by testing whether certain ground atoms are true, and is the precursor of the diagnosis algorithms. I then incorporated the algorithm into a general inductive inference algorithm, following ideas of Gold [37] and the Blums [13].

The next logical step was to implement the algorithm and test it. Although I did not know Prolog at the time, rumors suggested that it would be the ideal implementation language. Only after a restricted version of the the algorithm was implemented in Prolog did it occur to me that the Contradiction Backtracing Algorithm was applicable to program debugging, and that the inductive inference algorithm could be used to synthesize logic programs from examples. Since then, attempts to improve the performance of the Model Inference System are responsible for almost all of the theoretical development that ensued.

One example is the algorithm that diagnoses finite failure. The original inductive inference algorithm added axioms to the theory indiscriminately, in increasing order of size, when it found the theory to be too weak. The natural improvement (which was natural to Drew McDermott, not to me) was to add to the theory only axioms that are truly useful in the derivation. The question of how to detect these axioms effectively remained open for a while. The diagnosis algorithm for finite failure was developed to solve this problem.

Another example is the algorithm that diagnoses nontermination. For a while I thought that the synthesis of nonterminating programs could be avoided by restricting the syntactic structure of the target programs. However, when I attempted to synthesize nontrivial programs, this restriction became harder and harder to maintain.

A theoretical solution to the problem is available in the work of the Blums [13]: allow nonterminating programs, but execute them with an interpreter that can be supplied with a resource bound. This solution was never implemented in the Model Inference System since it required the specification of complexity bounds for which I had no intuitive *a priori* estimate, and since programs synthesized by such a system are useless outside of this artificial environment.

Indeed, people who experimented with the previous incarnation of the system [85], incorporated with refinement operators that I thought would generate only terminating programs, complained that ever so often the system would succeed in synthesizing a clever looping program, and would not terminate thereafter. This "theoretical bug" in the Model Inference System was not fixed until the development of the algorithm that diagnosed nontermination, and the incarnation of the Model Inference System

described in the thesis is the first to be correct in this sense.

These examples show that attempts to implement a theory can lead to its enhancement and development. But this is not necessarily the case: if I had implemented the inductive inference algorithm in assembly language I might have succeeded, but the theory would have remained untouched. Due to the nature of Prolog, the problems I faced in the implementation were interesting and abstract enough to stimulate further theoretical research, and to make results of this research relevant to their solution.

Prolog helped extend the theory in another way, by allowing easy experimentation with different ideas. For example, the three search strategies described in Section 4.4 originated in some Prolog hacks, whose goal was to reduce the number of queries I had to answer during the synthesis process. The new search strategies are responsible for the more interesting applications of the Model Inference System: without the development of adaptive search, for example, the inference of nontrivial context-free grammars would have been prohibitively expensive in human resources. Later, these hacks were abstracted, and a theory, spanning about a dozen pages of this thesis, was developed to justify them. Although these extensions were difficult to come by conceptually, they were easy to implement once they were conceived: the code that implements the three search strategies is but 32 lines long (See Appendix II, page 188).

## 6.4 Prolog versus Lisp

The importance of implementation in the development of theories has been the bread and butter of Artificial Intelligence since its beginnings, and the need for a high level language in such an endeavor was also recognized. However, I think that Lisp, the language of choice of AI for many years, has failed to fulfill this promise. With few exceptions, the major AI systems have failed to come up with a clean, precise, and mathematically valid theory that describes their underlying mechanism and explains their performance. As a consequence, it is hard to use past achievements as building blocks for new theories, and the structure of the resulting science is "flat".

One justification given to this situation is that the real world is messy, and if you want to solve real world problems you necessarily end up with vague, imprecise theories. I do not think that this is a valid excuse for theoretical sloppiness. Even though it is unresolved whether the underlying mechanisms of the universe are simple and comprehensible, all successful sciences search for the simplest principles that will help comprehend it. Computer Science has a great advantage over other experimental sciences in this respect: the world we investigate is our own creation, and, to a large degree, we are the ones to determine if it is simple or messy. I think I have demonstrated this principle in the case of debugging: debugging can be a messy problem, if one creates a messy environment in which it has to be solved, but it is not inherently so. The theory of algorithmic debugging suggests that if the programming language is simple enough, then programs in it can be debugged on the basis of several simple principles.

Therefore I conjecture that there are other reasons for this phenomenon — the lack of mathematically valid theories emerging from AI implementations. One of them may be the use of Lisp as the main research tool. Although Lisp is considered a high level language, it is not clear that it encourages precision and clarity of thought. As one Lisp hacker puts it [91]: "Lisp...is like a ball of mud. You can add any amount of mud to it...and it still looks like a ball of mud!". This aspect of Lisp is the hacker's delight, but the theoretician's nightmare. It is known that one may implement without too much effort a reasonable Prolog in Lisp. However, the issue is not implementation — the issue is the method of thought. Based on my experience with both languages I maintain that one's thoughts are better organized, and one's solutions are clearer and more concise, if one thinks in Prolog rather than in Lisp.

Contrasting my approach and the approach of MIT's Programmer's Apprentice Project [77] will sharpen these differences in research methodologies. The aspirations of the Programmer's Apprentice Project are similar in depth and much wider in scope than those of this thesis. However, since they commited themselves to supporting the full arsenal of Lisp hacks, including *rplaca*, *rplacd* and the like, they were bound to resort

to state transition semantics, and by doing so gave up the potential advantage of Lisp as a functional programming language. As a result, the complexity of the formalism they need in order to explain even the simplest program seems forbidding [76].

My approach was to deliberately restrict the target language to pure Prolog, at least in the initial stage of the research. As a result I succeeded in coming up with a clean, relatively simple theory. Surprisingly enough, the application of this theory to most of Prolog's extensions from its pure core seems to be not very difficult, as discussed in Section 3.6.

Since no analytic argument can irrevocably resolve a methodological disagreement, and since both research directions are in the experimental stage right now, I believe that only the future (perhaps with a little help from Japan) will determine who is right.

## 6.5 Programming environments and simplicity

I would like to reflect on the choice of Prolog both as the target language and the implementation language of the debugging algorithms, in light of Sandewall's discussion on programming environments [81]. Sandewall lists the following properties a programming language must have to support the development of a programming environment for it:

- "*Bootstrapping*. An obvious choice is to implement the system itself in the language it supports; then one needs to work only with a single language, and the system supports its own development.
- *Incrementality*. To achieve real interaction, the basic cycle of the programming system should be to read an expression from the user, execute it, and print out the results while preserving global side effects to its database. The expression itself may of course contain such things as procedure calls.
- *Procedure oriented*. For obvious reasons the language chosen should be procedure oriented.
- *Internal representation of programs*. Since most of the

operations [of a programming environment] are operations on programs, the language should make it as easy as possible to operate on programs. Therefore, there should be a predefined system-wide internal representation of programs... This structure should be a data-structure *in* the programming language, so that user-written programs may inspect the structure and generate new programs."

It so happens that these are exactly the properties a programming language must have to support an implementation of our debugging algorithms. Sandewall made this list with Lisp in mind, apparently unaware of Prolog at the time. However, I claim that Prolog scores on all these issues just as well as Lisp does, and on the most important of them — bootstrapping — even better. As shown in the Prolog introduction (Program 3, Page 29), a pure Prolog interpreter written in pure Prolog is just three simple clauses. All our diagnosis algorithms are augmented interpreters, hence the ease of implementing and experimenting with them correlates directly with the ease of implementing and augmenting an interpreter for the language within the language.

However, one important property is missing from Sandewall's list:

- *Simple semantics*. The future programming environment will not only manipulate programs, but also reason about them. To facilitate this task, the mapping from programs to their meanings should be simple and transparent.

I think that Prolog scores better than other programming languages on this important point as well.

This argument generalizes 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 programs be *L* itself. This can be accomplished only if *L* can interpret *L*-programs in a natural way.

One conclusion from this argument is that kitchen-sink languages such as Ada will lose in the long run, since the effort required to create a programming environment that understands all their features will be

insurmountable.

# Appendix I. Applications of the Model Inference System

## I.1 Inferring insertion sort

Below is an example of a session in which the Model Inference System, equipped with an eager search strategy, infers a Prolog program for insertion sort.

```
?- mis.

Next fact? isort([2,3,1],[1,2,3]),true.
Checking fact(s)...
Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...
Error diagnosed: isort([2,3,1],[1,2,3]) is uncovered.

Searching for a cover to isort([2,3,1],[1,2,3])...
Declare isort(X,Y)? declare(isort(+[x],-[x]),[total,determinate]).
Procedures called by isort(X,Y)? isort(_,_), insert(_,_,_).

Query: isort([3,1],[1,2,3])? n.

Query: insert(2,[3,1],[1,2,3])? n.

Query: isort([1],[1,2,3])? n.

Query: insert(2,[1],[1,2,3])? n.

Query: insert(3,[1],[1,2,3])? n.

Query: isort([3,1],X)? y.
Which X? [1,3].
```

```
Query: insert(2,[1,3],[1,2,3])? y.
Found clause: (isort([X|Y],Z)←isort(Y,V),insert(X,V,Z))
    after searching 61 clauses.
Listing of isort(X,Y):
    (isort([X|Y],Z)←isort(Y,V),insert(X,V,Z)).
```

After searching through 61 clauses, the system found a clause that covers *isort*([2,3,1],[1,2,3]), which happens to be also the right recursive definition of insertion sort. It found the necessary facts to determine the coverage by querying the user, as it uses eager search strategy. An unpruned breadth-first search through this refinement graph would have gone through 165 clauses before finding this clause.

```
Checking fact(s)...
Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...

Query: isort([1],X)? y.
Which X? [1].

Query: isort([],X)? y.
Which X? [].
Error diagnosed: isort([],[]) is uncovered.
```

And the system found that the base case for *isort* is missing; the facts known already by the system saved two queries for the diagnosis algorithm: *isort*([2,3,1],*X*), and *isort*([3,1],*X*).

```
Searching for a cover to isort([],[])...
Found clause: (isort([],[])←true)
    after searching 5 clauses.
Listing of isort(X,Y):
    (isort([X|Y],Z)←isort(Y,V),insert(X,V,Z)).
    (isort([],[])←true).

Checking fact(s)...
```

```
Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...

Query: insert(1,[],[1])? y.
Error diagnosed: insert(1,[],[1]) is uncovered.

Searching for a cover to insert(1,[],[1])...
```

Although we haven't supplied the system originally with facts about *insert*, the eager search strategy, together with the initialization information that *isort* calls *insert*, found facts about insert. Now the diagnosis algorithms leads the system to find the base clause for *insert*. We are first prompted for some initialization information.

```
Declare insert(X,Y,Z)? declare(insert(+x,+[x],-[x]),[total,determinate])
Procedures called by insert(X,Y,Z)? insert(_,_,_), _≤_.
Found clause: (insert(X,Y,[X|Y])←true)
   after searching 8 clauses.
Listing of insert(X,Y,Z):
   (insert(X,Y,[X|Y])←true).
```

The clause that was found is incorrect, but is consistent with the facts we have supplied to the system so far.

```
Checking fact(s)...
Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...

Query: insert(3,[1],[1,3])? y.
Error diagnosed: insert(3,[1],[1,3]) is uncovered.

Searching for a cover to insert(3,[1],[1,3])...

Query: insert(3,[],[1,3])? n.

Query: insert(1,[],[1,3])? n.
Found clause: (insert(X,[Y|Z],[Y,X|Z])←true)
```

```
   after searching 27 clauses.
Listing of insert(X,Y,Z):
   (insert(X,Y,[X|Y])←true).
   (insert(X,[Y|Z],[Y,X|Z])←true).

Checking fact(s)...no error found.
```

Everything seems fine to the system, so we break to Prolog, and test the resulting program.

```
Next fact? break.

[ Break (level 1) ]
| ?- isort([2,3,1],X).

X = [2,3,1]

[1] yes
```

We find an error, so we tell the system about it.

```
| ?- ↑Z
[ End Break (level 1) ]
Next fact? isort([2,3,1],[2,3,1]),false.
Checking fact(s)...
Error: wrong solution isort([2,3,1],[2,3,1]). diagnosing...

Query: insert(3,[1],[3,1])? n.
Error diagnosed: (insert(3,[1],[3,1])←true) is false.
Listing of insert(X,Y,Z):
   (insert(X,[Y|Z],[Y,X|Z])←true).

Checking fact(s)...
Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...
Error diagnosed: insert(1,[],[1]) is uncovered.
```

```
Searching for a cover to insert(1,[],[1])...
Found clause: (insert(X,[],[X])←true)
   after searching 9 clauses.
Listing of insert(X,Y,Z):
   (insert(X,[Y|Z],[Y,X|Z])←true).
   (insert(X,[],[X])←true).

Checking fact(s)...no error found.
```

Again the system is in equilibrium, so we test the program.

```
| ?- isort([2,3,1],X).

X = [1,2,3] ;

[1] no
| ?- isort([2,1,3],X).

X = [3,2,1]

[1] yes
```

The program behaves correctly on the original input, but not on the new input tried.

```
Next fact? isort([2,1,3],[3,2,1]),false.
Checking fact(s)...
Error: wrong solution isort([2,1,3],[3,2,1]). diagnosing...

Query: isort([3],[3])? y.

Query: isort([1,3],[3,1])? n.

Query: insert(1,[3],[3,1])? n.
```

```
Error diagnosed: (insert(1,[3],[3,1])←true) is false.
Listing of insert(X,Y,Z):
   (insert(X,[],[X])←true).

Checking fact(s)...
Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...
Error diagnosed: insert(3,[1],[1,3]) is uncovered.

Searching for a cover to insert(3,[1],[1,3])...

Query: insert(3,[],[3])? y.

Query: insert(1,[],[3])? n.
Found clause: (insert(X,[Y|Z],[Y,X|Z])←Y≤X)
   after searching 45 clauses.
Listing of insert(X,Y,Z):
   (insert(X,[],[X])←true).
   (insert(X,[Y|Z],[Y,X|Z])←Y≤X).

Checking fact(s)...no error found.

Next fact? break.

[ Break (level 1) ]
| ?- isort([2,1,3],X).

[1] no
| ?- ↑Z
[ End Break (level 1) ]
Next fact? isort([2,1,3],[1,2,3]),true.
Checking fact(s)...
Error: missing solution isort([2,1,3],[1,2,3]). diagnosing...

Query: isort([1,3],X)? y.
Which X? [1,3].
```

*Query: isort([3],X)?* y.
*Which X?* [3].

*Query: insert(1,[3],[1,3])?* y.
*Error diagnosed: insert(1,[3],[1,3]) is uncovered.*

*Searching for a cover to insert(1,[3],[1,3])...*
*Found clause: (insert(X,[Y|Z],[X,Y|Z])←X≤Y)*
*after searching 32 clauses.*
*Listing of insert(X,Y,Z):*
*(insert(X,[],[X])←true).*
*(insert(X,[Y|Z],[Y,X|Z])←Y≤X).*
*(insert(X,[Y|Z],[X,Y|Z])←X≤Y).*

*Checking fact(s)...no error found.*

There is not evident error in the program constructed, so we test it again,

```
| ?- isort([2,1,3],X).

X = [1,2,3] ;

[1] no
| ?- isort([3,2,1],X).

X = [1,3,2]

[1] yes
```

and find an error, which we inform the system via the following fact.

*Next fact?* isort([3,2,1],[1,3,2]),false.
*Checking fact(s)...*

*Error: wrong solution isort([3,2,1],[1,3,2]). diagnosing...*

*Query: isort([2,1],[1,2])?* y.

*Query: insert(3,[1,2],[1,3,2])?* n.
*Error diagnosed: (insert(3,[1,2],[1,3,2])←1≤3) is false.*
*Listing of insert(X,Y,Z):*
*(insert(X,[],[X])←true).*
*(insert(X,[Y|Z],[X,Y|Z])←X≤Y).*

*Checking fact(s)...*
*Error: missing solution isort([2,3,1],[1,2,3]). diagnosing...*
*Error diagnosed: insert(3,[1],[1,3]) is uncovered.*

*Searching for a cover to insert(3,[1],[1,3])...*

*Query: insert(3,[],[])?* n.

*Query: insert(1,[],[])?* n.
*Found clause: (insert(X,[Y|Z],[Y|V])←insert(X,Z,V),Y≤X)*
*after searching 51 clauses.*
*Listing of insert(X,Y,Z):*
*(insert(X,[],[X])←true).*
*(insert(X,[Y|Z],[X,Y|Z])←X≤Y).*
*(insert(X,[Y|Z],[Y|V])←insert(X,Z,V),Y≤X).*

*Checking fact(s)...no error found.*

The system needed 36 seconds CPU time to synthesize the program, and 30 facts. 5 facts were supplied by the user:

*isort([2,3,1],[1,2,3]), true.*
*isort([2,3,1],[2,3,1]), false.*
*isort([2,1,3],[3,2,1]), false.*
*isort([2,1,3],[1,2,3]), true.*
*isort([3,2,1],[1,3,2]), false.*

13 facts were queried by the search algorithm:

```
isort([3,1],[1,2,3]), false.
insert(2,[3,1],[1,2,3]), false.
isort([1],[1,2,3]), false.
insert(2,[1],[1,2,3]), false.
insert(3,[1],[1,2,3]), false.
isort([3,1],[1,3]), true.
insert(2,[1,3],[1,2,3]), true.
insert(3,[],[1,3]), false.
insert(1,[],[1,3]), false.
insert(3,[],[3]), true.
insert(1,[],[3]), false.
insert(3,[],[]), false.
insert(1,[],[]), false.
```

and 12 facts were queried by the diagnosis algorithms.

```
isort([1],[1]), true.
isort([],[]), true.
insert(1,[],[1]), true.
insert(3,[1],[1,3]), true.
isort([1,3],[1,3]), true.
insert(3,[1],[3,1]), false.
isort([3],[3]), true.
isort([1,3],[3,1]), false.
insert(1,[3],[3,1]), false.
insert(1,[3],[1,3]), true.
isort([2,1],[1,2]), false.
insert(3,[1,2],[1,3,2]), false.
```

## I.2 Inferring a context-free grammar

In this session we apply the Model Inference system, equipped with the refinement operator $\rho_2$ described in Section 4.5 and an adaptive search

strategy, to the inference of a context-free grammar for a subset of Pascal.

We have declared the the set of terminal and nonterminal symbols:

```
terminal(X) ← member(X,
   [begin, end, if, then, while, do, call, ':=', ';',
   '(', ')', +, - ,* , /, <, =]).
nonterminal(N) ← member(N,[
   statement(X,Y), statementlist(X,Y), condition(X,Y),
   expression(X,Y), term(X,Y), factor(X,Y), ident(X,Y), number(X,Y),
   comparator(X,Y), plusminus(X,Y), timesdiv(X,Y) ]).
```

The system was given definitions for some of the nonterminals:

```
ident([X|Xs],Xs) ← atom(X), \+terminal(X).
number([X|Xs],Xs) ← integer(X).
plusminus ⇒ [+] | [-].
timesdiv ⇒ [*] | [/].
comparator ⇒ ['='] | ['<'].
```

and these terminals were declared to be system predicates, i.e. to have correct programs from the point of view of the Model Inference System. We then invoked the system, and provided it with facts concerning Pascal expressions. The order of facts takes into account the hierarchical relationship between the nonterminals *expression*, *term* and *factor*.

```
| ?- mis.

Next fact? factor([a],[]),true.
Checking fact(s)...
Error: missing solution factor([a],[]). diagnosing...
Error diagnosed: factor([a],[]) is uncovered.

Searching for a cover to factor([a],[])...
Checking: (factor(X,Y)←ident(X,Y))
Found clause: (factor(X,Y)←ident(X,Y))
   after searching 3 clauses.
Listing of factor(X,Y):
   (factor(X,Y)←ident(X,Y)).
```

```
Checking fact(s)...no error found.

Next fact? factor([1],[]),true.
Checking fact(s)...
Error: missing solution factor([1],[]). diagnosing...
Error diagnosed: factor([1],[]) is uncovered.

Searching for a cover to factor([1],[])...
Checking: (factor(X,Y)←number(X,Y))
Found clause: (factor(X,Y)←number(X,Y))
   after searching 3 clauses.
Listing of factor(X,Y):
   (factor(X,Y)←ident(X,Y)).
   (factor(X,Y)←number(X,Y)).

Checking fact(s)...no error found.

Next fact? term([a],[]),true.
Checking fact(s)...
Error: missing solution term([a],[]). diagnosing...
Error diagnosed: term([a],[]) is uncovered.

Searching for a cover to term([a],[])...
Checking: (term(X,Y)←factor(X,Y))
Found clause: (term(X,Y)←factor(X,Y))
   after searching 4 clauses.
Listing of term(X,Y):
   (term(X,Y)←factor(X,Y)).

Checking fact(s)...no error found.
```

We have established the simple definitions of *factor* and *term*, and are ready for more complex examples.

```
Next fact? expression([a,+,1],[]),true.
Checking fact(s)...
Error: missing solution expression([a,+,1],[]). diagnosing...
Error diagnosed: expression([a,+,1],[]) is uncovered.

Searching for a cover to expression([a,+,1],[])...
Checking: (expression(X,Y)←term(X,U),plusminus(U,W),term(W,Y))
Found clause: (expression(X,Y)←term(X,U),plusminus(U,W),term(W,Y)
   after searching 29 clauses.
Listing of expression(X,Y):
   (expression(X,Y)←term(X,U),plusminus(U,W),term(W,Y)).

Checking fact(s)...no error found.

Next fact? term([a,*,b],[]),true.
Checking fact(s)...
Error: missing solution term([a,*,b],[]). diagnosing...

Query: factor([a,*,b],[])? n.
Error diagnosed: term([a,*,b],[]) is uncovered.

Searching for a cover to term([a,*,b],[])...
Checking: (term(X,Y)←factor(X,U),timesdiv(U,W),term(W,Y))
Found clause: (term(X,Y)←factor(X,U),timesdiv(U,W),term(W,Y))
   after searching 20 clauses.
Listing of term(X,Y):
   (term(X,Y)←factor(X,Y)).
   (term(X,Y)←factor(X,U),timesdiv(U,W),term(W,Y)).

Checking fact(s)...no error found.

Next fact? expression([a],[]),true.
Checking fact(s)...
Error: missing solution expression([a],[]). diagnosing...
```

```
    Query: term([a],X)? y.
    Which X? [].
    Which X? n.
```

Nonterminals in context-free grammars are not determinate procedures, hence an existential query may have several different solutions. We supply all solutions (in this case there is only one), following by *no*.

```
    Error diagnosed: expression([a],[]) is uncovered.

    Searching for a cover to expression([a],[])...
    Checking: (expression(X,Y)←term(X,Y))
    Found clause: (expression(X,Y)←term(X,Y))
       after searching 5 clauses.
    Listing of expression(X,Y):
       (expression(X,Y)←term(X,U),plusminus(U,W),term(W,Y)).
       (expression(X,Y)←term(X,Y)).

    Checking fact(s)...no error found.

    Next fact? expression([a,+,'(',2,*,b,')'],[]),true.
    Checking fact(s)...
    Error: missing solution expression([a,+,(,2,*,b,)],[]). diagnosing...

    Query: term([a,+,(,2,*,b,)],X)? y.
    Which X? [+,'(',2,*,b,')'].
    Which X? n.

    Query: term([(,2,*,b,)],[])? y.

    Query: factor([(,2,*,b,)],[])? y.
    Error diagnosed: factor([(,2,*,b,)],[]) is uncovered.

    Searching for a cover to factor([(,2,*,b,)],[])...
    Checking: (factor(X,Y)←expression(X,Y))
```

```
    Found clause: (factor(X,Y)←expression(X,Y))
       after searching 5 clauses.
    Listing of factor(X,Y):
       (factor(X,Y)←ident(X,Y)).
       (factor(X,Y)←number(X,Y)).
       (factor(X,Y)←expression(X,Y)).

    Checking fact(s)...
    Error: stack overflow on term([a,*,b],[]). diagnosing...
```

The following three of the grammar rules found by the system so far

$expression(X,Y) \leftarrow term(X,Y).$
$factor(X,Y) \leftarrow expression(X,Y).$
$term(X,Y) \leftarrow factor(X,Y).$

together form a nonterminating program, hence the stack overflow. The stack overflow diagnosis program is then invoked.

```
    term([a,*,b],[]) is looping.
    Is (term([a,*,b],[]),factor([a,*,b],[])) a legal call? y.
    Is (factor([a,*,b],[]),expression([a,*,b],[])) a legal call? n.
    Error diagnosed: (factor([a,*,b],[])←expression([a,*,b],[])) is diverging.
```

We have resolved the trilemma using the knowledge of the hierarchical relationship between the three nonterminals, *expression*, *term*, and *factor*, and the system responds accordingly.

```
    Listing of factor(X,Y):
       (factor(X,Y)←ident(X,Y)).
       (factor(X,Y)←number(X,Y)).

    Error: missing solution expression([a,+,(,2,*,b,)],[]). diagnosing...
    Error diagnosed: factor([(,2,*,b,)],[]) is uncovered.

    Searching for a cover to factor([(,2,*,b,)],[])...
```

```
   Checking: (factor(X,Y)←expression(X,Y))
   Looping: (factor([a,*,b],[])←expression([a,*,b],[]))
   Checking: (factor(X,Y)←term(X,Y))
   Refuted: (factor([a,*,b],[])←term([a,*,b],[]))
   Checking: (factor([(|X],Y)←expression(X,[)|Y]))
   Found clause: (factor([(|X],Y)←expression(X,[)|Y]))
      after searching 25 clauses.
   Listing of factor(X,Y):
      (factor(X,Y)←ident(X,Y)).
      (factor(X,Y)←number(X,Y)).
      (factor([(|X],Y)←expression(X,[)|Y])).

   Checking fact(s)...no error found.
```

The session so far lasted 95 CPU seconds. The grammar produced is a full grammar for PL0 arithmetic expressions:

```
      (expression(X,Y)←term(X,U),plusminus(U,W),term(W,Y)).
      (expression(X,Y)←term(X,Y)).

      (term(X,Y)←factor(X,Y)).
      (term(X,Y)←factor(X,U),timesdiv(U,W),term(W,Y)).

      (factor(X,Y)←ident(X,Y)).
      (factor(X,Y)←number(X,Y)).
      (factor([(|X],Y)←expression(X,[)|Y])).
```

12 facts were needed to infer this grammar:

```
   fact(factor([a],[]),true)
   fact(factor([1],[]),true)
   fact(term([a],[]),true)
   fact(expression([a,+,1],[]),true)
   fact(term([a,*,b],[]),true)
   fact(expression([a],[]),true)
   fact(term([a],[]),true)
   fact(expression([a,+,(,2,*,b,)],[]),true)
   fact(term([a,+,(,2,*,b,)],[+,(,2,*,b,)]),true)
   fact(term([(,2,*,b,)],[]),true)
   fact(factor([(,2,*,b,)],[]),true)
   fact(factor([a,*,b],[]),false)
```

We then tried to infer a grammar-rule for the assignment statement.

```
   Next fact? statement([a,:=,a,+,1],[]),true.
   Checking fact(s)...
   Error: missing solution statement([a,:=,a,+,1],[]). diagnosing...
   Error diagnosed: statement([a,:=,a,+,1],[]) is uncovered.

   Searching for a cover to statement([a,:=,a,+,1],[])...
   Checking: (statement(X,Y)←expression(X,[:=|U]),expression(U,Y))
   Found clause: (statement(X,Y)←expression(X,[:=|U]),expression(U,Y))
      after searching 27 clauses.
   Listing of statement(X,Y):
      (statement(X,Y)←expression(X,[:=|U]),expression(U,Y)).

   Checking fact(s)...no error found.
```

The rule the system came up with was too general, so we gave it the following fact in attempt to restrict it to a correct one.

```
   Next fact? statement([a,+,1,:=,a,+,2],[]),false.
   Checking fact(s)...
   Error: wrong solution statement([a,+,1,:=,a,+,2],[]). diagnosing...
```

```
Query: expression([a,+,1,:=,a,+,2],[:=,a,+,2])? y.

Query: term([a,+,2],[+,2])? y.

Query: term([2],[])? y.

Query: expression([a,+,2],[])? y.
Error diagnosed:
      (statement([a,+,1,:=,a,+,2],[])←expression([a,+,1,:=,a,+,2],
      [:=,a,+,2]),expression([a,+,2],[])) is false.
Listing of statement(X,Y):
Checking fact(s)...
Error: missing solution statement([a,:=,a,+,1],[]). diagnosing...
Error diagnosed: statement([a,:=,a,+,1],[]) is uncovered.
```

The error was diagnosed, and the search for a new clause begins. After giving the system three more negative facts:

```
statement([a,*,2,:=,a],[]),false.
statement([a,*,2,:=,a],[]),false.
statement([1,:=,a],[]),false.
```

The system went through some wrong alleys, but finally came up with the right rule:

```
(statement(X,Y)←ident(X,[:=|U]),expression(U,Y)).
```

From that point things became easier; we then gave the system the following facts:

```
fact(condition([2,*,a,=,b,+,1],[]),true)
fact(statement([while,a,<,b,do,a,:=,a,*,2],[]),true)
fact(statement([if,a,=,b,then,a,:=,a,+,1],[]),true)
fact(statementlist([a,:=,a,+,1,;,b,:=,b,-,1],[]),true)
fact(statement([begin,if,a,<,b,then,a,:=,2,*,a,;,b,:=,b,+,2,end],[]),true)
fact(factor([a,*,b],[]),false)
fact(statement([1,:=,2],[]),false)
```

And the final grammar it came up with was:

```
(statement(X,Y)←ident(X,[:=|U]),expression(U,Y)).
(statement([while|X],Y)←condition(X,[do|U]),statement(U,Y)).
(statement([if|X],Y)←condition(X,[then|U]),statement(U,Y)).
(statement([begin|X],Y)←statementlist(X,[end|Y])).
(statementlist(X,Y)←statement(X,[;|U]),statement(U,Y)).
(condition(X,Y)←expression(X,U),comparator(U,W),expression(W,Y)).
(expression(X,Y)←term(X,U),plusminus(U,W),term(W,Y)).
(expression(X,Y)←term(X,Y)).
(term(X,Y)←factor(X,Y)).
(term(X,Y)←factor(X,U),timesdiv(U,W),term(W,Y)).
(factor(X,Y)←ident(X,Y)).
(factor(X,Y)←number(X,Y)).
(factor([(|X],Y)←expression(X,[)|Y])).
```

Which after macro deexpansion becomes:

```
statement ⇒ ident, [:=], expression.
statement ⇒ [while], condition, [do], statement.
statement ⇒ [if], condition, [then], statement.
statement ⇒ [begin] statementlist [end].
statementlist ⇒ statement [;], statement.
condition ⇒ expression, comparator, expression.
expression ⇒ term, plusminus, term.
expression ⇒ term.
term ⇒ factor.
term ⇒ factor, timesdiv, term.
factor ⇒ ident.
```

*factor* ⇒ *number*.
*factor* ⇒ [(], *expression*, [)].

The whole session took approximately 10 CPU minutes, and we supplied 35 facts. To see the utility of the pruning strategy, note that the size of the unpruned refinement graph up to depth 5 is 196,325,668. Although some of the grammar rules found in this session are in that depth, the maximal number of clauses searched for each goal was 68.

# Appendix II. Listings

This appendix contains listings of the systems described in the thesis. Most of the code has already appeared in the previous chapters. Instead of commnets and documentation, we provide pointers to the parts of the thesis that described the theory behind each piece of code. In some sense the thesis is one long comment to the code in this appendix.

The systems described in the thesis reside in the following files:

| | |
|---|---|
| PDSDC. | the diagnosis algorithms, described in Chapter 3. |
| PDS5. | the diagnosis system, described in Section 3.5. |
| MIS. | The Model Inference System, described in Section 4.3 |
| PDSREF. | The general refinement operator, used in most of the examples. Not described. |
| DCGREF. | The refinement operator for definite clause grammars, described in Section 4.5, and used in the example of inferring the subset of Pascal, in Appendix I. |
| MISRG. | The pruning breadth-first search algorithm, described in Section 4.5 |
| MISSRC. | The test-for-cover strategies, described in Section 4.4 |
| PDS6. | The interactive debugging system, described in Section 5.3. |
| PDSRG. | A derefinement procedure, used in the interactive debugging system, described in Section 5.2. |
| PDSDB. | Data base module for all systems, and implementation of the query procedure. |
| DSUTIL. | Utility procedures. These are all the utility procedures used in the systems described in the thesis. |
| PDSINI. | Some initialization declarations. |
| TYPE. | Type inference and checking procedures, used by the general refinement operator in PDSREF, not explained. |

XREF.DEF A file containing the definition of the predicate *system(_)*, that, for some reason, is not part of the standard list of evaluable predicates (stolen from Prolog's cross reference programm, not included here).

The files above contain all the code needed to run the systems described in the thesis on the Prolog-10. Each file that contains the top level procedure of the system also contains the list of files it requires to run.

Some statistics on the size of the systems is provided in Figure 7. A number that does not appear there is the total number of clause in the systems, which is 319.

| | | Lines | Print pages | Words |
|---|---|---|---|---|
| PDSDC | : | 121 | 3 | 650 |
| PDS5 | : | 30 | 1 | 162 |
| MIS | : | 37 | 1 | 158 |
| PDSREF | : | 121 | 3 | 573 |
| DCGREF | : | 41 | 1 | 155 |
| MISRG | : | 45 | 1 | 267 |
| MISSRC | : | 36 | 1 | 173 |
| PDS6 | : | 87 | 2 | 401 |
| PDSRG | : | 24 | 1 | 255 |
| PDSDB | : | 168 | 3 | 758 |
| TYPE | : | 107 | 2 | 438 |
| DSUTIL | : | 296 | 5 | 1041 |
| | Total: | 1123 | 24 | 5031 |

**Figure 7:** System statistics

## II.1 The diagnosis programs

```
%%%%% PDSDC
/* the diagnosis component */
```

```
% A depth-bounded interpreter
% see Program 9, page 61.
solve(P,X) ←
        solve(P,25,X), ( X\==true, ! ; true ).

solve(true,D,true) ← !.
solve(A,0,(overflow,[])) ← !.
solve((A,B),D,S) ←  !,
        solve(A,D,Sa),
        ( Sa=true → solve(B,D,Sb), S=Sb ; S=Sa ).
solve(A,D,Sa) ←
        system(A) → A, Sa=true ;
        D1 is D-1,
        clause(A,B), solve(B,D1,Sb),
        ( Sb=true → Sa=true ;
          Sb=(overflow,S) → Sa=(overflow,[(A←B)|S]) ).

% Tracing an incorrect procedure by divide-and-query
% see Program 6, page 47
false_solution(A) ←
        writel(['Error: wrong solution ',A,'. diagnosing...']), nl,
        fpm((A,W),_,0), % just to find W, the length of the computation
        fp(A,W,X) → handle_error('false clause',X) ;
        write('!Illegal call to fp'), nl.

fp(A,Wa,X) ←
        fpm((A,Wa),((P←Q),Wm),Wa),
        ( Wa=1 → X=(P←Q) ;
          query(forall,P,true) → Wa1 is Wa-Wm, fp(A,Wa1,X) ;
          fp(P,Wm,X) ).

% An interpreter that computes the middle point of a computation)
% see Program 5, page 46

% fpm((A,Wa),(M,Wm),W) ← solve A, whose weight is Wa.  find
% a goal M in the computation whose weight, Wm, is less then W/2,
% and is the heaviest son of a node whose weight exceeds (W+1)/2.
fpm(((A,B),Wab),M,W) ← !,
        fpm((A,Wa),(Ma,Wma),W), fpm((B,Wb),(Mb,Wmb),W),
        Wab is Wa+Wb,
        ( Wma>=Wmb → M=(Ma,Wma) ; M=(Mb,Wmb) ).
fpm((A,0),(true,0),W) ←
        system(A), !, A ;
        fact(A,true).
```

```
fpm((A,Wa),M,W) ←
        clause(A,B), fpm((B,Wb),Mb,W),
        Wa is Wb+1,
        ( Wa>(W+1)/2 → M=Mb ; M=((A←B),Wa) ).

% Tracing an incomplete procedure (improved)
% see Program 8, page 55
missing_solution(A) ←
        writel(['Error: missing solution ',A,'. diagnosing...']), nl,
        query(exists,A,true), \+solve(A,true) →
                ip(A,X), handle_error('uncovered atom',X);
        write('!Illegal call to ip'), nl.

ip(A,X) ←
        clause(A,B), ip1(B,X) → true ; X=A.
ip1((A,B),X) ← !,
        ( query(exists,A,true), ( A, ip1(B,X) ; \+A, ip(A,X) ) ).
        % cannot use → because need to check all solutions
        % in case of a nondeterministic procedure.
ip1(A,X) ←
        query(exists,A,true), ( A → break(ip1(A,X)) ; ip(A,X) ).


% Tracing a stack overflow
% see Program 10, page 62
stack_overflow(P,S) ←
        writel(['Error: stack overflow on ',P,'. diagnosing...']), nl,
        ( find_loop(S,S1) → check_segment(S1) ;
          check_segment(S) ).

find_loop([(P←Q)|S],Sloop) ←
        looping_segment((P←Q),S,S1) → Sloop=[(P←Q)|S1] ;
        find_loop(S,Sloop).

looping_segment((P←Q),[(P1←Q1)|S],[(P1←Q1)|Sl]) ←
        same_goal(P,P1) → writel([P,' is looping.']), nl, Sl=[] ;
        looping_segment((P←Q),S,Sl).

check_segment([(P←Q),(P1←Q1)|S]) ←
        query(legal_call,(P,P1),true) →
            check_segment([(P1←Q1)|S]) ;
        false_subgoal(P,Q,P1,C) → false_solution(C) ;
        handle_error('diverging clause',(P←Q)).
```

```
false_subgoal(P,(Q1,Q2),P1,Q) ←
        % search for a subgoal Q of P to the left of P1 that returned a false
        % solution.
        Q1\==P1,
        ( query(forall,Q1,false) → Q=Q1 ; false_subgoal(P,Q2,P1,Q) ).


% An interpreter that monitors errors
% see Program 12, page 75
msolve(P,X) ←
        msolve(P,25,X), ( X\==true, ! ; true ).

msolve(A,0,(overflow,[])) ← !.
msolve((A,B),D,S) ← !,
        msolve(A,D,Sa),
        ( Sa=true → msolve(B,D,Sb), S=Sb ; S=Sa ).
msolve(A,D,Sa) ←
        system(A) → A, Sa=true ;
        D1 is D−1,
        setof0((A,B,Sb), (clause(A,B), msolve(B,D1,Sb)),R),
        result(R,A,Sa).

result(R,A,(overflow,[(A←B)|St])) ←
        member((A,B,(overflow,St)),R), !.
result(R,A,false) ←
        member((A,_,false),R), ! ;
        member((A,B,true),R), fact(A,false) ,!,
            false_solution(A) ;
        fact(A,true), \+(member((A,_,true),R)), !,
            missing_solution(A).
result([],A,false) ←
        attribute(A,total), !,
            writel(['Error trapped: no solution to ',A]), nl,
            missing_solution(A).
result([A1,A2|R],A,false) ←
        attribute(A,determinate), !,
            writel(['Error trapped: too many solutions to ',A]), nl,
            member((A,_,_),[A1,A2|R]), query(forall,A,false), !,
                false_solution(A).
result(R,A,true) ←
        member((A,_,true),R).
```

## II.2 The diagnosis system

```
%%%%% PDS5.

← initialized → true ;
   % these files are required to run pds.
   ['xref.def'], [pdsdc,pdsdb,pdsini], compile([dsutil,type]),
   assert(initialized).

% A diagnosis system
% see Program 11, page 64
pds ←
       nl, read('',P), ( P=exit ; solve_and_check(P), pds ).

solve_and_check(P) ←
       bagof0((P,X),solve(P,X),S), confirm_solutions(P,S).

confirm_solutions(P,[(P1,X)|S]) ←
       member((P1,(overflow,X)),S) → stack_overflow(P1,X) ;
       writel(['solution: ',P1, ';']),
       ( ( system(P1) ; fact(P1,true) ) → nl, confirm_solutions(P,S) ;
         confirm(' ok') → assert_fact(P1,true), confirm_solutions(P,S) ;
         assert_fact(P1,false), false_solution(P1) ).
confirm_solutions(P,[]) ←
       write('no (more) solutions.'),
       ( system(P) → nl ;
         confirm(' ok') → true ;
         ask_for_solution(P), assert_fact(P,true), missing_solution(P) ).

handle_error('false clause',X) ← !,
       writel(['error diagnosed: ',X,' is false.']), nl, plisting(X).
handle_error('uncovered atom',X) ← !,
       writel(['error diagnosed: ',X,' is uncovered.']), nl, plisting(X).
handle_error('diverging clause',X) ← !,
       writel(['error diagnosed: ',X,' is diverging.']), nl, plisting(X).
```

## II.3 The Model Inference System

```
%%%%% MIS.
← initialized → true ;
   % These files are required to run the Model Inference System
```

```
   ['xref.def'], compile([misrg,dsutil,pdsref,type]),
   [pdsini,pdsdc,pdsdb,missrc],
   assert(initialized).

% The Model Inferece System
% see Program 13, page 95.
mis ←  nl, ask_for('Next fact',Fact),
      ( Fact=check → check_fact(_) ;
        Fact=(P,V), (V=true ; V=false)  → assert_fact(P,V), check_fact(P) ;
        write('!Illegal input'), nl ),
      !, mis.

check_fact(P) ←
      write('Checking fact(s)...'), ttyflush,
      ( fact(P,true), \+solve(P) →
          nl, missing_solution(P), check_fact(_) ;
        fact(P,false), solve(P) →
          nl, false_solution(P), check_fact(_) ;
        write('no error found.'), nl ).

solve(P) ←
      solve(P,X),
      ( X=(overflow,S) → nl, stack_overflow(P,S), solve(P) ; true ).

handle_error('false clause',X) ←
      writel(['Error diagnosed: ',X,' is false.']), nl,
      retract(X), plisting(X).
handle_error('uncovered atom',X) ←
      writel(['Error diagnosed: ',X,' is uncovered.']), nl,
      search_for_cover(X,C),
      assert(C),  plisting(X).
handle_error('diverging clause',X) ←
      writel(['Error diagnosed: ',X,' is diverging.']), nl,
      retract(X), plisting(X).

← assert(value(search_strategy,adaptive)).
```

## II.4 A general refinement operator

```
%%%%% PDSREF.
% General refinement operator for pds
← public
```

```
        refinement/2,
        create_io/2,
        derefine/2.

refinement(((P←Q),(Vi,Vo,Vf)),((P←Q),(Vi,[],[]))) ←
        % Close a clause
        Vo\==[],
        unisubset(Vo,Vi),
        eqdifset(Vf,Vo,[]),
        noduplicate_atom(P,Q).

refinement(((P←true),(Vi,Vo,[])),((P←true),(Vi2,Vo,[]))) ←
        % instantiate head, inputs.
        dmember(Var,Vi,Vi1),
        term_to_vars(Var,NewVars),
        append(Vi1,NewVars,Vi2).

refinement(((P←true),(Vi,Vo,[])),((P←true),(Vi,Vo2,[]))) ←
        % instantiate head, outputs.
        dmember(Var,Vo,Vo1),
        term_to_vars(Var,NewVars),
        append(Vo1,NewVars,Vo2).

refinement(((P←true),(Vi,Vo,[])),((P←true),(Vi1,Vo,[]))) ←
        % unify two input vars
        dmember(Var1,Vi,Vi1),
        member(Var2,Vi1),
        Var1 @< Var2,  % not to create duplicates
        Var1=Var2.
refinement(((P←Q1),(Vi,Vo,Vf)),((P←Q2),(Vi1,Vo,Vf1))) ←
        % add output producing goal
        Vo\==[],
        body_goal(P,Q,QVi,QVo),
        QVo\==[],
        unisubset(QVi,Vi),
        noduplicate_atom(Q,(P,Q1)),
        free_vars(Vf,QVi,QVo,Vf1),
        append(Vi,QVo,Vi1),
        qconc(Q,Q1,Q2).

refinement(((P←Q1),(Vi,[],[])),((P←Q2),(Vi,[],[]))) ←
        % add test predicate
        body_goal(P,Q,QVi,[]),
        unisubset(QVi,Vi),
```

```
        noduplicate_atom(Q,(P,Q1)),
        qconc(Q,Q1,Q2).

body_goal(P,Q,QVi,QVo) ←
        called(P,Q),
        input_vars(Q,QVi), output_vars(Q,QVo).


qconc(A,true,A) ← !.
qconc(A,(B,X),(B,Y)) ← !, qconc(A,X,Y).
qconc(A,B,(B,A)).

% unisubset(V1,V2) ← V1 is a subset of V2
unisubset([],_) ← !.
unisubset([X|V1],V2) ←
        dmember(X,V2,V3), unisubset(V1,V3).


% dmember(X,L1,L2) ← the difference between list L1 and list L2 is X.
dmember(X,[X|L],L).
dmember(X,[Y|L1],[Y|L2]) ←
        dmember(X,L1,L2).

% check no goals with duplicate inputs
noduplicate_atom(P1,(P2,Q)) ← !,
        ( same_goal(P1,P2), !,  fail ;  noduplicate_atom(P1,Q) ).
noduplicate_atom(P1,P2) ←
        same_goal(P1,P2), !, fail ; true.

% eqdifset(V1,V2,V3) ←  variable set V1 – V2 is V3.
eqdifset(V,[],V) ← !.
eqdifset(V1,[X|V2],V3) ←
        eqdelmember(X,V1,V4), !,
            eqdifset(V4,V2,V3) ;
            writel(['type conflict in ',eqdifset(V1,[X|V2],V3)]), break.


% eqdelmember(X,L1,L2) ← the difference between list L1 and list L2 is X.
eqdelmember(X1,[],[]) ← !.
eqdelmember(X1,[X2|L],L) ← X1==X2, !.
eqdelmember(X,[Y|L1],[Y|L2]) ←
        eqdelmember(X,L1,L2).
```

```
% create the input set Xi and output set Xo and free set Xf of variables
% of a clause P←Q.  does also typecheking.

create_io((P←Q),(Xi,Xo,Xf)) ←
        atom_vartype(P,Vi,Vo),
        create_io1(Vi,Xi,Vo,Xo,[],Xf,Q).

% create_io1(Vi,Xi,Vo,Xo,Vf,Xf,Q) ← if Vi, Vo and Vf are given input
% variable set, output variable set and free variable set, then together
% with Q, Xi, Xo and Xf are the input, output and free variable sets.

create_io1(Xi,Xi,Xo,Yo,Xf,Yf,true) ← !,
        eqdifset(Xo,Xi,Yo),
        eqdifset(Xo,Yo,Xf1),
        eqdifset(Xf,Xf1,Yf).
create_io1(Xi,Yi,Xo,Yo,Xf,Yf,(P,Q)) ← !,
        atom_vartype(P,Vi,Vo),
        eqdifset(Vi,Xi,Vdif),
        ( Vdif=[], !,
            append(Xi,Vo,Xi1),
            free_vars(Xf,Vi,Vo,Xf1),
            !, create_io1(Xi1,Yi,Xo,Yo,Xf1,Yf,Q) ;
            writel(['uninstantiated input variables ',Vdif,' in atom ',P]),
            fail ).
create_io1(Xi,Yi,Xo,Yo,Xf,Yf,P) ←
        create_io1(Xi,Yi,Xo,Yo,Xf,Yf,(P,true)).

% free_vars(Vf,Vi,Vo,Vf1) ← remove from Vf Vi, and add Vo, getting Vf1.
free_vars(Vf,Vi,Vo,Vf2) ←
        eqdifset(Vf,Vi,Vf1),
        append(Vf1,Vo,Vf2).
```

## II.5 A refinement operator for definite clause grammars

```
%%%%% DCGREF
% Refinement operator for definite clause grammars.
% See definition of ρ2, Page 118.
← public
        refinement/2,
        create_io/2,
        ntlisting/0,
        clearnt/0.
```

```
refinement(((P←Q1),(Vi,Vi,Vo)),((P←Q2),(Vi1,Vi1,Vo))) ←
        % add goal
        Vi\==[],
        nonterminal(Q),
        Q=..[F,Qi,Qo],
        Vi=[Qi], Vi1=[Qo],
        \+(( P=..[F,Pi,_], Pi==Qi )),
        qconc(Q,Q1,Q2).

refinement(((P←Q),(Vi,Vi,Vo)),((P←Q),([],[],[]))) ←
        % Close a clause
        Vi\==[], Vi=Vo.

refinement(((P←Q),(Vi,Vi,Vo)),((P←Q),(Vi1,Vi1,Vo))) ←
        % instantiate.
        Vi=[[X|Xs]],
        terminal(X),
        Vi1=[Xs].

qconc(A,true,A) ← !.
qconc(A,(B,X),(B,Y)) ← !, qconc(A,X,Y).
qconc(A,B,(B,A)).

% create the input set Xi and output set Xo and free set Xf of variables
% of a clause P←Q.  does also typecheking.
create_io((P←true),([Xi],[Xi],[Xo])) ←
        P=..[_,Xi,Xo].

ntlisting ←
        nonterminal(X), \+system(X), plisting(X), fail ; true.
clearnt ←
        nonterminal(X), \+system(X), X=..[F|_], abolish(F,2), fail ; true.
```

## II.6 Search strategies

```
%%%%% MISSRC.
%% An implementation of the search strategies
covers(C,P) ←
        ( value(search_strategy,S), member(S,[eager,lazy,adaptive]) → true ;
          break('Incorrect or missing search strategy') ),
        covers(S,C,P).
```

```
% The eager covers test)
% see Program 14, page 105
covers(eager,((P←Q),(Vi,Vf,Vo)),P1) ←
        ( Q=true ; Vo=[] ) → verify(( P=P1, satisfiable(Q) )) ;
        verify(P=P1).
% The lazy covers test)
% see Program 15, page 110
covers(lazy,((P←Q),_),P1) ←
        verify(( P=P1, fact_satisfiable(Q) )).
% The adaptive covers test)
% see Program 16, page 113
covers(adaptive,((P←Q),_),P1) ←
        verify(( P=P1, fact_solve(Q) )).

fact_satisfiable((P,Q)) ← !,
        fact_satisfiable(P), fact_satisfiable(Q).
fact_satisfiable(P) ←
        system(P) → P ; fact(P,true).

fact_solve(P) ←
        fact_solve(P,25,X),
        ( X=(overflow,S) → stack_overflow(P,S), fact_solve(P) ; true ).

fact_solve(A,0,(overflow,[])) ← !.
fact_solve((A,B),D,S) ← !,
        fact_solve(A,D,Sa),
        ( Sa=true → fact_solve(B,D,Sb), S=Sb ; S=Sa ).
fact_solve(A,D,Sa) ←
        system(A) → A, Sa=true ;
        fact(A,true) → Sa=true ;
        D1 is D-1,
        clause(A,B), fact_solve(B,D1,Sb),
        ( Sb=true → Sa=true ;
          Sb=(overflow,S) → Sa=(overflow,[(A←B)|S]) ).
```

## II.7 Pruning search of the refinement graph

```
%%%%% MISRG.
← public search_for_cover/2,
        check_refinements/6,
        good_clause/3,
```

```
        looping/1,
        refuted/1.

% A pruning breadth-first search of the refinement graph.
% see Program 17, page 124.
search_for_cover(P,Clause) ←
        nl, writel(['Searching for a cover to ',P,'...']), nl,
        mgt(P,P1), create_io((P1←true),Vs),
        search_for_cover([((P1←true),Vs)|Xs],Xs,P,Clause,1).

% search_for_cover(Head,Tail,Goal,Clause,Length) ←
        % The list between Head and Tail is the current queue of clauses.
        % search in it for a true Clause that covers Goal
        % Whenever you take a clause from the Head of the queue, add
        % all its refinements that cover Goal to Tail, setting it to
        % the new Tail of the queue. Length is the number of clauses
        % searched so far.

search_for_cover(Qhead,Qtail,P,C,Qlength) ←
        Qhead==Qtail,
        writel(['Failed to find a cover for ',P,'. queue is empty']), nl,
        !, fail.
search_for_cover([X|Qhead],Qtail,P,Clause,Qlength) ←
        X=(Xclause,_), writel(['Refining: ',Xclause]), nl,
        bagof0(Y,X^( refinement(X,Y), covers(Y,P) ),Qnew),
        length(Qnew,Qnewlength),
        Qlength1 is Qlength + Qnewlength,
        % writel(['New refinements:'|Qnew],v,nl), nl,
        check_refinements(Qnew,Qhead,Qtail,P,Clause,Qlength1).

check_refinements(Qnew,Qhead,Qtail,P,Clause,Qlength) ←
        member((Clause,Cv),Qnew), good_clause((Clause,Cv),Qlength).
check_refinements(Qnew,Qhead,Qtail,P,Clause,Qlength) ←
        append(Qnew,Qnewtail,Qtail),
        search_for_cover(Qhead,Qnewtail,P,Clause,Qlength).

good_clause((X,(Xi,[],[])),L) ←
        writel(['Checking: ',X]), nl,
        ( refuted(X), !, writel(['Refuted: ',X]), nl, fail ;
          looping(X), !, writel(['Looping: ',X]), nl, fail ;
          writel(['Found clause: ',X]), nl,
          writel(['     after searching ',L,' clauses.']), nl ).

looping((P←Q)) ←
```

```
        \+legal_calls(P,Q).
refuted((P←Q)) ←
        fact(P,false), fact_satisfiable(Q).
```

## II.8 The interactive debugging system

```
%%%%% PDS6.
← initialized → true ;
   % files required to run the interactive debugger
   ['xref.def'], compile([dsutil,type,misrg,pdsref]), [pdsdc,pdsdb,pdsini],
   [missrc,pdsrg,pdsref],
   assert(initialized).

% An interactive debugging system.
% See Section 5.3
pds ←
        nl, read('',P), ( P=exit ; solve_and_check(P), pds ).

solve_and_check(P) ←
        writel(['Solving ',P,'...']), nl,
        bagof0((P,X),msolve(P,X),S), confirm_solutions(P,S).

confirm_solutions(P,[(P1,(overflow,S))]) ← !,
        stack_overflow(P1,S),
        solve_and_check(P).
confirm_solutions(P,[(P1,false)]) ← !,
        solve_and_check(P).
confirm_solutions(P,[(P1,X)|S]) ←
        writel(['solution: ',P1, ';']),
        ( ( system(P1) ; fact(P1,true) ) → nl, confirm_solutions(P,S) ;
          confirm(' ok') → assert_fact(P1,true), confirm_solutions(P,S) ;
          assert_fact(P1,false),
          false_solution(P1), solve_and_check(P) ).
confirm_solutions(P,[]) ←
        write('no (more) solutions.'),
        ( system(P) → nl ;
          confirm(' ok') → true ;
          missing_solution(P), solve_and_check(P) ).

handle_error('false clause',X) ← !,
        writel(['Error diagnosed: ',X,' is false.']), nl,
        X=(P←Q),
```

```
        ask_then_do(
           ['retract (y), (m)odify, or (r)eplace it'],
           [(false, true),
            (true, retract(X)),
            (r, ( ask_for(['with what'],C),
                  retract(X), assert(C) ) ),
            (m, (mgt(P,P1), clause(P1,Q1,_), verify(((P←Q)=(P1←Q1))),
                 % can't use Ref because of a Prolog bug.
                 modify((P1←Q1),Y), retract(X), assert(Y) ) )
           ]),
        plisting(P), !.
handle_error('uncovered atom',P) ← !,
        writel(['Error diagnosed: ',P,' is uncovered.']), nl,
        ask_then_do(
        ['add (y) or (m)odify a clause'],
        [(false, true),
         (true, ( ask_for('which',C), assert(C) ) ),
         (m, ( ask_for('which',C1),
               ( C1=(_←_), !, retract(C1), C=C1 ;
                 C1=any, !, mgt(P,P1), C=(P1←true) ;
                 C=(C1←true), retract(C1) ),
               modify(C,P,Y), assert(Y) ) )
        ]),
        plisting(P), !.

handle_error('diverging clause',(P←Q)) ← !,
        writel(['Error diagnosed: ',(P←Q),' is diverging.']), nl,
        X=(P←Q),
        ask_then_do(
           ['retract (y), (m)odify, or (r)eplace it'],
           [(false, true),
            (true, retract(X)),
            (r, ( ask_for(['with what'],C),
                  retract(X), assert(C) ) ),
            (m, (mgt(P,P1), clause(P1,Q1,_), verify(((P←Q)=(P1←Q1))),
                 % can't use Ref because of a Prolog bug.
                 modify((P1←Q1),Y), retract(X), assert(Y) ) )
           ]),
        plisting(P), !.

modify(X,Y) ←
        reason(P,X), modify(X,P,Y).

modify(X,P,Y) ←
```

```
        search_rg(X,P,Y), confirm(ok), ! ; break(modify(X,P,Y)).

reason(P,X) ←
        reason1(P,X) → true ;
        ask_for(['What is a reason for ',X],P) →
           assert(reason1(X,P)).

← assert(value(search_strategy,eager)).
```

## II.9 The bug-correction program

```
%%%%% PDSRG.
% A bug-correction algorithm.
% See Algorithm 8, Page 139.
search_rg(X,P,Y) ← % search for Y that covers P, starting from X.
        create_io(X,Vx), !,
        search_rg1((X,Vx),P,Y).

search_rg1(X,P,Y) ←
        covers(X,P), X=(Xc,_), \+looping(Xc) →
           check_refinements([X],Xs,Xs,P,Y,1) ;
        derefine(X,X1,P), search_rg1(X1,P,Y).

% derefine(X,Y) ←  Y is the result of derefining X. which means,
% in the meantime, omitting the last condition from X.
derefine((X,Vx),Y,_) ←
        writel(['Derefining ',X,'...']), nl, derefine1((X,Vx),Y).
derefine1(((X←Xs),Vx),Y) ←
        deconc(Xs,Ys), create_io((X←Ys),Vy), new(((X←Ys),Vy),Y).
derefine1(((X←true),Vx),((Y←true),Vy)) ←
        mgt(X,Y), \+variants(X,Y), create_io((Y←true),Vy).

% delete the last conjunct
deconc((X1,(X2,Xs)),(X1,Ys)) ← !, deconc((X2,Xs),Ys).
deconc((X1,X2),X1) ← !.
deconc(X,true) ← X\==true.
```

## II.10 Database interface utilities

```
%%%%% PDSDB
% Data base for pds.

% The base relation is solutions(P,S), which denotes that
% the solutions of goal P are exactly S. This relation stores
% results of existential queries.
% On top of it, we compute the relation fact(P,V), which says P is known
% to have truth value V, were 'known' is defined in the broadest way possible.
% i.e., can contains any clauses that represent our current knowledge.
% Using the 'fact' relation, we encode constraints, etc.

fact(P,V) ←
        var(P) → ( solutions(_,S), member(P,S), V=true ;
                      solutions(P,[]), V=false ) ;
        solutions(P,S), ( member(P,S), V=true ; \+member(P,S), V=false ).

listfact(P) ←
        fact(P,V), write(fact(P,V)), nl, fail ; true.

is_instance(P1,P2) ←
        % P1 is an instance of P2
        verify(( numbervars(P1,0,_), P1=P2 )).

assert_fact(P,V) ←
        fact(P,V1) → ( V=V1, !, true ; break(assert_fact(P,V)) ) ;
        \+ground(P) → break(assert_fact(P,V)) ;
        % writel(['Asserting: ',fact(P,V)]), nl,
        ( V=true → assert(solutions(P,[P])) ;
          V=false → assert(solutions(P,[])) ;
          break(assert_fact(P,V)) ).


query(exists,P,V) ←
        system(P) → ( P → V=true ; V=false ) ;
        mgt(P,P1), solutions(P1,S), is_instance(P,P1) →
           ( member(P,S), V=true ; \+member(P,S), V=false ) ;
        fact(P,true), V=true ;
        ask_for_solutions(P,S) →
           ( S=[] → V=false ; member(P,S), V=true ).
query(forall,P,V) ←
        ground(P) → query(exists,P,V) ;
        break(query(forall,P,V)).
```

```
query(solvable,P,V) ←
        system(P) → ( P → V=true ; V=false ) ;
        fact(P,V1) → V=V1 ;
        ask_for(['Query: ',P],V1,(V1=true;V1=false)) → V=V1.

ask_for_solutions(P,S) ←
          bagof0(P,ask_for_solution(P),S),
          % writel(['Asserting: ', solutions(P,S)]), nl,
          assert(solutions(P,S)).

ask_for_solution(P) ←
        nl, ask_for(['Query: ',P],V,(V=true;V=false)),
        ( V=false → fail ;
          ground(P) → true ;
          varand(P,Pvars),
          repeat,
            writel(['Which ',Pvars,'? ']), ttyflush,
            reade(Answer),
            ( Answer=false, !, fail ;
              Answer=Pvars → true ;
              write('does not unify; try again'), nl ),
            ( attribute(P,determinate), ! ; true ) ).


query(legal_call,(P1,P2),V) ←
        same_goal(P1,P2), !, V=false ;
        legal_call((Q1,Q2),V1), same_goal(P1,Q1), same_goal(P2,Q2), !,
            V=V1 ;
        confirm(['Is ',(P1,P2),' a legal call']), !,
            assert(legal_call((P1,P2),true)), V=true ;
        assert(legal_call((P1,P2),false)), V=false.

known_illegal_call(P1,P2) ←
        same_goal(P1,P2), !, V=false ;
        legal_call((Q1,Q2),false), same_goal(P1,Q1), same_goal(P2,Q2).


same_goal(P,Q) ←
        functor(P,F,N), functor(Q,F,N),
        input_vars(P,Pi), input_vars(Q,Qi), !, variants(Pi,Qi).

satisfiable((P,Q)) ←!,
        query(exists,P,true), satisfiable(Q).
```

```
satisfiable(P) ←
        query(exists,P,true).


legal_calls(P,true) ←!.
legal_calls(P,Q) ←
        ( Q=(Q1,Q2), !, true ; Q=Q1, Q2=true ),
        ( known_illegal_call(P,Q1), !, fail ;  true ),
        ( fact(Q1,true), !, legal_calls(P,Q2) ; true ).
            % for all true solutions to Q1, Q2 shouldn't loop.

clear ←
        abolish(solutions,2),
        abolish(legal_call,2).

clear(P) ←
        ( retract(solutions(P,_)), fail ; true ).

edit_facts ←
        solutions(P,S),
        confirm(['Retract ',solutions(P,S)]),
        retract(solutions(P,S)),
        fail ; true.

/* Information about a procedure:

        ← declare(P,A), where
            P is, for example qs(+[x],-[x]), and
            A is, for example [determinate,total]

This will create the resulting data:
        declared(P,Inv,OutV,A), where InV (OutV) are pairs of input (output)
            variables and their types, and
            A is the list of attribute.

*/

declare(Pmode,Ps) ←
        mgt(Pmode,P),
        P=..[F|Pargs],
        Pmode=..[F|Fargs],
        varplusminus(Pargs,Fargs,InV,OutV),
        ( retract(declared1(P,_,_,_)), fail ; true ),
        % writel(['Declaring ',(P,InV,OutV,Ps)]), nl,
        assert(declared1(P,InV,OutV,Ps)).

varplusminus([V|Pargs],[+(T)|Fargs],[(V,T)|PlusV],MinusV) ← !,
        varplusminus(Pargs,Fargs,PlusV,MinusV).
varplusminus([V|Pargs],[-(T)|Fargs],PlusV,[(V,T)|MinusV]) ← !,
        varplusminus(Pargs,Fargs,PlusV,MinusV).
varplusminus([],[],[],[]) ← !.
varplusminus(Pargs,Fargs,PlusV,MinusV) ←
        break( varplusminus(Pargs,Fargs,PlusV,MinusV) ).

declared(P,Pi,Po,[]) ←
        nonterminal(P), P=..[_,Pi,Po].

declared(P,Pi,Po,Pa) ←
        declared1(P,Pi1,Po1,Pa1), !, Pi1=Pi, Po1=Po, Pa1=Pa ;
        ask_for(['Declare ',P],declare(Pv,Pa)), declare(Pv,Pa),
        declared(P,Pi,Po,Pa).

attribute(X,Xa) ←
        declared(X,_,_,Xas), !, member(Xa,Xas).

input_vars(P,InV) ←
        declared(P,InV,_,_).

output_vars(P,OutV) ←
        declared(P,_,OutV,_).

atominfo(P,_,_,_) ← break( atominfo(P,_,_,_) ).

declare_called(P,Ps) ←
        ( retract(called1(P,_)), fail ; true ),
        assert(called1(P,Ps)).

called(P,Q) ←
        system(P), !, fail ;
        called1(P,Qs), !, member(Q,Qs) ;
        ask_for(['Procedures called by ',P],Ps),
        and_to_list(Ps,Ps1),
        declare_called(P,Ps1).
```

## II.11 General utilities

```
    %%%%% DSUTIL.
    /* Utilities used in the debugging system */

    ← public
            member/2, append/3, reverse/2, rev/3, set/2, add1/2, ask_for/2,
            ask_for/3, confirm/1, writel/3, write/2, writel/1, read/2,
            reade/1, directive/1, writev/1, lettervars/1, unify_vars/2,
            break/1, varand/2, varlist/2, mgt/2, size/2, verify/1, ground/1,
            variants/2, list_to_and/2, and_to_list/2, and_member/2,
            forall/3, portray/1, portray1/1, bagof0/3, setof0/3, new/2,
            plisting/1, ask_then_do/2.

    ←mode(member(?,+)).
    member(X,[X|_]).
    member(X,[_|L]) ← member(X,L).

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

    reverse(X,Y) ← rev(X,[],Y).
    rev([X|Xs],Ys,Zs) ← rev(Xs,[X|Ys],Zs).
    rev([],Xs,Xs).


    set(P,V) ←
            retract(value(P,_)), !, set(P,V) ;
               assert(value(P,V)).

    add1(P,V1) ←
            retract(value(P,V)) , integer(V), !,
               V1 is V+1, assert(value(P,V1)) ;
               writel(['no value for ',P,', initializing it to 1']), nl,
               set(P,0), V1=1.

    ask_for(Request,Answer,Test) ←
            repeat,
               ask_for(Request,Answer), Test, !.

    ask_for(Request,Answer) ←
            repeat,
               writel(Request), write('? '), ttyflush,
               reade(X),
        ( directive(X) , !,
          ( X, ! ; write('?'), nl ),
          ask_for(Request,Answer) ;
          Answer=X ), !.

confirm(P) ←
        ask_for(P,V),
        ( V=true , !, true ;
          V=false, !, fail ;
          confirm(P) ).

% writel(L,E,S) ← write list L, with list elements format E and
% seperator S.

writel(L,E,S) ←
        var(L), !, write(E,L) ;
        L=[], !, true ;
        L=[X], !, write(E,X) ;
        L=[X|L1], !, writel(X,E,nil), write(s,S), writel(L1,E,S) ;
        write(E,L).

write(w,X) ← write(X).
write(v,X) ← writev(X).
write(s,S) ←
        S=nil, !, true ;
        S=nl, !, nl ;
        S=bl, !, write(' ') ;
        S=comma, !, write(', ') ;
        write(S).

writel(L) ←
        writel(L,v,nil).

read(P,X) ← prompt(P1,P), read(X), prompt(P,P1).

reade(X) ←
        read(X1),
        ( expand(X1,X), !, true ; X=X1 ).

expand(t,true). expand(yes,true). expand(y,true). expand(f,false).
expand(no,false). expand(n,false). expand(a,abort). expand(b,break).
expand(push,exe).

directive(abort).
```

```
directive(trace).
directive(break).
directive(info).
directive(X) ←
        X=true, !, fail ;
        X=( _ =< _ ), !, fail ;
        X=( _ < _ ), !, fail ;
        X=( _ > _ ), !, fail ;
        X=( _ >= _ ), !, fail ;
        system(X).


writev(X) ←
        lettervars(X), write(X), fail.
writev(X).

lettervars(X) ←
        varlist(X,V1),
        % sort(V1,V2),
        V1=V2,
        unify_vars(V2,
        ['X','Y','Z','U','V','W','X1','Y1','Z1','U1','V1','W1',
        'X2','Y2','Z2','U2','V2','W2','X3','Y3','Z3','U3','V3','W3',
        'X4','Y4','Z4','U4','V4','W4']), !.


unify_vars([X|L1],[X|L2]) ← !,
        unify_vars(L1,L2).
unify_vars([_|L1],[_|L2]) ← !,
        unify_vars(L1,L2).
unify_vars(_,_).

break(P) ← portray(P), nl, call(break).

← mode varlist(+,-).
% varlist(T,L,[]) ← L is all occurances of distinct variables in term T
varlist(X,L) ← varlist(X,L,[]), !.

← mode varlist(+,-,?).
varlist(X,[X|L],L) ← var(X),!.
varlist(T,L0,L) ← T =.. [F|A], !, varlist1(A,L0,L).

varlist1([T|A],L0,L) ← varlist(T,L0,L1), !, varlist1(A,L1,L).
varlist1([],L,L).
```

```
←mode mgt(+,−).
mgt(P,P0) ←
        functor(P,F,N),
        functor(P0,F,N).


verify(P) ← \+(\+(P)).

ground(P) ← numbervars(P,0,0).

variants(P,Q) ←
        verify(( numbervars(P,0,N), numbervars(Q,0,N), P=Q )).

varand(P,Vs1) ←
        varlist(P,Vs),
        list_to_and(Vs,Vs1).

list_to_and([],true) ← !.
list_to_and([X],X) ← !.
list_to_and([X|Xs],(X,Ys)) ← !,
        list_to_and(Xs,Ys).

and_to_list((X,Y),[X|Z]) ← !,
        and_to_list(Y,Z).
and_to_list(true,[]) ← !.
and_to_list(X,[X]) ← !.


and_member(P,(P,Q)).
and_member(P,(P1,Q)) ← !, and_member(P,Q).
and_member(P,P).

forall(X,P,Y) ←
        setof(Y,X↑P,S), forall1(S).

forall1([]).
forall1([X|S]) ← X, forall1(S).


portray(X) ←
        lettervars(X),
        portray1(X,6),
        fail.
```

```
portray(X).


portray1(X,N) ←
        N1 is N−1,
      ( var(X), !, write(X) ;
        atomic(X), !, write(X) ;
        N=0, !, write('#') ;
        X=[_|_], !, write('['), portray_list(X,N1,5), write(']') ;
        X=(_,_), !, write('('), portray_and(X,N1), write(')') ;
        X=..[F|A], !, portray_term(F,A,N1) ;
        break(portray1(X,N)) ).

portray_args(X,N) ←
        X=[], !, true ;
        X=[Y], !, portray1(Y,N) ;
        X=[Y|Ys], !, portray1(Y,N), write(','), !, portray_args(Ys,N).

portray_list(X,N,D) ←
        var(X), !, portray1(Y,N) ;
        X=[], !, true ;
        D=0, !, write('..#') ;
        X=[Y1|Y2], Y2==[], !, portray1(Y1,N) ;
        X=[Y1|Y2], var(Y2), !, portray1(Y1,N), write('|'), !, portray1(Y2,N) ;
        X=[Y1,Y2|Ys], !,
           portray1(Y1,N), write(','), D1 is D−1, !,
              portray_list([Y2|Ys],N,D1) ;
        X=[Y1|Y2], !, portray1(Y1,N), write('|'), !, portray1(Y2,N).

portray_and(X,N) ←
        var(X), !, portray1(X,N);
        X=(Y,Ys), !, portray1(Y,N), write(','), !, portray_and(Ys,N) ;
        portray1(X,N).

portray_term(F,[A],N) ←
            current_op(P,T,F), !,
            write(F), write(' '), portray1(A,N) .
portray_term(F,[A,B],N) ←
            current_op(P,T,F), !,
            portray1(A,N), write(F), portray1(B,N).
portray_term(F,A,N) ←
            write(F), write('('), portray_args(A,N), write(')').
```

```
bagof0(X,P,S) ←
        bagof(X,P,S), !, true ; S=[].

setof0(X,P,S) ←
        setof(X,P,S), !, true ; S=[].

new(X,Y) ←  % Y is a fresh copy of X (with new variables)
        abolish('gross hack',1),
        assert('gross hack'(X)),
        retract('gross hack'(Y)).


plisting([]) ← !.
plisting([P|Ps]) ← !,
        plisting(P), nl, !, plisting(Ps).
plisting(X) ←
        ( X=(P←_), !, mgt(P,P1) ; mgt(X,P1) ),
        writel(['Listing of ',P1,':']), nl,
        ( clause(P1,Q), tab(4), writev((P1←Q)), write('.'), nl, fail ;
          true ), nl.


ask_then_do(Question,Responses) ←
        % display question. A response is a list of (Answer,Action) pairs;
        % verify that the answer the user gives is in a pair;
        % if so, perform the action associated with it.
        ask_for(Question,Answer),
        member((Answer,Action),Responses) → Action ;
        setof(Answer, Action|member((Answer,Action),Responses),Answers),
        writel('legal answers are ',Answers), nl,
        ask_then_do(Question,Responses).
```

## II.12 Initialization

```
%%%%% PDSINI.
% initialization stuff.

← declare(qsort(+[x],-[x]),[determinate,total]).
← declare_called(qsort(X,Y),
        [qsort(Z,U),partition(V,W,X1,Y1),append(Z1,U1,V1),c(W1,X2,Y2)]).
← declare(partition(+[x],+x,-[x],-[x]),[determinate,total]).
← declare_called(partition(X,Y,Z,U),[partition(V,W,X1,Y1),Z1<U1,V1=<W1]).
```

```
← declare(append(+[x],+[x],-[x]),[determinate,total]).
← declare_called(append(X,Y,Z),[append(U,V,W)]).
← declare(le(+0,+0),[determinate]).
← declare_called(le(_,_),[le(Z,U)]).
← declare(insert(+x,+[x],-[x]),[determinate,total]).
← declare_called(insert(X,Y,Z),[insert(U,V,W),X2<Y2,X1=<Y1]).
← declare(isort(+[x],-[x]),[determinate,total]).
← declare_called(isort(X,Y),[isort(Z,U),insert(V,W,X1)]).
← declare(+x'=<'+x,[determinate]).
← declare(+x'<'+x,[determinate]).
```

## II.13 Type inference and checking

```
%%%%% TYPE
%%%%% Typing utilities for pds

← public
        term_to_vars/2,
        typed_term/2,
        atom_vartype/3,
        vartype/4,
        type_check/1.


←mode type(?,?,-,-).
% type(Type,Name,Terms,TermsType).

type(x,object,[],[]).
type(0,integer,[0,s(_)],[s(0)]).
type(1,integer,[0],[]).
type(10,boolean,
    [0,1,not(_),and(_,_),or(_,_)],
    [not(01),and(01,01),or(01,01)]).
type(io,binary,[nil,o(_),i(_)],[o(io),i(io)]).
type([],list,[[],[_|_]],[[a|[]]]).
type([X],'list of',[[],[_|_]],[[X|[X]]]).
type(bt(L),'binary tree',
    [leaf(_),t(_,_)],[leaf(L),t(bt(L),bt(L))]).
type(lbt(X),'labeled binary tree',
    [nil,t(_,_,_)],[t(lbt(X),X,lbt(X))]).
type(ttt(L),'two-three tree',
    [leaf(_),t(_,_,_)],[leaf(L),t(L,L,[ttt(L)])]).
```

```
type(terminal,terminal,[X],[]) ←
        terminal(X).

% input a variable and its type, instantiate it to a term and
% return a list of the variables in the term + their types.
term_to_vars((Term,TermType),Vars) ←
        term_of(TermType,Term),
        setof_vartype((Term,TermType),Vars).

term_of(Type,Term) ←
        type(Type,_,TermList,_),
        member(Term,TermList).

typed_term(Type,Term) ←
        type(Type,_,_,TypedTerms),
        member(Term,TypedTerms).

% atom_vartype(P,Vi,Vo) ← get type of vars in P.
atom_vartype(P,Vi,Vo) ←
        input_vars(P,Pi),
        output_vars(P,Po),
        terms_to_vartype(Pi,Vi),
        terms_to_vartype(Po,Vo).

% terms_to_vartype(T,V) ← take a list of (Term,Type) and return a list
%   of (Var,Type) for all vars in the terms
terms_to_vartype([],[]).
terms_to_vartype([(Term,Type)|T],Vs) ←
        setof_vartype((Term,Type),Vs1),
        terms_to_vartype(T,Vs2),
        append(Vs1,Vs2,Vs).

setof_vartype((Term,TermType),Vars) ←
        setof((Var,Type),vartype(Term,TermType,Var,Type),Vars), !.
setof_vartype((Term,TermType),[]).

% vartype(Term,TermType,Var,VarType) ←
%       The type of variable Var that occurs in
%       term Term of type TermType is VarType
% reports on type violation?
vartype(Var,Type,Var1,Type1) ←
        var(Var), !,
        Var=Var1,
        Type=Type1.
```

```
vartype(Term,TermType,Var,VarType) ←
        Term=..[Functor|Args],
        typed_term(TermType,Term1), Term1=..[Functor|ArgsType],
        vartype1(Args,ArgsType,Var,VarType).

←mode vartype1(+,+,-,?).
vartype1([Term|_],[TermType|_],Var,VarType) ←
        vartype(Term,TermType,Var,VarType).
vartype1([_|Args],[_|ArgsType],Var,VarType) ←
        vartype1(Args,ArgsType,Var,VarType).


←mode type_check(+).
type_check(←(P,Q)) ← !,
        type_check(P), type_check(Q).
type_check((P,Q)) ← !,
        type_check(P), type_check(Q).
type_check(Atom) ←
        type(Atom,_,_,[AtomType]),
        type_check(Atom,AtomType).

←mode type_check(+,+).
type_check(Term,TermType) ←
        term_of(TermType,Term),
        ( atomic(Term) ;
          Term=..[Functor|Args],
            typed_term(TermType,Term1),
            Term1=..[Functor|ArgsType],
            type_check1(Args,ArgsType) ).

←mode type_check1(+,+).
type_check1([],[]).
type_check1([Term|Args],[TermType|ArgsType]) ←
        type_check(Term,TermType),
        type_check1(Args,ArgsType).
```

## References

[1] A. H. Aho and J. D. Ullman.
*The Theory of Parsing, Translation and Compiling.*
Prentice-Hall, 1972.

[2] D. Angluin.
Finding patterns common to a set of strings.
*Journal of Computer and System Sciences* 21:46-62, 1980.

[3] D. Angluin.
Inference of reversible languages.
JACM, to appear.

[4] D. Angluin.
A note on the number of queries needed to identify regular languages.
Information and Control, to appear.

[5] D. Angluin.
Term sequence inference.
1982, in preparation.

[6] D. Angluin and C. H. Smith.
A brief survey of inductive inference.
1982, in preparation.

[7] 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.

[8] Robert Balzer.
*Automatic Programming.*
Technical Report 1, USC/ISI, September, 1972.

[9] A. W. Biermann.
The inference of regular Lisp programs from examples.
*IEEE Transactions on Systems, Man, and Cybernetics* SMC-8:585-600, August, 1978.

[10] A. W. Biermann.
On the inference of Turing machines from sample computations.
*Artificial Inelligence.* 3:181-198, 1972.

[11] A. W. Biermann, R. I. Baum, and F. E. Petry.
Speeding up the synthesis of programs from traces.
*IEEE Transactions on Computers* C-24:122-136, 1975.

[12] A. W. Biermann and R. Krishnaswamy.
Constructing programs from example computations.
*IEEE Transactions on Software Engineering* SE-2:141-153, 1976.

[13] Lenore Blum and Manuel Blum.
Towards a mathematical theory of inductive inference.
*Information and Control* 28, 1975pages"125-155".

[14] D. L. Bowen, L. Byrd, L. M. Pereira, F. C. N. Pereira and D. H. D. Warren.
*PROLOG on the DECSystem—10 User's Manual.*
Technical Report , Department of Artificial Intelligence, University of Edinburgh, October, 1981.

[15] P. Brazdil.
*A Model of Error Detection and Correction.*
PhD thesis, University of Edinburgh, 1981.

[16] Martin Brooks.
*Determining Correctness by Testing.*
Technical Report STAN-CS-80-804, Computer Science Department, Stanford University, May, 1980.

[17] Bruce C. Buchanan and Tom M. Mitchell.
Model-directed learning of production rules.
In D. A. Waterman & Frederick Hayes-Roth (editors), *Pattern Directed Inference Systems*, pages 297-312. Academic Press, 1978.

[18] Timothy A. Budd.
*Mutation Analysis of Program Test Data.*
PhD thesis, Yale University, 1980.

[19] T. Budd, R. DeMillo, R. Lipton and F. Sayward.
Theoretical and empirical studies on using program mutation to test the functional correctness of programs.
In *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages.* ACM, January, 1980.

[20] A. Bundy and B. Silver.
*A Critical Survey of Rule Learning Programs.*
Technical Report 169, Department of Artificial Intelligence, University of Edinburgh, January, 1982.

[21] R. A. DeMillo, R. J. Lipton, and A. J. Perlis.
Social processes and proofs of theorems and programs.
*Communications of the ACM* 22(5), 1979.

[22] Lawrence Byrd.
Prolog debugging facilities.
1980.
Technical note, Department of Artificial Intelligence, Edinburgh University.

[23] J. Case and C. Smith.
*Comparison of Identification Criteria for Mechanized Inductive Inference.*
Technical Report 154, Department of Computer Science, SUNY Buffalo, 1979.

[24] A. K. Chandra, D. C. Kozen, L. J. Stockmeyer.
Alternation.
*Journal of the ACM* 28(1):114-133, January, 1981.

[25] C. L. Chang and R. C. T. Lee.
*Symbolic Logic and Mechanical Theorem Proving.*
Academic Press, New York, 1973.

[26] Keith L. Clark.
Negation as failure.
In H. Gallaire and J. Minker (editors), *Logic and Data Bases*, . Plenum, 1978.

[27] A. Colmerauer.
Metamorphosis grammars.
In L. Bolc (editor), *Natural language communication with computers*, . Springer-Verlag, 1978.

[28] Alan Colmerauer.
Infinite trees and inequalities in Prolog.
In *Proceedings of the Prolog Programming Environments Workshop.* Datalogi, Linkoping, Sweden, March, 1982.
To appear.

[29] S. Crespi-Reghizzi.
An effective model for grammar inference.
In *Information Processing* 71, pages 524-529. North-Holland, Amsterdam, 1972.

[30] S. Crespi-Reghizzi, G. Guida and D. Mandrioli.
Noncounting context-free languages.
*Journal of the ACM* 25:571-580, 1978.

[31] 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.

[32] T. G. Dietterich and R. S. Michalski.
Learning and generalization of characteristic descriptions: evaluation criteria and comparative review of selected methods.
In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 223-231. IJCAI, 1979.

[33] M. H. van Emden and R. A. Kowalski.
The semantics of predicate logic as a programming language.
*Journal of the ACM* 23:733-742, October, 1976.

[34] R. W. Floyd.
Assigning meanings to programs.
In J. T. Schwartz (editor), *Proceedings of Symposium in Appliead Mathematics*, pages 19-32. American Mathematical Society, 1967.

[35] Susan L. Gerhart and Lawrence Yelowitz.
Observations of fallibility in applications of modern programming methodologies.
*IEEE Transactions on Software Engeneering* SE-2:195-207, September, 1976.

[36] E. Mark Gold.
Limiting recursion.
*Journal of Symbolic Logic* 30, 1965.

[37] E. M. Gold.
Language identification in the limit.
*Information and Control* 10:447-474, 1967.

[38] E. M. Gold.
Complexity of automaton identification from given data.
*Information and Control* 37:302-320, 1978.

[39] J. B. Goodenough and S. L. Gerhart.
Towards a theory of test data selection.
*IEEE Transactions on Software Engeneering* SE-1:156-173, June, 1975.

[40] C. Cordell Green.
Theorem proving by resolution as a basis for question answering.
In B. Meltzer and D. Michie (editors), *Machine Intelligence* 4, pages 183-205. Edinburgh University Press, Edinburgh, 1969.

[41] S. Hardy.
Synthesis of LISP programs from examples.
In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 268-273. IJCAI, 1975.

[42] F. Hayes-Roth, P. Klahr, and D. J. Mostow.
*Knowledge Acquisition, Knowledge Refinement, and Knowledge Programming.*
Technical Report R-2540-NSF, The Rand Corporation, may, 1980.

[43] Carl Hewitt.
*Description and Theoretical Analysis (Using Schemata) of Planner: a Language for Proving Theorems and Manipulating Models in a Robot.*
Technical Report TR-258, MIT Artificial Intelligence Lab, 1972.

[44] William E. Howden.
Algebraic program testing.
*Acta Informatica* 10(1):53-66, 1978.

[45] J. D. Ichbiah et al.
Preliminary Ada reference manual.
*ACM SIGPLAN Notices* 14(6), June, 1979.
Part A.

[46] Ingalls, Daniel H. H.
The smalltalk-76 programming system: design and implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 9-16. Association for Computing Machinery, January, 1978.

[47] Kenneth E. Iverson.
*A Programming Language.*
John Wiley and Sons, Inc., New York, 1962.

[48] Kathleen Jensen and Niklaus Wirth.
*Pascal User Manual and Report.*
Springer-Verlag, New York, 1974.

[49] Mark Scott Johnson.
A software debugging glossary.
*SIGPLAN notices* 17(2):53-70, February, 1982.

[50] J. P. Jouannaud and G. Guiho.
Inference of functions with an interactive system.
In J. E. Hayes, D. Michie, and L. I. Mikulich (editors), *Machine Intelligence* 9, pages 227-250. John Wiley and Sons, N. Y., 1979.

[51] J. P. Jouannaud and Y. Kodratoff.
An automatic construction of LISP programs by transformations of functions synthesized from their input-output behavior.
*International Journal of Policy Analysis and Information Systems* 4:331-358, 1980.

[52] J. P. Jouannaud and Y. Kodratoff.
Characterization of a class of functions synthesized by a Summers-like method using a B.M.W. matching technique.
In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 440-447. IJCAI, 1979.

[53] B. Knobe and K. Knobe.
A method for inferring context-free grammars.
*Information and Control* 31:129-146, 1976.

[54] Robert A. Kowalski.
Predicate logic as a programming language.
In *Information Processing* 74, pages 569-574. North-Holland, Amsterdam, 1974.

[55] P. Langley.
*Language Acquistion Through Error Recovery.*
Technical Report CIP-432, Carnegey-Mellon University, June, 1981.

[56] Zohar Manna and Richard Waldinger.
The logic of computer programming.
*IEEE Transactions on Software Engeneering* SE-4:199-229, May, 1978.

[57] Frank G. McCabe.
*Micro–PROLOG programmer's reference manual.*
Logic Programming Associates Ltd., London, 1981.

[58] Drew V. McDermott.
The Prolog phenomenon.
*SIGART Newsletter* 72, July, 1980.

[59] R. Michalski.
Pattern recognition as rule guided inductive inference.
*IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-2:349-361, 1980.

[60] Tom Michael Mitchell.
*Version Spaces: An Approach to Concept Learning.*
Technical Report STAN-CS-78-711, Stanford Artificial Intelligence Laboratory, December, 1978.

[61] James G. Mitchell, William Maybury, and Richard Sweet.
*Mesa language manual.*
Technical Report CSL-79-3, Xerox Palo Alto Research Center, April, 1979.

[62] T. Moto-oka et al.
Challenge for knowledge information processing systems (preliminary report on fifth generation computer systems).
In *Proceedings of Internation Conference on Fifth Generation Computer Systems*, pages 1-85. JIPDEC, 1981.

[63] P. Naur.
Programming by action clusters.
*BIT* 9:250-258, 1969.

[64] P. Naur & B. Randel, Eds.,
*Software Engineering.*
Nato Scientific Affairs Division, Brussels, Belgium, 1969.

[65] F. C. N. Pereira and D. H. D. Warren.
Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks.
*Artificial Intelligence* 13:231-278, 1980.

[66] Alan J. Perlis.
Controlling software development through the life cycle model.
In A. J. Perlis, F. G. Sayward, and M. Shaw (editors), *Software Metrics*, pages 95-110. The MIT Press, 1981.

[67] G. D. Plotkin.
A note on inductive generalization.
In B. Meltzer and D. Michie (editors), *Machine Intelligence* 5, pages 153-165. Edinburgh University Press, Edinburgh, 1970.

[68] Gordon D. Plotkin.
*Automatic Methods of Inductive Inference.* .
PhD thesis, Edinburgh University, August, 1971.

[69] G. D. Plotkin.
A further note on inductive generalization.
In B. Meltzer and D. Michie (editors), *Machine Intelligence* 6, pages 101-124. Edinburgh University Press, Edinburgh, 1971.

[70] Wolfgang H. Polak.
*Theory of Compiler Specification and Verification.*
Technical Report STAN-CS-80-802, Stanford Artificial Intelligence Laboratory, May, 1980.

[71] Karl R. Popper.
*The Logic of Scientific Discovery.*
Basic Books, New York, 1959.

[72] Karl R. Popper.
*Conjectures and Refutations: The Growth of Scientific Knowledge.*
Harper Torch Books, New York, 1968.

[73] Scott Renner.
A Comparison of PROLOG systems available at the University of Illinois.
April, 1982.
Internal report UIUCDCS-F-82-897, Knowledge Based Programmer's Assistant Project, University of Illinois.

[74] Scott Renner.
Location of Logical Errors in Pascal Programs with an Appendix on Implementation Problems in Waterloo Prolog/C.
April, 1982.
Internal report UIUCDCS-F-82-896, Knowledge Based Programmer's Assistant Project, University of Illinois.

[75] C. Rich.
Initial report on the Lisp programmer's apprentice.
*IEEE Transactions on Software Engeneering* SE-4(6):342-376, 1979.

[76] C. Rich.
Formal representation of plans in the Programmer's Apprentice.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 1044-1052". IJCAI, 1981.

[77] C. Rich and R. Waters.
*Abstraction, Inspection and Debugging in Programming.*
Technical Report AIM-634, MIT, Artificial Intelligence Laboratory, June, 1981.

[78] J. A. Robinson.
A machine oriented logic based on the resolution principle.
*Journal of the ACM* 12, January, 1965.

[79] Hartley Rogers, Jr.
*Theory of Recursive Functions and Effective Computability.*
McGraw-Hill Book Company, 1967.

[80] P. Roussel.
*Prolog: Manuel Reference et d'Utilisation.*
Technical Report, Groupe d'Intelligence Artificielle, Marseille-Luminy, September, 1975.

[81] Erik Sandewall.
Programming in an interactive environment: the LISP experience.
*Computing Surveys* , March, 1978.

[82] Ehud Y. Shapiro.
*Inductive Inference of Theories from Facts.*
Technical Report 192, Yale University, Department of Computer Science, February, 1981.

[83] Daniel G. Shapiro.
*Sniffer: a System that Understands Bugs.*
Technical Report AIM-638, MIT, Artificial Intelligence Laboratory, June, 1981.

[84] Ehud Y. Shapiro.
An algorithm that infers theories from facts.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* IJCAI, August, 1981.

[85] Ehud Y. Shapiro.
The model inference system.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* IJCAI, August, 1981.
Program demonstration.

[86] Ehud Y. Shapiro.
Alternation and the computational complexity of logic programs.
1982.
Submitted.

[87] D. Shaw, W. Swartout, and C. Green.
Inferring LISP programs from example problems.
In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 260-267. IJCAI, 1975.

[88] L. Siklossy and D. Sykes.
Automatic program synthesis for example problems.
In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 268-273. IJCAI, 1975.

[89] Douglas R. Smith.
A survey of synthesis of LISP programs from examples.
In *International Workshop on Program Contruction, Chateo de Bonas.* INRIA, 1980.

[90] E. Soloway, B. Woolf, P. Barth, and E. Rubin.
MENO-II: Catching runtime errors in novice's Pascal programs.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 975-977. IJCAI, 1981.

[91] Guy Lewis Steele, Jr. and Gerald Jay Sussman.
*The Revised Report on Scheme, a Dialect of Lisp.*
AI Memo 452, Massachusetts Institute of Technology Artificial Intelligence Laboratory, January, 1978.

[92] Philip D. Summers.
*Program Construction from Examples.*
PhD thesis, Yale University, 1976.
Computer Science Dept. research report No. 51.

[93] Phillip D. Summers.
A methodology for LISP program construction from examples.
*Journal of the ACM* 24:161-175, January, 1977.

[94] Gerald J. Sussman.
*Artificial Intelligence Series.* Number 1: *A Computer Model of Skill Acquisition.*
North-Holland, 1975.

[95] Warren Teitelman.
*INTERLISP Reference Manual.*
Technical Report, Xerox Palo Alto Research Center, September, 1978.

[96] M. H. van Emden.
A proposal.
*Logic Programming Newsletter* (2):11, 1981.

[97] Warren D. H. D. , Pereira L. M. , Pereira F. C. N.
Prolog - the language and its imlementation compared with Lisp.
In *Symposium on Artificial Intelligence and programming Languages*, pages 109-115. SIGART/SIGPLAN, August, 1977.

[98] R. M. Wharton.
Grammar enumeration and inference.
*Information and Control* 33:253-272, 1977.

[99] L. White and E. Cohen.
A domain strategy for computer program testing.
*IEEE Transactions on Software Engeneering* SE-6, May, 1980.

[100] P. H. Winston.
Learning structural descriptions from examples.
In P. H. Winston (editor), *The Psychology of Computer Vision*, . McGraw-Hill, New York, 1975.

[101] Niklaus Wirth.
Program development by stepwise refinement.
*Communications of the ACM* 14:221-227, April, 1971.

[102] Niklaus Wirth.
*Algorithms + Data Structures = Programs.*
Prentice Hall, 1976.

[103] R. M. Young, G. D. Plotkin, and R. F. Lintz.
Analysis of an extended concept learning task.
In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 285. IJCAI, August, 1977.