

CS-1993-11

**Exploiting Program Schemata in a Prolog Tutoring System**

Timothy S. Gegg-Harrison

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

April 1993



**Exploiting Program Schemata  
in a  
Prolog Tutoring System**

by

Timothy S. Gegg-Harrison

April 20, 1993

Supervised by Donald W. Loveland

Dissertation submitted in partial fulfillment  
of the requirements for the degree  
of Doctor of Philosophy  
in the Department of Computer Science  
in the Graduate School of  
Duke University

This document is a reformatted version of the dissertation, and equivalent in content.  
Note, however, that some of the spacing and fonts differ slightly from the dissertation.

Copyright © 1993 by Timothy S. Gegg-Harrison  
All rights reserved



# Abstract

After their beginnings in computer-aided instruction, automated tutors have re-emerged as intelligent tutoring systems. These intelligent tutors have obtained considerable success by using results from cognitive psychology and artificial intelligence to permit non-traditional instruction which is tailored to their individual students. The success of these automated tutors is due to their precise understanding and modeling of both the student and the domain being taught. A common measure of the robustness of an automated tutor is the size of the domain that it can understand. The schema-based Prolog tutor described in this dissertation is capable of recognizing a larger class of programs than existing Prolog tutors. By using powerful generalized transformations, our Prolog tutor can generate this class of programs from a very small set of normal form programs. Thus, our Prolog tutor recognizes a larger class of programs using fewer normal form programs than existing Prolog tutors. One of the biggest shortcomings of automated tutors for complex domains is their lack of a characterization of their domain. In computer programming tutors, for example, no current tutoring system has the ability to precisely state the class of programs that its tutor is capable of understanding. In addition to being more robust than existing programming tutors, our Prolog tutor provides a characterization of the class of programs it understands.



## Acknowledgements

I would like to thank my advisor, Don Loveland, for his guidance and support throughout my graduate studies at Duke University. His insistence on clarity and precision, Although frustrating at times, has lead to a vastly improved final product. I am grateful to him for that.

I would also like to acknowledge the efforts of my committee, Alan Biermann, Gopalan Nadathur, Mark Holliday, and Ruth Day. They all provided helpful comments which have greatly enhanced this dissertation. Alan's optimism and excitement about my research coupled with Gopalan's eye to detail have provided me with the motivation and guidance necessary to complete this dissertation. I would like to express a special thanks to Ruth who provided me with a pleasant and intellectually stimulating environment in her research lab. In addition to teaching me a lot about cognitive psychology, she also provided me with an invaluable "escape" from my computer science studies.

My career at Duke University was considerably longer and much more involved than I ever imagined. I want to thank my wife, Barbara, for putting up with me during these times. She provided me with support when I needed it. She also produced the figures that appear throughout the dissertation. Finally, I would like to thank my children, Whitney and Ryan. Dad's finally graduating!



# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Chapter 1 - Overview</b>	<b>1</b>
<b>1.1 Teaching Prolog</b>	<b>2</b>
<b>1.2 Debugging Prolog Programs</b>	<b>3</b>
<b>Chapter 2 - Logical Foundations</b>	<b>9</b>
<b>2.1 Preliminary Definitions</b>	<b>9</b>
<b>2.2 Declarative Semantics of Logic Programs</b>	<b>13</b>
<b>2.3 Procedural Semantics of Logic Programs</b>	<b>15</b>
<b>2.4 Transforming Logic Programs</b>	<b>17</b>
<b>2.4.1 Unfolding/Folding Logic Programs</b>	<b>18</b>
<b>2.4.2 Termination of Logic Programs</b>	<b>30</b>
<b>Chapter 3 - ADAPT Debugger</b>	<b>35</b>
<b>3.1 Automated Program Debugging</b>	<b>35</b>
<b>3.1.1 Algorithm Recognition</b>	<b>35</b>
<b>3.1.2 Bug Detection</b>	<b>37</b>
<b>3.2 Previous Approaches to Automated Debugging</b>	<b>38</b>
<b>3.2.1 LAURA</b>	<b>38</b>
<b>3.2.2 Lisp Tutor</b>	<b>40</b>

3.2.3	PROUST	41
3.2.4	TALUS	43
3.2.5	APROPOS2	46
3.3	ADAPT's Approach to Automated Program Debugging	48
3.3.1	ADAPT's Approach to Algorithm Recognition	50
3.3.2	ADAPT's Approach to Bug Detection	57
Chapter 4	Characterizing ADAPT's Domain	61
4.1	Program Transformations	64
4.2	Decomposing/Composing Programs	75
4.3	Standard Recursive Form Programs	81
4.4	Constructing Intelligent Tutoring Systems	102
Chapter 5	Teaching Prolog Programming	107
5.1	Basic Prolog Schemata	108
5.2	Schema-Based Instruction	110
Chapter 6	Summary	119
6.1	Major Contributions	119
6.2	Future Research	119
6.3	Conclusions	123
	Bibliography	124
	Appendix A. Most Specific Generalizations	131
	Appendix B. Sample Analyses from the ADAPT Debugger	138

# Chapter 1

## Introduction

Automated tutors have been around for several years. They began with the study of *computer-aided instruction* where instructional materials (e.g., textbooks or workbooks) were simply stored in computer files and students were able to use them in a variety of structured ways. *Intelligent computer-aided instruction* (also known as *intelligent tutoring systems*) emerged in the 1970s with the goal of enhancing the instruction available on traditional CAI systems by facilitating instruction that was tailored to its individual students. Not only did this shift in focus enable an enhancement over traditional CAI, it also provided a style of teaching that was not possible in traditional classrooms with one teacher per thirty students. This adaptive tutoring is made possible by a precise understanding and modeling of both the student and the domain being taught. Intelligent tutors now exist for a wide range of domains, including arithmetic, medical diagnosis, chemistry, geography, economics, electronics, game playing, and computer programming.

Many tutoring systems have been implemented that teach novices how to program a computer. There are automated tutors for nearly every programming language. With the exception of the LISP Tutor (Anderson & Reiser, 1985), however, existing programming language tutoring systems are actually not complete tutoring systems, but merely the program debugging component (or diagnosis component). The authors of these systems claim that tutoring and student modeling components can be added to their systems, but give no advice as to how to build these components. Although the focus of our research has been on the construction of a robust automated Prolog program debugger, the current implementation of the system also contains a tutoring component for recursive Prolog programming.

One of the biggest shortcomings of automated tutors for programming or any complex domain is their lack of a precise characterization. We have shown that it is possible to precisely define a non-trivial class of recursive list processing Prolog programs which empirical studies have shown to be sufficient for novice Prolog programmers. When supplemented with a general equivalence-preserving transformation scheme, this class of programs can be represented by a very small set of representative normal form programs. The resultant debugger is more robust than current Prolog debuggers and provides a characterization of the class of programs it can recognize.

The underlying knowledge base of our schema-based Prolog tutor is a Prolog schema library which consists of a selected set of recursive list processing programs and their abstractions (or schemata). The schema library's schemata represent classes of Prolog programs which use similar programming techniques, while its programs represent normal form implementations which can be systematically transformed into dozens of other distinct implementations. The Prolog schemata can be combined to form schema hierarchies where each parent node is a generalization of each of its children. The schema-based instructional technique employed by the tutoring system facilitates the learning of the general programming techniques encapsulated in the program schemata by requiring the student to complete templates (which are abstractions of the assigned program). Thus, students are encouraged to write structured Prolog programs. The schema hierarchies provide a wealth of similar programs that can be used to tailor the instruction to the students' individual needs.

Furthermore, by dynamically adjusting the difficulty of the template that the student is asked to complete, the tutor is able to adapt the lesson to its students' capabilities.

ADAPT (Automated Debugger for an Adaptive Prolog Tutor) uses the normal form Prolog programs of the schema library to generate a class of alternative implementations. This class of representative programs includes programs which accumulate results and reverse the processing. The assessment of the correctness of the student's program is based on the comparison of the student's program to the closest matching representative program. This assessment can be improved by unfolding clauses in the student's program and permuting subgoals in the representative program to obtain a closer match. Because Prolog's computation rule processes subgoals from left-to-right, an infinite loop can be introduced into an otherwise correct program by reordering the subgoals of some of its clauses. ADAPT prevents programs which contain loops by requiring them to be structural recursive which has been shown to be sufficient to ensure universal termination (Plümer, 1990).

## 1.1 Teaching Prolog

Recursion is a very difficult concept for most novice programmers to grasp (Anderson, Pirolli, & Farrell, 1988; Taylor & du Boulay, 1987; van Someren, 1985). One of the major causes of difficulties is the lack of structured programming constructs for recursive programming languages. Our schema-based Prolog tutor attempts to alleviate some of these difficulties by providing the novice Prolog programmer with a set of standard structures (or *program schemata*) with which to build complex and interesting recursive Prolog programs. Empirical evidence indicates that this is one of the keys to the development of effective programmers. Adelson (1981) has shown that a major difference between novice and expert programmers is in their organization of programming concepts. She found that novices tend to use a syntax-based organization, while expert programmers use a more abstract hierarchical organization of algorithms. Van Someren (1990) found similar differences between novice and expert Prolog programmers. He found that experts' criteria for categorizing Prolog programming exercises was based on the structure of the problems whereas novices tended to categorize the same programming exercises using more superficial criteria. Conventional programming languages provide a set of structured programming constructs. However, there are no equivalent constructs in Prolog.

Many Prolog programs share a common underlying structure or schema (Gegg-Harrison, 1989; Tinkham, 1990). Such classes of programs exemplify general programming techniques. For example, the method of recursively processing all elements of a list is a programming technique which is general enough to encompass programs which determine if their argument is a valid list (`list/1`), count the number of elements in a list (`length/2`), sum a list of elements (`sum/2`), find the product of a list of elements (`product/2`), sort a list of elements (`insertion_sort/2`), reverse the order of all the elements in a list (`reverse/2`), append two lists together (`append/3`), and enqueue an element onto the end of a list (`enqueue/3`). These programs are all captured by the following global list processing (`glp`) schema:

```
glp_schema([], <, α1>i).
glp_schema([H|T], <, α2>i) :- <θ1(α3<, α4>j, >glp_schema(T<, α5>i)<, θ2(α6<, α7>k)>.
```

where  $\alpha_n$  is a first-order schema variable which can be replaced by any valid Prolog term,  $\theta_i$  is a second-order schema variable which can be replaced by any valid Prolog predicate, and  $\langle t \rangle^n$  is a term consisting of  $n$  (possibly distinct) instances of  $t$ . Note that our Prolog Schema Language has two types of first-order variables: standard Prolog variables and schema variables. Prolog variables are represented by character strings which begin with a capital letter (e.g., H and T). First-order schema variables permit the generalization of terms, while second-order schema variables permit the generalization of predicates.

Prolog schemata are the basic constructs of a *structured Prolog* for simple recursive list processing providing the core knowledge base for the recursive programming segment of the Prolog tutoring system (Gegg-Harrison, 1991). Schema-based instruction emphasizes the learning of general programming techniques or schemata. This approach of presenting Prolog to novices through a set of basic construct schemata is in sharp contrast to the approach advocated by the current Prolog texts. After presenting the list data structure and possibly giving the Prolog code for `list/1` (which succeeds if its argument is a list), most introductory and intermediate Prolog texts (e.g., (Clocksin & Mellish, 1987; Sterling & Shapiro, 1986)) present `member/2` (which succeeds if its element argument is contained in its list argument) as the first recursive list processing program. Rather than continuing with more examples from this class of programs, these texts immediately switch to a global list processing program like `append/3` (which succeeds if its third list argument is its second list argument concatenated onto the end of its first list argument).

There are four fundamental differences between these traditional approaches and our approach to teaching recursive Prolog programming. First of all, we present global list processing programs like `append/3` and `length/2` (which succeeds if its integer argument is a count of the number of elements in its list argument) before the non-global list processing programs like `member/2` and `position/3` (which succeeds if its element argument occurs in a specified position in its list argument) under the assumption that the termination condition is more straightforward. Secondly, we introduce recursion through iteration since iteration is a special case of recursion (i.e., tail recursion).<sup>1</sup> Thirdly, our approach stresses the importance of classes of programs and promotes the acquisition of basic Prolog programming constructs by explicitly presenting Prolog schemata and presenting programs in the same class as a single coherent unit. Finally, teaching with schemata provides the tutor with a more accurate measurement of the student's problem solving capability permitting instruction that is tailored to the individual needs of the students.

The tutor begins by selecting the schema (i.e., programming technique) to be taught. Then an appropriate problem is selected to test the student's knowledge of the programming technique. A template is then selected to provide the student with the basic structure of the program solution. In addition, the general programming technique is described and examples and analogies are given when they are appropriate and available. Then the student is asked to write the desired program by completing the template.

The basic lesson plan is the same for each student. Prior to presenting the first lesson, the

---

<sup>1</sup>Note, however, that our approach is flexible enough to modify its presentation to teach tail recursion after general recursion if that is more appropriate for a given student (assuming the existence of a student modeling component that could determine that a student naturally thinks recursively).

tutor gives the student an overview of the list data structure in Prolog. The student is then introduced to recursion through iteration using the tail recursive global list processing schema:

$$\begin{aligned} & \text{tr\_glp\_schema}([], \langle, \alpha_1 \rangle^i). \\ & \text{tr\_glp\_schema}([H|T], \langle, \alpha_2 \rangle^i) \quad :- \quad \langle \theta_1(\alpha_3, \langle, \alpha_4 \rangle^j, \rangle, \text{tr\_glp\_schema}(T, \langle, \alpha_5 \rangle^i)). \end{aligned}$$

This schema enables the introduction of recursion without the additional complications of more generalized forms of recursion which permit subgoals following the recursive call. Not only does tail recursion simplify recursion, it is also more "natural" for most students. Because of limitations on their memory, most human beings appear more capable of thinking about complex procedures iteratively than recursively (Anderson, Pirolli, & Farrell, 1988). Thus, tail recursion provides a familiar bridge for novices to cross into the otherwise overwhelming generality of recursion.

After the initial introductory lesson on recursion, the remaining lessons are tailored to the student's capabilities (Gegg-Harrison, 1992). This is done via the selection of the template and program that are assigned to the student. Individual lessons can either be expanded (i.e., additional remedial lessons can be inserted into the lesson plan) or modified (i.e., the template can be made more specific for struggling students or more general for unchallenged students).

## 1.2 Debugging Prolog Programs

Automated program debugging is a difficult task which requires recognizing the algorithm employed by the programmer and detecting any errors that exist in her program. Several attempts have been made to tackle either or both of these tasks for a number of programming languages. All approaches to automated program debugging require some form of plan library which consists of a collection of representative programs. Plan library approaches (e.g., the LISP Tutor (Anderson & Reiser, 1985), PROUST (Johnson, 1986), TALUS (Murray, 1988), and APROPOS2 (Looi, 1988)) decompose the problems into collections of well-defined operations. These collections of operations are stored in a library of plans. The general approach to constructing these debuggers is to build a system with an initial set of representative programs, run empirical studies on novice programmers to test the sufficiency of this set of programs, and then add new representative programs and repeat the empirical studies.

Because the decision to add new representative programs to the library is dictated by empirical studies, the set of programs in the library has no precise characterization. Often the introduction of new representative programs is justified because it captures a new *algorithm*. Although every computer scientist is familiar with the term algorithm, the determination of what constitutes a distinct algorithm for a given task is vague. For the APROPOS2 Prolog debugger, Looi (1988) attempts to capture classes of programs that "use one common strategy of solving the task" where "all the programs in a class can be explained in a common way." Because new representative programs are added as a result of empirical studies, however, Looi's selection of algorithms is actually arbitrary and inconsistent. For example, the `sorting/2` task has four algorithms: bubble sort, permutation sort, quicksort, and insertion sort. For the `reverse/2` task, he defines three algorithms: naive, inverse naive, and accumulator. But note that

insertion\_sort/2:

```
insertion_sort([],[]).
insertion_sort([H|T],R) :- insertion_sort(T,S),insert(S,H,R).
insert([],X,[X]).
insert([H|T],X,[X,H|T]) :- X<H,! .
insert([H|T],X,[H|M]) :- insert(T,X,M).
```

and reverse/2:

```
reverse([],[]).
reverse([H|T],R) :- reverse(T,S),append(S,[H],R).
append([],L,L).
append([H|T],X,[H|R]) :- append(T,X,R).
```

share the same basic top-level structure. Thus, there should be two more algorithms for `sorting/2` to cover the "inverse naive" and "accumulator" implementations of `insertion_sort/2`. This inconsistency can be eliminated by noting that all three algorithms for `reverse/2` are actually different implementations of the same algorithm where the common strategy for solving the task is that they all remove exactly one element from the front/back of the list. Note that this common strategy is also consistent with the `sorting/2` task since none of the other algorithms remove a single element from the front/back of the input list on each pass.

While plan library approaches analyze student programs by comparing them to a set of correct implementations, transformation approaches represent the correct implementation by a single normal form<sup>2</sup> program and attempt to transform the student's program into the canonical form of the normal form program. The only system that uses this technique is the LAURA system for debugging Fortran programs (Adam & Laurent, 1980). LAURA represents its normal form programs as graphs and transforms the student's Fortran program into this canonical graph form by using a set of equivalence-preserving transformation rules.

ADAPT also employs a transformational approach to automated program debugging. Rather than transforming the normal form and student's programs into another representation (e.g., LAURA's flow graphs), ADAPT transforms Prolog programs into Prolog programs. Furthermore, the major transformations (including accumulating results, reversing the processing, reordering clauses within the program, reordering subgoals within the clauses, and reordering arguments within the subgoals) are performed on the normal form program in order to make it match the student's program. This enables bug detection and explanation that is related to the student's Prolog program rather than some abstract representation of it.

By transforming normal form Prolog programs into Prolog programs (rather than some intermediate form), ADAPT maintains the strengths of plan library approaches with respect to bug explanation while relieving the instructor of the laborious and error-prone task of deriving all possible implementations for each new program that is introduced to the system. For example, the following "naive" implementation serves as a normal form for `reverse/2` programs:

---

<sup>2</sup>Adam and Laurent (1980) use the term *model*. We prefer term *normal form* to *model*, however, to avoid confusion with the use of the term *model* in the logic programming community.

```

(N1) reverse([], []).
(N2) reverse([H|T], R) :- reverse(T, S), append(S, [H], R).
(N3) append([], L, L).
(N4) append([H|T], X, [H|R]) :- append(T, X, R).

```

and is robust enough to capture all the correct implementations given in Looi's dissertation (Looi, 1988), including the naive, inverse naive, railway-shunt, difference-list, standard accumulator, and switch implementations. These implementations are *generated* from the normal form program through the application of a set of general equivalence-preserving transformations which can be used on all the programs supported by ADAPT. Thus, ADAPT can generate the following "accumulator" `reverse/2` implementation:

```

(A1) reverse(L, R) :- reverse(L, [], R).
(A2) reverse([], L, L).
(A3) reverse([H|T], X, R) :- reverse(T, [H|X], R).

```

and the following "inverse naive" `reverse/2` implementation:

```

(I1) reverse([], []).
(I2) reverse(L, [H|T]) :- append(M, [H], L), reverse(M, T).
(I3) append([], L, L).
(I4) append([H|T], X, [H|R]) :- append(T, X, R).

```

and also the following "switch" `reverse/2` implementation:

```

(S1) reverse([], []).
(S2) reverse([X], [X]).
(S3) reverse([H|L], [G|M]) :- append(T, [G], L), reverse(T, X),
                             append(X, [H], M).
(S4) append([], L, L).
(S5) append([H|T], X, [H|R]) :- append(T, X, R).

```

These generated programs are capable of capturing programs whose subgoals have arguments in permuted order or within different structures (e.g., combining two arguments as in the difference list `reverse/2` implementation where the accumulator and result arguments are combined). For example, the following "bizarre" railway-shunt `reverse/2` program:

```

(SA1) reverse(L, R) :- rs_reverse(L, [R]).
(SA2) rs_reverse([], [X|X]).
(SA3) rs_reverse([H|T], S) :- rs_reverse(T, Y), insert(S, H, Y).
(SA4) insert([A|B], H, Y) :- enqueue([A], H, X), append(X, B, Y).
(SA5) enqueue([], E, [E]).
(SA6) enqueue([H|T], E, [H|R]) :- enqueue(T, E, R).
(SA7) append([], L, L).
(SA8) append([H|T], L, [H|R]) :- append(T, L, R).

```

is recognized as a correct program. ADAPT arrives at the assessment of the correctness of this

program by the following processing.<sup>3</sup> First of all, the normal form program is transformed into a standard accumulator implementation:

```
(TA1) reverse(L,R) :- acc_reverse(L,[],R).
(TA2) acc_reverse([],X,X).
(TA3) acc_reverse([H|T],X,R) :- Y=[H|X], acc_reverse(T,Y,R).
```

Then the accumulator is located and generalized in the student's program and the transformed normal form program is transformed into the same form by the following procedure. First of all, the original accumulator initializing clause is replaced by the argument reordering clauses (TB1)-(TC1):

```
(TA2) acc_reverse([],X,X).
(TA3) acc_reverse([H|T],X,R) :- Y=[H|X], acc_reverse(T,Y,R).
(TB1) reverse(L,R) :- acc_reverse(L,[R]).
(TC1) acc_reverse(L,[R|G]) :- acc_reverse(L,G,R).
```

Then the `acc_reverse(L,G,R)` subgoal in the clause (TC1) is unfolded and clause (TC1) is folded into the resultant recursive clause producing:

```
(TB1) reverse(L,R) :- acc_reverse(L,[R]).
(TB2) acc_reverse([], [X|X]).
(TB3) acc_reverse([H|T], [R|X]) :- Y=[R,H|X], acc_reverse(T,Y).
```

A comparison of the transformed normal form program and the student's program shows that the two programs would be the same if the `Y=[R,H|X]` and `acc_reverse(T,Y)` subgoals were permuted in clause (TB3) and the subgoal `insert(S,H,Y)` (where `S` is bound to `[R|X]`) in clause (SA3) produces the same answers as the subgoal `Y=[R,H|X]` in clause (TB3). Permuting the subgoals in clause (TB3) is possible since the first argument `T` of the recursive call `rs_reverse(T,Y)` is bound and the length of this argument has decreased. In order to show that of `insert(S,H,Y)` and `Y=[R,H|X]` produce the same answers, ADAPT must unfold the `insert(S,H,Y)` subgoal in clause (SA3) using clause (SA4) and then unfold the resultant subgoals using clauses (SA5)-(SA8) producing the following program:

```
(SA1) reverse(L,R) :- rs_reverse(L,[R]).
(SA2) rs_reverse([], [X|X]).
(SB3) rs_reverse([H|T],S) :- rs_reverse(T,Y), S=[A|B],
                             Y=[A,H|B].
```

Since there are no variables in common with the subgoals `rs_reverse(T,Y)` and `S=[A|B]`, they can be permuted and then the `S=[A|B]` subgoal can be embedded into the head. Furthermore, since the recursive call's first argument `T` is bound and decreasing there is no restriction on the mode

---

<sup>3</sup>Details of this process are given in Chapters 2 and 4. Specifically, a precise definition of *unfolding* is given in Section 2.4. For the present discussion, one can think of *unfolding* as merely the partial evaluation of a subgoal resulting in a simplified clause.

of the recursive call's second argument  $Y$  so the  $Y=[A,H|B]$  subgoal can be moved to precede the recursive call `rs_reverse(T,Y)` producing the following program:

```
(SA1) reverse(L,R) :- rs_reverse(L,[R]).
(SA2) rs_reverse([], [X|X]).
(SC3) rs_reverse([H|T],[A|B]) :- Y=[A,H|B],rs_reverse(T,Y).
```

which is identical to the transformed normal form program. Because the analysis is performed by leaving the student's program very close to its original form, bug explanations generated by the tutor can be given in terms of subgoals in the student's original program.

The focus so far has been on ADAPT's transformational approach to algorithm recognition. Schemata can be exploited in the task of bug detection as well. One set of bugs is associated with the schema that is used and is therefore *task* related. For example, a `glp`<sup>4</sup> schema-level bug is the invalid use of a single element list rather than the empty list in the base case clause as in the following `reverse/2` program:

```
reverse([X],[X]).
reverse([H|T],R) :- reverse(T,S),append(S,[H],R).
append([],L,L).
append([H|T],X,[H|R]) :- append(T,X,R).
```

A second set of bugs is associated with the technique (Brna, Bundy, Dodd, Eisenstadt, Looi, Pain, Robertson, Smith, & van Someren, 1991) that is used and is therefore *implementation* related. For example, an accumulator technique-level bug is the attempted use of a single argument for both the accumulator and result as in the following incorrect `reverse/2` program:

```
reverse(L,R) :- reverse(L, []).
reverse([],R).
reverse([H|T],R) :- R=[H|R],reverse(T,R).
```

Note that the previous program also contains an example of a third class of bugs that is not related to the task or implementation, but rather is a set of general *language* related bugs. It contains a general Prolog language-level bug that results from an invalid attempt at variable reassignment (e.g.,  $R=[H|R]$ ).

ADAPT has been tested on all 55 `reverse/2` implementations given in Looi's dissertation (Looi, 1988) and an additional set of 70 `reverse/2` implementations ranging from the above "bizarre" railway-shunt implementation to the following naive `reverse` implementation which invokes `enqueue/3` (which enqueues a single element onto the end of a list) in place of `append/3`:

```
(T1) reverse([], []).
(T2) reverse([H|T],R) :- reverse(T,S),enqueue(H,S,R).
(T3) enqueue(X,[],[X]).
```

---

<sup>4</sup>Note that the `glp` schema was given previously in this chapter.

(T4) `enqueue(X, [H|T], [H|R]) :- enqueue(X, T, R).`

This dissertation presents a Characterization Theorem that ADAPT is capable of recognizing **all** correct `reverse/2` program implementations which:

1. remove a single element from the front or back of their input list
2. use simple variations of standard `append/3` for input decomposition and output composition
3. restrict the use of increasing arguments (i.e., arguments which increase in length on each pass) to those that are necessary for the computation (e.g., accumulators for outputs)
4. use a single recursive clause

using "naive" `reverse/2` as the only normal form program. The set of 125 `reverse/2` implementations used to test the ADAPT debugger are given in Appendix B. That set contains over 40 distinct correct implementations of `reverse/2`.

In addition to being able to recognize a much larger set of correct solutions than APROPOS2, ADAPT also overcomes five of APROPOS2's major shortcomings. First of all, ADAPT limits the number of program implementations that the system implementor (or teacher) must provide for each problem by maintaining normal form programs which are transformed into other implementations through a robust equivalence-preserving transformation scheme which introduces *general* programming techniques (e.g., accumulating results). Secondly, ADAPT is able to recognize subgoals which take their arguments in different orders (e.g., `enqueue(H, S, R)` in place of `append(S, [H], R)`) or forms (e.g., `rs_reverse(L, [R])` in place of `acc_reverse(L, [], R)`) from the subgoals of the normal form program. Thirdly, ADAPT is capable of verifying that its students' programs are loop-free by requiring them to be structural recursive (Plümer, 1990) which is a sufficient condition for termination for the class of programs supported by the tutoring system. Fourthly, ADAPT provides enhanced bug explanation by separating bugs into task-related, implementation-related, and general language-related bugs. Finally, ADAPT is the only debugger to provide a characterization of the class of programs it can recognize.

This completes the overview which provided a high level description and motivation of our research. The details are contained in the remaining chapters. Chapter 2 provides the basic terminology necessary to describe logic programs. Chapter 3 reviews the field of automated program debugging and provides the details of the ADAPT program debugger. The theoretical foundations of our research are presented in Chapter 4. That chapter includes formal descriptions of the transformations and provides a statement and proof of the Characterization Theorem which defines the class of `reverse/2` programs supported by the ADAPT debugger. The role of schemata in the teaching of recursion to novice Prolog programmers is highlighted in Chapter 5. The last chapter summarizes the major contributions of our research and discusses possible directions for future research. There are also two appendices. The first appendix provides a new algorithm for finding most specific generalizations. The second appendix gives sample program analyses that were generated by the ADAPT debugger.

## Chapter 2

# Logical Foundations

This chapter presents the basic terminology and notation that are necessary to formally discuss logic programs. Unless otherwise stated, the definitions given in this chapter are standard definitions used throughout the logic programming community. It is not intended to be a complete introduction to Prolog and its logical foundations. The interested reader is referred to (Lloyd, 1987) for a thorough treatment of logic programming.

### 2.1 Preliminary Definitions

We begin by giving the formal definition for terms which can be combined to form well-formed formulas for first-order predicate logic. We assume the existence of an alphabet containing 6 disjoint sets of symbols:

1.  $\mathcal{V}$  - set of variables
2.  $\mathcal{C}$  - set of constant symbols
3.  $\mathcal{F}$  - set of function symbols
4.  $\mathcal{P}$  - set of predicate symbols
5. Set of connectives:
  - $\neg$  - negation
  - $\wedge$  - conjunction
  - $\vee$  - disjunction
  - $\rightarrow$  - implication
  - $\leftrightarrow$  - equivalence
6. Set of quantifiers:
  - $\forall$  - universal quantifier
  - $\exists$  - existential quantifier

**Definition.** The set of terms,  $\mathcal{T}$ , is defined inductively as follows:

1. A variable is a term.
2. A constant is a term.
3. If  $f$  is an  $n$ -ary function symbol and  $t_1, \dots, t_n$  are terms, then  $f(t_1, \dots, t_n)$  is a term.

**Definition.** A ground term is a term containing no variables.

**Definition.** The set of atoms,  $\mathcal{A}$ , consists of all  $p(t_1, \dots, t_n)$  such that  $p$  is an  $n$ -ary predicate symbol and  $t_1, \dots, t_n$  are terms.

**Definition.** A ground atom is an atom containing no variables.

**Definition.** The set of well-formed formulas,  $\mathcal{W}$ , is defined inductively as follows:

1. An atom is a well-formed formula.
2. If  $F$  and  $G$  are well-formed formulas, then  $\neg F$ ,  $F \wedge G$ ,  $F \vee G$ ,  $F \rightarrow G$ , and  $F \leftrightarrow G$  are well-formed formulas.<sup>5</sup>
3. If  $F$  is a well-formed formula and  $x$  is a variable, then  $(\forall x F)$  and  $(\exists x F)$  are well-formed formulas.

**Definition.** The scope of  $\forall x$  in  $(\forall x F)$  is  $F$ . Likewise, the scope of  $\exists x$  in  $(\exists x F)$  is  $F$ .

**Definition.** An occurrence of a variable  $x$  is a bound occurrence in formula  $F$  if it occurs in a part of  $F$  of the form  $(\forall x G)$  or  $(\exists x G)$ . Otherwise, it is a free occurrence in  $F$ .

**Definition.** A variable  $x$  is bound in formula  $F$  iff some occurrence of  $x$  is bound in  $F$ .

**Definition.** A variable  $x$  is free in formula  $F$  iff some occurrence of  $x$  is free in  $F$ .

**Definition.** A closed formula is a formula with no free occurrence of any variable.

**Definition.** The formula  $\forall(F)$  denotes the universal closure of  $F$  which is the formula obtained by adding a universal quantifier for every free variable in  $F$ .

**Definition.** The formula  $\exists(F)$  denotes the existential closure of  $F$  which is the formula obtained by adding an existential quantifier for every free variable in  $F$ .

**Definition.** A literal is an atom or a negated atom.

**Definition.** A p-literal is a literal with predicate symbol  $p$ .

**Definition.** An expression is either a term, a literal, or the conjunction or disjunction of literals. A simple expression is either a term or an atom.

A notation is needed to distinguish a subexpression within an expression. For example, it is useful to be able to say that  $x$  is in the  $\langle 2, 3 \rangle^{\text{th}}$  place of  $\theta(u, \phi(v, w, x, y), z)$  which means that  $x$  is the third argument in the second argument in  $\theta(u, \phi(v, w, x, y), z)$ . This notion is formalized in the following definitions.

---

<sup>5</sup>Note that  $F \rightarrow G$  can equivalently be written  $G \rightarrow F$ .

**Definition.** An expression  $s$  is a subexpression of expression  $t$  if:

1.  $s = t$ .
2.  $t$  is of the form  $\theta(t_1, \dots, t_m)$  where  $\theta \in \mathcal{F} \cup \mathcal{P}$  and  $\exists i \in \{1, \dots, m\}$  such that  $s$  is a subexpression of  $t_i$ .
3.  $t$  is of the form  $t_1 \wedge \dots \wedge t_m$  or  $t_1 \vee \dots \vee t_m$  and  $\exists i \in \{1, \dots, m\}$  such that  $s$  is a subexpression of  $t_i$ .

**Definition.** A subexpression  $s$  is in the  $I^{\text{th}}$  place in expression  $t$  if:

1.  $I = \langle \rangle$  and  $s = t$ .
2.  $I = \langle i_1, \dots, i_n \rangle$  if  $t$  is of the form  $\theta(t_1, \dots, t_m)$  where  $\theta \in \mathcal{F} \cup \mathcal{P}$ ,  $i_1 \leq m$ , and  $s$  is in the  $\langle i_2, \dots, i_n \rangle^{\text{th}}$  place in  $t_{i_1}$ .
3.  $I = \langle i_1, \dots, i_n \rangle$  if  $t$  is of the form  $t_1 \wedge \dots \wedge t_m$  or  $t_1 \vee \dots \vee t_m$ , and  $s$  is in the  $\langle i_2, \dots, i_n \rangle^{\text{th}}$  place in  $t_{i_1}$ .

Now we want to define a special class of well-formed formulas called clauses which are the basic components of logic programs.

**Definition.** A clause has the form  $L_1 \wedge \dots \wedge L_n \rightarrow A$  where  $A$  is an atom and  $L_1 \wedge \dots \wedge L_n$  are literals.  $A$  is called the head and  $L_1 \wedge \dots \wedge L_n$  the body of the clause. We will also use the following shorthand notions for clauses:

$$A \leftarrow L_1, \dots, L_n \quad \text{-or-} \quad A \text{ :- } L_1, \dots, L_n$$

**Definition.** A goal is a clause with an empty head.

**Definition.** A unit clause is a clause with an empty body.

**Definition.** The empty clause, denoted  $\square$ , is the clause with empty head and empty body. This clause is to be understood as a contradiction.

**Definition.** A definite clause is a clause in which  $L_1, \dots, L_n$  are atoms.

**Definition.** A definite goal is a goal in which  $L_1, \dots, L_n$  are atoms.

**Definition.** A program is a set of clauses. A program which contains only definite clauses is called a definite program.

**Definition.** The set of all clauses in a program whose heads have the same predicate symbol  $p$  and arity  $n$  is called the definition of  $p/n$ .

As with any computer program, logic programs obtain their flexibility through the use of variables.

The instantiation of variables through substitutions is formalized in the following definitions.

**Definition.** A substitution  $\sigma$  is a finite set of the form  $\{v_1/t_1, \dots, v_n/t_n\}$  where each  $v_i$  is a variable, each  $t_i$  is a term distinct from  $v_i$ , and the variables  $v_1, \dots, v_n$  are distinct. Each element  $v_i/t_i$  is called a binding for  $v_i$ .

**Definition.** The composition  $\sigma\rho$  of substitutions  $\sigma = \{u_1/s_1, \dots, u_m/s_m\}$  and  $\rho = \{v_1/t_1, \dots, v_n/t_n\}$  is the substitution obtained from the set  $\{u_1/s_1\rho, \dots, u_m/s_m\rho, v_1/t_1, \dots, v_n/t_n\}$  by deleting any binding  $u_i/s_i\rho$  for which  $u_i = s_i\rho$  and deleting any binding  $v_j/t_j$  for which  $v_j \in \{u_1, \dots, u_m\}$ .

**Definition.** The expression  $E\sigma$  is the instance of  $E$  by  $\sigma$  if  $E\sigma$  is the expression obtained by simultaneously replacing each occurrence of the variable  $v_i$  in  $E$  by the term  $t_i$ , where  $\sigma = \{v_1/t_1, \dots, v_n/t_n\}$ .

**Definition.** Expressions  $E_1$  and  $E_2$  are variants of each other if there exist substitutions  $\sigma_1$  and  $\sigma_2$  such that  $E_1 = E_2\sigma_2$  and  $E_1\sigma_1 = E_2$ .

Now we want to define a special class of substitutions which unify a set of expressions. These substitutions are used by the resolution principle (Robinson, 1965) to give a procedural interpretation to logic programs.

**Definition.** A substitution  $\sigma$  is a unifier for a set of simple expressions  $S$  if  $S\sigma$  is a singleton.

**Definition.** A unifier  $\sigma$  is a most general unifier (mgu) for a set of simple expressions  $S$  if for each unifier  $\rho$  of  $S$ , there exists a substitution  $\tau$  such that  $\rho = \sigma\tau$ .

**Theorem.** Every non-empty set of unifiable simple expressions has a unique most general unifier up to variable renaming.

**Proof.** See (Robinson, 1965).

Unification provides a specialization of a set of expressions and is necessary to execute logic programs. When analyzing and classifying logic programs, however, it is often useful to find a generalization which captures the essence of a class of expressions. The following notions of generalization and most specific generalization are due to Plotkin (1970) and Reynolds (1970). We provide an improved algorithm for finding the most specific generalization for simple expressions in Appendix A.

**Definition.** An expression  $E$  is a generalization of a set of simple expressions  $S$  if for all  $s \in S$  there exists a substitution  $\sigma$  such that  $E\sigma = s$ .

**Definition.** An expression  $E$  is a most specific generalization (msg) for a set of simple expressions  $S$  if  $E$  is a generalization of  $S$  and for each generalization  $F$  of  $S$ , there exists a substitution  $\sigma$  such that  $E = F\sigma$ .

**Definition.** A set of simple expressions is compatible if every simple expression in the set has the same predicate symbol and arity.

**Theorem.** Every non-empty set of compatible simple expressions has a unique most specific generalization up to variable renaming.

**Proof.** See (Plotkin, 1970), (Reynolds, 1970), or Appendix B.

## 2.2 Declarative Semantics of Logic Programs

One possible semantics that can be assigned to a logic program is a declarative one in which the meaning of a logic program is the set of all logical consequences of the logic program.

**Definition.** An interpretation  $I$  of a closed well-formed formula  $F$  consists of a nonempty domain  $\mathcal{D}$  and an assignment of values to each constant, function symbol, and predicate symbol occurring in  $F$  as follows:

- An element of  $\mathcal{D}$  is assigned to each constant
- A mapping from  $\mathcal{D}^n \rightarrow \mathcal{D}$  is assigned to each  $n$ -ary function symbol
- A mapping from  $\mathcal{D}^n \rightarrow \{\text{true}, \text{false}\}$  is assigned to each  $n$ -ary predicate symbol

**Definition.** A valuation  $\nu$  is a mapping from variables of the alphabet to the domain of the interpretation.

The assignment of meaning to an arbitrary term  $t$  (denoted  $\hat{\nu}(t)$ ) with respect to interpretation  $I$  and valuation  $\nu$  is as follows:

- If  $t$  is a constant  $d$  then  $\nu(t) = d_I$  where  $d_I \in \mathcal{D}$
- If  $t$  is a variable  $x$  then  $\hat{\nu}(t) = \nu(x)$
- If  $t$  is of the form  $f(t_1, \dots, t_n)$  then  $\hat{\nu}(t) = f_I(\hat{\nu}(t_1), \dots, \hat{\nu}(t_n))$  where  $f_I$  is the function mapping of  $I$

The assignment of truth values for a closed well-formed formula  $F$  with respect to interpretation  $I$  and valuation  $\nu$  is determined as follows:

- If  $F$  is an atom of the form  $p(t_1, \dots, t_n)$  then the truth value is given by  $p_I(\hat{\nu}(t_1), \dots, \hat{\nu}(t_n))$  where  $p_I$  is the predicate mapping of  $I$ .
- If  $F$  is of the form  $\neg F_1, F_1 \wedge F_2, F_1 \vee F_2, F_1 \rightarrow F_2$ , or  $F_1 \leftrightarrow F_2$  then the truth value is given in the following table:

$F_1$	$F_2$	$\neg F_1$	$F_1 \wedge F_2$	$F_1 \vee F_2$	$F_1 \rightarrow F_2$	$F_1 \leftrightarrow F_2$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

- If  $F$  is of the form  $(\exists x G)$  then the truth value is `true` if  $F$  has truth value `true` for some valuation  $\hat{v}$  such that  $\hat{v}(y) = \hat{v}(y)$  for every  $y \neq x$ . The truth value is `false` otherwise.
- If  $F$  is of the form  $(\forall x G)$  then the truth value is `true` if  $F$  has truth value `true` for all valuations  $\hat{v}$  such that  $\hat{v}(y) = \hat{v}(y)$  for every  $y \neq x$ . The truth value is `false` otherwise.

**Definition.** A closed formula  $F$  is satisfiable if there exists an interpretation  $I$  which evaluates  $F$  to `true`. In this case, we say that  $I$  is a model of  $F$ .

**Definition.** A closed formula  $F$  is unsatisfiable if no interpretation  $I$  is a model of  $F$ .

**Definition.** A closed formula  $F$  is valid if every interpretation of  $F$  is a model of  $F$ .

**Definition.** A closed formula  $F$  is nonvalid if there exists an interpretation of  $F$  which is not a model of  $F$ .

**Definition.** A formula  $F$  is a logical consequence of formulas  $F_1, \dots, F_n$  if every model of  $F_1 \wedge \dots \wedge F_n$  is also a model of  $F$ .

Logic programs are written to be executed by querying them. We next define the declarative concept of correct answer to a query which provides a declarative description of the desired output from a logic program and goal.

**Definition.** Let  $P$  be a definite program and  $G$  be a definite goal. An answer for  $P \cup \{G\}$  is a substitution for variables of  $G$ .

**Definition.** Let  $P$  be a definite program,  $G$  be a goal of the form  $\leftarrow G_1, \dots, G_n$  and  $\sigma$  be an answer for  $P \cup \{G\}$ . Then  $\sigma$  is a correct answer for  $P \cup \{G\}$  if  $\forall (G_1\sigma \wedge \dots \wedge G_n\sigma)$  is a logical consequence of  $P$ .

Next we introduce special class of interpretations, Herbrand interpretations, to which we will restrict our attention since it has been shown that a logical formula has a model if and only if it has a Herbrand model.

**Definition.** The Herbrand universe of  $P$ ,  $\mathcal{U}_P$ , is the set of all ground terms which can be formed out of the constants and functions symbols appearing in  $P$ . If  $P$  has no constants then an arbitrary constant (e.g.,  $a$ ) is added to form ground terms.

**Definition.** The Herbrand base of  $P$ ,  $\mathcal{B}_P$ , consists of all ground atoms  $p(u_1, \dots, u_n)$  such that  $p$  is an  $n$ -ary predicate symbol of  $P$  and  $u_1, \dots, u_n$  are terms in  $\mathcal{U}_P$ .

**Definition.** A Herbrand interpretation,  $I_P$ , of logic program  $P$  is an interpretation which satisfies the following:

- The domain of  $I_P$  is  $\mathcal{U}_P$
- The constants of  $\mathcal{U}_P$  are assigned to themselves
- The mapping  $\mathcal{U}_P^n$  to  $\mathcal{U}_P$  defined by  $(t_1, \dots, t_n) \rightarrow f(t_1, \dots, t_n)$  is assigned to each  $n$ -ary function symbol  $f$ .

**Definition.** A Herbrand interpretation  $I_P$  is a Herbrand model if it is a model of  $P$ .

Van Emden and Kowalski (1976) have shown that the intersection of all Herbrand models is a Herbrand model which is called the least Herbrand model,  $M_P$ . They also showed that the least Herbrand model  $M_P$  is equivalent to the least fixed point of a function  $T_P$  which maps subsets of the Herbrand base to subsets of the Herbrand base as follows:

$$T_P(I) = \{A\sigma \in \mathcal{B}_P \mid A \leftarrow L_1, \dots, L_n \text{ is a clause in } P \text{ and there exists a substitution } \sigma \text{ such that } A\sigma \text{ is ground and } \{L_1\sigma, \dots, L_n\sigma\} \subseteq I\}$$

Application of  $T_P$  corresponds to one-step forward chaining using  $P$ , deriving ground atoms from ground atoms.  $T_P$  provides the link between the declarative and procedural semantics of the logic program  $P$ .

## 2.3 Procedural Semantics of Logic Programs

Another possible semantics for logic programs is a procedural one in which a procedure is defined which infers the desired logical consequences of the logic program. The refutation procedure described below is SLD-resolution (Apt & van Emden, 1982). We begin with basic definitions for the resolution inference rule (Robinson, 1965).

**Definition.** Let  $G$  be the goal  $\leftarrow A_1, \dots, A_m, \dots, A_n$  and  $C$  be the clause  $A \leftarrow B_1, \dots, B_l$ . Then  $G'$  is derived from  $G$  and  $C$  using mgu  $\sigma$  if the following conditions hold:

1.  $A_m$  is an atom in  $G$  ( $A_m$  is called the selected atom)
2.  $\sigma$  is an mgu of  $A_m$  and  $A$
3.  $G'$  is the goal  $\leftarrow A_1, \dots, A_{m-1}\sigma, \dots, B_1\sigma, \dots, B_l\sigma, A_{m+1}\sigma, \dots, A_n\sigma$  ( $G'$  is called the resolvent of  $G$  and  $C$ )

**Definition.** Let  $P$  be a definite program and  $G_0$  a definite goal. An SLD-derivation of  $P \cup \{G_0\}$  consists of a (finite or infinite) sequence  $G_0, G_1, \dots$  of goals, a sequence  $C_0, C_1, \dots$  of variants of program clauses of  $P$ , and a sequence  $\sigma_0, \sigma_1, \dots$  of mgu's such that each  $G_{i+1}$  is derived from  $G_i$  and  $C_i$  using  $\sigma_i$ .

**Definition.** Let  $P$  be a definite program and  $G_0$  a definite goal. An SLD-refutation of  $P \cup \{G_0\}$  is a finite SLD-derivation of  $P \cup \{G_0\}$  which has the empty clause  $\square$  as the last goal in the derivation. If  $G_n = \square$  then we say that the length of the SLD-refutation is  $n$ .

**Definition.** Let  $P$  be a definite program. The success set of  $P$  is the set of all  $A \in B$  tilde sub  $P$  such that  $P \cup \{A\}$  has an SLD-refutation.

**Definition.** Let  $P$  be a definite program and  $G_0$  be a definite goal. A computed answer  $\sigma$  for  $P \cup \{G_0\}$  is the substitution obtained by restricting the composition  $\sigma_1 \dots \sigma_n$  to the variables of  $G_0$ , where  $\sigma_1, \dots, \sigma_n$  is the sequence of mgu's used in an SLD-refutation of  $P \cup \{G_0\}$ .

**Definition.** Let  $P$  be a definite program and  $G_0$  be a definite goal. An SLD-tree for  $P \cup \{G_0\}$  is a tree satisfying the following conditions:

1. The root node is  $G_0$
2. Each node in the tree is a (possibly empty) definite goal
3. Nodes which are the empty clause have no children
4. If  $\neg A_1, \dots, A_m, \dots, A_n$  is a node in the tree where  $n > 0$  and  $A_m$  is the selected atom then for each input clause  $A \leftarrow B_1, \dots, B_l$  such that  $A_m$  and  $A$  are unifiable with mgu  $\sigma$ , the node has a child:

$$\neg A_1 \sigma, \dots, A_{m-1} \sigma, B_1 \sigma, \dots, B_l \sigma, A_{m+1} \sigma, \dots, A_n \sigma$$

Branches corresponding to successful derivations are called success branches, branches corresponding to failed derivations are called failure branches, and branches corresponding to infinite derivations are called infinite branches.

**Definition.** A finitely failed SLD-tree is an SLD-tree which contains only failure branches.

**Definition.** A computation rule is a function from a set of definite goals to a set of atoms such that the value of the function for a goal is the selected atom.

**Definition.** A search rule defines a strategy for searching SLD-trees to find success branches.

**Definition.** An SLD-refutation procedure is specified by a computation rule together with a search rule.

Prolog is a logic programming system which employs the computation rule which always selects the leftmost atom in the goal together with a depth-first search rule. Prolog's refutation procedure is incomplete in the sense that it is not guaranteed to find all success branches due to its depth-first search which may unproductively traverse an infinite branch in the SLD-tree. In addition to its lack of completeness, Prolog is also unsound in the sense that it may produce an incorrect answer because Prolog's unification algorithm does not implement the "occur check." The "occur check" ensures that a term  $t$  is not unified with a term which contains  $t$ . Since Prolog is used primarily as a programming language rather than as a theorem prover, the sacrifice of completeness and soundness for efficiency is an acceptable one as long as Prolog programmers are aware of the potential problems. Another efficiency "feature" of Prolog which can sacrifice completeness is the "cut" control facility which is denoted with an exclamation point "!" in Prolog programs. The "cut" is a goal which succeeds once and then fails so that if backtracking returns to the cut then the goal which caused the clause containing the cut to be activated is failed (i.e., the cut causes the remainder of that subtree to be pruned from the SLD-tree).

So far, we have restricted our attention to definite programs. By definition, these programs do not contain clauses with negative goals in their bodies. Prolog permits such negative information by incorporating the "negation as failure" (Clark, 1978) non-monotonic inference rule. This inference rule simply states that if every branch in the SLD-tree for goal  $G$  and program  $P$  is a failure branch then we can infer that  $\neg G$  is true.

## 2.4 Transforming Logic Programs

One of the most common differences between correct student programs is the order of arguments in subgoals, subgoals within clauses, and clauses within programs. Because we are only considering the case of universal termination (i.e., all derivations for a given query are finite), permuting arguments within subgoals and clauses within programs simply enables distinct implementations that must be considered. Permuting subgoals within clauses, on the other hand, enables distinct implementations while introducing the potential of infinite loops.

When applying program transformations, it becomes necessary to show that the semantics of the program has not changed (i.e., the transformed program is equivalent to the original program). We formalize the notion of program equivalence in the next definition. Intuitively, two programs are equivalent to each other for a given predicate if each program produces the same answer on some arbitrary query.

**Definition.** A program  $P$  is equivalent to program  $Q$  for  $p/n$  if both  $P$  and  $Q$  are structural recursive and both of the following hold:

1. every query  $p(t_1, \dots, t_n)$  that succeeds exactly  $m$  times for  $P$  also succeeds exactly  $m$  times for  $Q$  with the same variable bindings and every query  $p(t_1, \dots, t_n)$  that finitely fails for  $P$  also finitely fails for  $Q$
2. every query  $p(t_1, \dots, t_n)$  that succeeds exactly  $m$  times for  $Q$  also succeeds exactly  $m$  times for  $P$  with the same variable bindings and every query  $p(t_1, \dots, t_n)$  that finitely fails

for  $Q$  also finitely fails for  $P$

In this case, we say that program  $P$  is correct with respect to program  $Q$  for  $p/n$ .

### 2.4.1 Unfolding/Folding Logic Programs

Unfolding/folding (Burstall & Darlington, 1977; Tamaki & Sato, 1984) is a method of partially evaluating a Prolog program. It essentially amounts to executing one resolution step for a totally uninstantiated query. The set of clauses produced by unfolding a clause with respect to one of its subgoals is the set of clauses obtained by replacing the subgoal by the body of each clause whose head unifies with the subgoal. Thus, the size of the unfolded clause set is the number of clauses with heads that are unifiable with the selected subgoal. Folding is essentially the inverse operation to unfolding. The basic idea is to "back up" one resolution step by replacing a set of subgoals with a single subgoal. The body of the clause used in the folding must unify with the set of subgoals in another clause (which was previously obtained via unfolding) and then a new clause is formed that replaces the set of subgoals with the head of the folding clause. These notions are formalized in the following definitions.

**Definition.** Let  $X$  be a clause in program  $P$  of the form  $A \leftarrow B_1, \dots, B_m, C, D_1, \dots, D_n$  and let  $C_1, \dots, C_i$  be all the clauses in  $P$  whose heads are unifiable with  $C$  with mgu's  $\sigma_1, \dots, \sigma_i$ . Then the unfolded clause set of clause  $X$  with respect to subgoal  $C$  and program  $P$  is the set of clauses resulting from resolving  $X$  with  $C_j$  upon  $C$  for  $j \in \{1, \dots, i\}$ . In this case, we say that subgoal  $C$  is unfolded in clause  $A \leftarrow B_1, \dots, B_m, C, D_1, \dots, D_n$ .

**Definition.** Let  $X$  be a clause of the form  $A \leftarrow B_1, \dots, B_m$  and let  $Y$  be a clause of the form  $C \leftarrow D_1, \dots, D_n$ . Let  $B_{i+1}, \dots, B_{i+n}$  be a subsequence of  $B_1, \dots, B_m$  and let  $\sigma$  be a substitution such that the following hold:

1.  $\forall j \in \{1, \dots, n\} (B_{i+j} = D_j \sigma)$
2.  $\sigma$  substitutes distinct variables for the internal variables of  $Y$ , and moreover those distinct variables do not occur in  $\{A, B_1, \dots, B_i, B_{i+n+1}, \dots, B_m\}$
3.  $Y$  is the only clause whose head is unifiable with  $C \sigma$

Then the clause  $A \leftarrow B_1, \dots, B_i, C \sigma, B_{i+n+1}, \dots, B_m$  is the folded clause of clause  $X$  with respect to clause  $Y$ . In this case, we say that we fold clause  $Y$  into clause  $X$ .

It has been shown that equivalence is preserved in a logic program that undergoes the unfolding and folding under certain circumstances (Kanamori & Kawamura, 1988). Kanamori and Kawamura have shown that clause  $X$  may be folded into clause  $Y$  as long as:

1. clause  $Y$  was produced from the original program via unfolding clause  $X$
2. the predicate in the head of clause  $X$  does not appear (with the same arity) anywhere else in the original program

Note that by original program, we mean the clauses of the program prior to the above two step process of unfolding and folding. Consider the following program:

```
(A1) exists(E,L) :- found(L,E).
(A2) found(A,B) :- member(A,B).
(A3) member([E|T],E).
(A4) member([H|T],E) :- member(T,E).
```

If the found(L,E) subgoal is unfolded in clause (A1), then the following program is created:

```
(A2) found(A,B) :- member(A,B).
(A3) member([E|T],E).
(A4) member([H|T],E) :- member(T,E).
(A5) exists(E,L) :- member(L,E).
```

It is to this program that we need to apply the two step process of unfolding and folding. As such, it is this program that is the original program with respect to the two step process of unfolding and folding. The first step of the process requires unfolding the member(L,E) subgoal in clause (A5) producing:

```
(A2) found(A,B) :- member(A,B).
(A3) member([E|T],E).
(A4) member([H|T],E) :- member(T,E).
(A6) exists(E,[E|T]).
(A7) exists(E,[H|T]) :- member(T,E).
```

Given that the predicate exists/2 only appears in the head of clause (A5) and that clause (A7) was produced by unfolding clause (A5), it follows that folding clause (A5) into clause (A7) produces a semantically equivalent program:

```
(A6) exists(E,[E|T]).
(A8) exists(E,[H|T]) :- exists(E,T).
```

when the no longer used definitions for found/2 and member/2 are removed.

We need to provide some definitions pertaining to predicate arguments. Previously, in this chapter we presented the notions of subexpression and places within subexpressions. We want to apply this notion to arguments within a literal. First, we place some restrictions on the types of input arguments contained in our class of programs. Input arguments (i.e., those arguments that are ground upon invocation) are restricted to:

1. lists
2. invariant variables (i.e., variable arguments that are ground on invocation and remain unchanged through recursive calls)

An example of an invariant variable argument is the second argument of member/2:

$$\begin{aligned} & \text{member}([E|T], E). \\ \text{member}(L, E) & :- \text{member}(T, E), L = [H|T]. \end{aligned}$$

Note that although the value of this argument is variable on the initial invocation, it remains constant throughout the execution of the program.

**Definition.** An argument  $A$  is in position  $i$  of the literal  $p(t_1, \dots, t_n)$  if  $A = t_i$  or  $i = \langle j, k \rangle$  and  $\exists \sigma$  such that  $[H|T] \sigma = t_j$  and  $A$  is in position  $k$  in  $q(H \sigma, T \sigma)$ .

**Definition.** An argument in position  $i$  has mode input (or  $+$ ) in  $p(t_1, \dots, t_n)$  if the term in position  $i$  in  $p(t_1, \dots, t_n)$  is ground. An argument in position  $i$  has mode output (or  $-$ ) in  $p(t_1, \dots, t_n)$  if the term in position  $i$  in  $p(t_1, \dots, t_n)$  does not have mode input.

**Definition.** A partially ordered set  $(S, >)$  consists of a set  $S$  and a transitive and irreflexive binary relation  $>$  defined on the elements of  $S$ .

**Definition.** A partial ordering  $>_{wf}$  on a set of terms is said to be well-founded if it admits no infinite descending sequences  $s_1 >_{wf} s_2 >_{wf} s_3 >_{wf} \dots$  of unique terms.

The standard mathematical notion of  $>$  is a well-founded ordering on non-negative integers. Likewise, a proper sublist ordering where  $A >_{wf} B$  iff  $B$  is a proper sublist of  $A$  is a well-founded ordering on lists. We will restrict our attention to this well-founded ordering.

**Definition.** Argument position  $i$  is non-increasing in clause  $p(A_1, \dots, A_n) :- \theta_1, p(B_1, \dots, B_n), \theta_2$  if the term  $t_A$  is the  $i^{\text{th}}$  position in  $p(A_1, \dots, A_n)$  and the term  $t_B$  is the  $i^{\text{th}}$  position in  $p(B_1, \dots, B_n)$  and  $t_A \sigma \geq_{wf} t_B \sigma$  for all substitutions  $\sigma$  subject to the bindings produced by solutions to  $\theta_1$ . Likewise, argument position  $i$  is strictly decreasing in clause  $p(A_1, \dots, A_n) :- \theta_1, p(B_1, \dots, B_n), \theta_2$  if the term  $t_A$  is the  $i^{\text{th}}$  position in  $p(A_1, \dots, A_n)$  and the term  $t_B$  is the  $i^{\text{th}}$  position in  $p(B_1, \dots, B_n)$  and  $t_A \sigma >_{wf} t_B \sigma$  for all substitutions  $\sigma$  subject to the bindings produced by solutions to  $\theta_1$ .

**Definition.** Argument position  $i$  is non-decreasing in clause  $p(A_1, \dots, A_n) :- \theta_1, p(B_1, \dots, B_n), \theta_2$  if the term  $t_A$  is the  $i^{\text{th}}$  position in  $p(A_1, \dots, A_n)$  and the term  $t_B$  is the  $i^{\text{th}}$  position in  $p(B_1, \dots, B_n)$  and  $t_A \sigma \leq_{wf} t_B \sigma$  for all substitutions  $\sigma$  subject to the bindings produced by solutions to  $\theta_1$ . Likewise, argument position  $i$  is strictly increasing in clause  $p(A_1, \dots, A_n) :- \theta_1, p(B_1, \dots, B_n), \theta_2$  if the term  $t_A$  is the  $i^{\text{th}}$  position in  $p(A_1, \dots, A_n)$  and the term  $t_B$  is the  $i^{\text{th}}$  position in  $p(B_1, \dots, B_n)$  and  $t_A \sigma <_{wf} t_B \sigma$  for all substitutions  $\sigma$  subject to the bindings produced by solutions to  $\theta_1$ .

**Definition.** Argument position  $i$  is non-increasing in clause  $p(A_1, \dots, A_n) :- q(B_1, \dots, B_m)$  if for all argument positions  $j$  in  $q/m$  such that the term in position  $i$  in  $p(A_1, \dots, A_n)$  and the term in position  $j$  in  $q(B_1, \dots, B_m)$  are identical, position  $j$  is non-increasing in  $q/m$ . Similar definitions can be made for non-decreasing, strictly increasing, and strictly decreasing.

**Definition.** Argument position  $i$  is non-increasing in  $p/n$  if position  $i$  is non-increasing in every clause in the definition of  $p/n$ . Similar definitions can be made for non-decreasing, strictly

increasing, and strictly decreasing.

**Definition.** An argument position  $i$  is invariant in the clause  $p(A_1, \dots, A_n) :- \theta_1, p(B_1, \dots, B_n) \theta_2$  if the term  $t$  is the  $i^{\text{th}}$  position in  $p(A_1, \dots, A_n)$  and the same term  $t$  is the  $i^{\text{th}}$  position in  $p(B_1, \dots, B_n)$ .

**Definition.** An anonymous variable is the symbol `_` or it is a variable which occurs only once in a clause.

Let's look at some examples. Consider the following accumulator reverse/2 program:

```
reverse(L,R) :- reverse(L, [], R).
reverse([], L, L).
reverse([H|T], X, R) :- reverse(T, [H|X], R).
```

The first argument position in reverse/2 and the first argument position in reverse/3 (which contain the input list) are strictly decreasing. The second argument position in reverse/3 (which holds the accumulator) is strictly increasing. Finally, the second argument position of reverse/2 and the third argument position of reverse/3 (which contain the output list) are invariant.

Now let's return to our unfolding/folding procedure. Assume we are given the following standard Prolog definition for append/3:

```
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

but want to use it in a more specialized way to enqueue a single element onto the end of a list:

```
(A1) enqueue(L, E, R) :- append(L, [E], R).
(A2) append([], L, L).
(A3) append([H|T], X, [H|L]) :- append(T, X, L).
```

We can use unfolding/folding to produce a more compact definition of enqueue/3 which has its recursive processing explicit rather than using append/3 to recursively process the list. We begin by unfolding the `append(L,[E],R)` subgoal in clause (A1):

```
(A2) append([], L, L).
(A3) append([H|T], X, [H|L]) :- append(T, X, L).
(A4) enqueue([], E, [E]).
(A5) enqueue([H|T], E, [H|L]) :- append(T, [E], R).
```

Now we can fold clause (A1) into clause (A5) and remove the no longer invoked append/3 producing the desired program:

```
(A4) enqueue([], E, [E]).
(A6) enqueue([H|T], E, [H|L]) :- enqueue(T, E, L).
```

Note that clause (A1) is the only clause in the original program that contains the predicate enqueue/3. Thus, folding clause (A1) into clause (A5) produces a program (consisting of clauses (A4) and (A6)) which is semantically equivalent to the original program.

Unfolding is also useful in eliminating mutual recursion from some Prolog programs while keeping the recursion at the same level. Štěpánková and Štěpánek (1987) have shown that mutual recursion can be eliminated from any Prolog program. However, their technique has the disadvantage that it levels out the recursion. Plümer (1990) has shown that mutual recursion can be eliminated via unfolding for a special class of Prolog programs which have a feedback vertex in every maximal strongly connected component of their predicate dependency graphs. Any Prolog program which has a predicate that is eventually invoked from all mutually recursive procedures can be made non-mutually recursive by unfolding its clauses to the point where this predicate is invoked. This class of programs is formalized with the following definitions.

**Definition.** A predicate dependency graph for program  $P$  is a directed graph  $G(V,E)$  where  $V$  is the set of all predicate symbols occurring in  $P$  and  $\langle p,q \rangle \in E$  iff  $q$  occurs in the body of some clause defining  $p$ .

**Definition.** A set of vertices  $S$  is a strongly connected component of graph  $G(V,E)$  if  $\langle v,w \rangle \in S$  iff there is a path from  $v$  to  $w$  and from  $w$  to  $v$  in  $G(V,E)$ . A set of vertices  $S$  is a maximal strongly connected component if there is no strongly connected component  $T \subseteq V$  such that  $S \subset T$ .

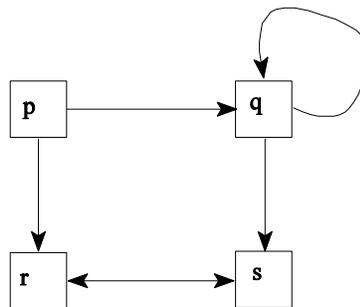
**Definition.** A set of vertices  $v_1, \dots, v_n$  is a cycle in graph  $G(V,E)$  if  $n > 2$ ,  $v_1 = v_n$ , and  $i \in \{1, \dots, n-1\} \langle v_i, v_{i+1} \rangle \in E$ .

**Definition.** A vertex  $v$  is a feedback vertex of graph  $G(V,E)$  if  $v$  is contained in every cycle of  $G(V,E)$ .

Let's look at an example. Assume we have the following Prolog program:

```
p :- q,r.
q :- s,q.
r :- s.
s :- r.
```

This program has the following predicate dependency graph:



and it contains the following strongly connected component (which happens to be maximal in this case):



Also note that both node  $r$  and  $s$  are feedback vertices. Plümer (1990) has shown that mutual recursion can be eliminated via unfolding for a large class of Prolog programs. Non-mutually recursive programs have predicate dependency graphs which contain no cycles. For such programs, the predicate dependency graph defines a partial ordering on the predicates.

The following definitions of logic program complexity, complete definition of  $p/n$ , and main predicate are not standard definitions.

**Definition.** Let  $G(V,E)$  be the predicate dependency graph for program  $P$ . Then for each predicate  $p$  that occurs in the head of some clause in  $P$ , the complexity of  $p$  is defined inductively as follows:

1.  $\text{complexity}(p) = 0$  if all  $q \neq p$  it follows that  $\langle p, q \rangle \notin E$
2.  $\text{complexity}(p) = 1 + \max\{\text{complexity}(q) \mid p \neq q \text{ and } \langle p, q \rangle \in E\}$  otherwise

We need to have the notion of a set of clauses that constitute a complete definition of a predicate (i.e., all the clauses whose heads match the predicate or the predicate of any of these clauses' auxiliary subgoals, etc.). The following inductive definition formalizes this notion.

**Definition.** The complete definition of  $p/n$  in a program is defined inductively as follows:

1. If  $\text{complexity}(p/n) = 0$  then the complete definition of  $p/n$  is the definition of  $p/n$ .
2. If  $\text{complexity}(p/n) \neq 0$  then the complete definition of  $p/n$  is the set of all clauses in the definition of  $p/n$  unioned with the set of all clauses in the complete definition of  $q/m$  (for each subgoal of the form  $q(t_1, \dots, t_m)$  in each of the clauses in the definition of  $p/n$ ).

Consider the following reverse/2 program:

```

reverse([], []).
reverse(L,R) :- enqueue(H,T,L), reverse(T,S), append(S, [H], R).
enqueue(A,B,C) :- append(A, [B], C).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
  
```

The following partial ordering is defined:  $\text{reverse}/2 > \text{enqueue}/3 > \text{append}/3$  for this program. The complexity of  $\text{append}/3$  is 0. The predicate  $\text{enqueue}/3$  has a complexity of 1 and  $\text{reverse}/2$  has a complexity of 2.

Another important notion is that of a main predicate for a logic program. Intuitively, a main predicate is one that is not an auxiliary subgoal to some other predicate in the program (i.e., there are no arcs from other predicates to it in the program's predicate dependency graph).

**Definition.** A predicate  $p/n$  is a main predicate in program  $P$  if the predicate dependency graph  $G(V,E)$  for  $P$  is such that  $\langle q, p \rangle \in E$  iff  $p = q$ .

Now consider the following program which defines two mutually recursive predicates where  $\text{even\_length}/1$  succeeds if the list contains an even number of elements and  $\text{odd\_length}/1$  succeeds if the list contains an odd number of elements:

```
(B1) even_length([]).
(B2) even_length([H|T]) :- odd_length(T).
(B3) odd_length([H|T]) :- even_length(T).
```

The predicate dependency graph for this program is:



The entire graph is a maximal strongly connected component and it has a feedback vertex (actually both nodes are feedback vertices) so the mutual recursion can be removed by unfolding the last two clauses producing the following program:

```
(B1) even_length([]).
(B4) even_length([H,X|T]) :- even_length(T).

(B5) odd_length([H]).
(B6) odd_length([H,X|T]) :- odd_length(T).
```

Another use of unfolding is in making explicit unifications implicit by unfolding them into the head of the clause. For example:

$$p(L, X, R) \text{ :- } L=[H|T], p(T, X, M), R=[H|M].$$

can be unfolded to produce a more concise clause. The  $L=[H|T]$  subgoal can be unfolded (into the head of the clause) producing:

$$p([H|T], X, R) \text{ :- } p(T, X, M), R=[H|M].$$

Note, however, that the  $R=[H|M]$  subgoal cannot necessarily be unfolded into the head since such a transformation actually permutes the order of the subgoals in the clause.

It is well known that the predicate `member/2` (which succeeds if its element argument is contained in its list argument) can be defined in terms of the predicate `append/3` as in the following definition (where the list argument is the first argument and the element argument is the second argument):

```
(C1) member(L, E) :- append(_, [E|_], L).
(C2) append([], L, L).
(C3) append([H|T], X, [H|L]) :- append(T, X, L).
```

Applying unfolding/folding enables the transformation of this program into the "standard" form of the `member/2` program. To do this, we unfold the `append(_, [E|_], L)` subgoal in clause (C1) producing the following program:

```
(C2) append([], L, L).
(C3) append([H|T], X, [H|L]) :- append(T, X, L).
(C4) member([E|T], E).
(C5) member([H|T], E) :- append(_, [E|_], T).
```

Then clause (C1) is folded into clause (C5) and the no longer invoked `append/3` is removed producing the following program:

```
(C4) member([E|T], E).
(C6) member([H|T], E) :- member(T, E).
```

Once again, note that the predicate `member/2` occurs only the head of clause (C1) in the original program so the transformed program is semantically equivalent to the original program.

The use of unfolding/folding procedure that was just shown with `member/2` is a general way of permuting and restructuring arguments within our class of logic programs. We can use unfolding/folding to verify that a subgoal in the student's program is equivalent to a subgoal in the target program. Let's look at a couple of examples of subgoal comparisons. Let's start with the following target implementation for `reverse/2`:

```
(A1) reverse([], []).
(A2) reverse([A|B], C) :- reverse(B, D), add_element(A, D, C).
(A3) add_element(X, [], [X]).
(A4) add_element(X, [H|T], [H|R]) :- add_element(X, T, R).
```

and see if this program is equivalent to the "standard" naive reverse/2:

```
(NNF1) reverse([], []).
(NNF2) reverse([H|T], R) :- reverse(T, S), append(S, [H], R).
(NNF3) append([], L, L).
(NNF4) append([H|T], X, [H|R]) :- append(T, X, R).
```

First, the variables are bound ( $H \leftarrow A$ ,  $T \leftarrow B$ ,  $R \leftarrow C$ ,  $S \leftarrow D$ ) and then each of the subgoals is replaced by a common subgoal producing the following extended programs:

```
(A1) reverse([], []).
(A2) reverse([A|B], C) :- reverse(B, D), p(D, [A], C).
(Ax) p(D, [A], C) :- add_element(A, D, C).
(A3) add_element(X, [], [X]).
(A4) add_element(X, [H|T], [H|R]) :- add_element(X, T, R).
```

and

```
(NNF1) reverse([], []).
(NNF2) reverse([H|T], R) :- reverse(T, S), p(S, [H], R).
(NNFx) p(S, [H], R) :- append(S, [H], R).
(NNF3) append([], L, L).
(NNF4) append([H|T], X, [H|R]) :- append(T, X, R).
```

The newly added clauses ((Ax) and (NNFx)) each contain the only occurrence of the predicate p/n. Thus, the following transformation sequence is guaranteed to produce programs which are semantically equivalent to these original programs. We can normalize these programs with respect to each other by unfolding the new clause in each program (i.e., (Ax) and (NNFx)) with respect to their only subgoal (i.e., `add_element(A,D,C)` and `append(S,[H],R)`) producing the following programs:

```
(A1) reverse([], []).
(A2) reverse([A|B], C) :- reverse(B, D), p(D, [A], C).
(A3) add_element(X, [], [X]).
(A4) add_element(X, [H|T], [H|R]) :- add_element(X, T, R).
(A5) p([], [X], [X]).
(A6) p([H|T], [X], [H|R]) :- add_element(X, T, R).
```

and

```

(NNF1) reverse([], []).
(NNF2) reverse([H|T], R) :- reverse(T, S), p(S, [H], R).
(NNF3) append([], L, L).
(NNF4) append([H|T], X, [H|R]) :- append(T, X, R).
(NNF5) p([], [L], [L]).
(NNF6) p([H|T], [X], [H|R]) :- append(T, [X], R).

```

Folding clause (Ax) into clause (A6) and removing the no longer invoked `add_element/3` produces the following transformed target program:

```

(A1) reverse([], []).
(A2) reverse([A|B], C) :- reverse(B, D), p(D, [A], C).
(A5) p([], [X], [X]).
(A7) p([H|T], [X], [H|R]) :- p(T, [X], R).

```

Folding clause (NNFx) into clause (NNF6) and removing the no longer invoked `append/3` produces:

```

(NNF1) reverse([], []).
(NNF2) reverse([H|T], R) :- reverse(T, S), p(S, [H], R).
(NNF5) p([], [L], [L]).
(NNF7) p([H|T], [X], [H|R]) :- p(T, [X], R).

```

which is identical modulo variable renaming to the transformed target program. The important thing to note is that neither program retains its original form since the first program did not invoke `add_element/3` with its element `A` within a list structure and the "standard" program invokes the more general program `append/3` to append a single element onto the end of another list. It turns out that in this case it is possible to maintain the original form of the target program by using the following extended programs:

```

(A1) reverse([], []).
(A2) reverse([A|B], C) :- reverse(B, D), p(A, D, C).
(Ax) p(A, D, C) :- add_element(A, D, C).
(A3) add_element(X, [], [X]).
(A4) add_element(X, [H|T], [H|R]) :- add_element(X, T, R).

```

and

```

(NNF1) reverse([], []).
(NNF2) reverse([H|T], R) :- reverse(T, S), p(H, S, R).
(NNFx) p(H, S, R) :- append(S, [H], R).
(NNF3) append([], L, L).
(NNF4) append([H|T], X, [H|R]) :- append(T, X, R).

```

since the target program is using `add_element/3` which is not more general than necessary for the task. The problem is that `append/3` is actually a more general program than is necessary since the second argument can be a list of any length, but it is used in the specialized case of a single element list in the `reverse/2` program. Unfolding results in partial evaluation which has a specializing effect

which results in programs which have different forms.

Let's look at a slightly more complicated example which employs the use of different argument structures. Let's start with the following target implementation for `reverse/2`:

```
(B1) reverse(A,B) :- rs_reverse(A,[B]).
(B2) rs_reverse([], [X|X]).
(B3) rs_reverse([H|T],[R|X]) :- rs_reverse(T,[R,H|X]).
```

and see if this program is equivalent to the "standard" accumulator `reverse/2`:

```
(ANF1) reverse(L,R) :- reverse(L,[],R).
(ANF2) reverse([],L,L).
(ANF3) reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

First, the variables are bound ( $L \leftarrow A$  and  $R \leftarrow B$ ) and then each of the subgoals is replaced by a common subgoal producing the following extended programs:

```
(B1) reverse(A,B) :- p(A,[B]).
(Bx) p(A,[B]) :- rs_reverse(A,[B]).
(B2) rs_reverse([], [X|X]).
(B3) rs_reverse([H|T],[R|X]) :- rs_reverse(T,[R,H|X]).
```

and

```
(ANF1) reverse(L,R) :- p(L,[R]).
(ANFx) p(L,[R]) :- reverse(L,[],R).
(ANF2) reverse([],L,L).
(ANF3) reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

Now the non-general (i.e., non-variable) increasing arguments in clause (B<sub>x</sub>) and (ANF<sub>x</sub>) must be generalized producing the following extended programs:

```
(B1) reverse(A,B) :- p(A,[B]).
(Bx) p(A,[B|G]) :- rs_reverse(A,[B|G]).
(B2) rs_reverse([], [X|X]).
(B3) rs_reverse([H|T],[R|X]) :- rs_reverse(T,[R,H|X]).
```

and

```
(ANF1) reverse(L,R) :- p(L,[R]).
(ANFx) p(L,[R|G]) :- reverse(L,G,R).
(ANF2) reverse([],L,L).
(ANF3) reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

We can normalize these programs with respect to each other by unfolding the new clause of each program (i.e., (B<sub>x</sub>) and (ANF<sub>x</sub>)) with respect to their only subgoal (i.e., `rs_reverse(A,[B|G])` and `reverse(L,G,R)`) producing the following programs:

```

(B1) reverse(A,B) :- p(A,[B]).
(B2) rs_reverse([], [X|X]).
(B3) rs_reverse([H|T],[R|X]) :- rs_reverse(T,[R,H|X]).
(B4) p([], [L|L]).
(B5) p([H|T],[R|X]) :- rs_reverse(T,[R,H|X]).

```

and

```

(ANF1) reverse(L,R) :- p(L,[R]).
(ANF2) reverse([],L,L).
(ANF3) reverse([H|T],X,R) :- reverse(T,[H|X],R).
(ANF4) p([], [L|L]).
(ANF5) p([H|T],[R|X]) :- reverse(T,[H|X],R).

```

Folding clause (Bx) into clause (B5) and removing the no longer invoked `rs_reverse/2` produces the following transformed target program:

```

(B1) reverse(A,B) :- p(A,[B]).
(B4) p([], [L|L]).
(B6) p([H|T],[R|X]) :- p(T,[R,H|X]).

```

Folding clause (ANFx) into clause (ANF5) and removing the no longer invoked `reverse/3` produces:

```

(ANF1) reverse(L,R) :- p(L,[R]).
(ANF4) p([], [L|L]).
(ANF6) p([H|T],[R|X]) :- p(T,[R,H|X]).

```

which is identical modulo variable renaming to the transformed target program. Earlier in this section, `member/2` defined in terms of `append/3` was shown to be equivalent to the "standard" `member/2` definition. It is important to note that using the following slightly different program (where the `append` has been pulled out from the head and placed after the recursive call in clause (C3)) produces a different unfolded definition of `member/2`:

```

(C1) member(L,E) :- append(_, [E|_], L).
(C2) append([], L, L).
(C3) append([H|T], X, L) :- append(T, X, M), L=[H|M].

```

First of all, clause (C1) is unfolded with respect to its `append(_, [E|_], L)` subgoal producing the following program:

```

(C2) append([], L, L).
(C3) append([H|T], X, L) :- append(T, X, M), L=[H|M].
(C4) member([E|T], E).
(C5) member(L, E) :- append(_, [E|_], T), L=[H|T].

```

Folding clause (C1) into clause (C5) and removing the no longer invoked append/3 produces the following program:

```
(C4) member([E|T],E).
(C6) member(L,E) :- member(T,E),L=[H|T].
```

Note that this program acts quite differently from the previous version of member/2. While the previous version of member/2 executes as desired, this version of member/2 infinitely loops on the goal member([1],2). The problem stems from the ordering of the subgoals in clause (C6).

## 2.4.2 Termination of Logic Programs

The ordering of subgoals within a clause and clauses within a logic program is irrelevant given a fair computation rule. However, Prolog's computation rule is not fair. Thus, we cannot always switch the order of the subgoals within a clause and clauses within a program because some permutations of programs lead to programs that enter infinite loops on some queries. The calling mode of a predicate is vital in determining what permutations are loop-free.

Vasak and Potter (1986) define two forms of termination for logic programs: existential and universal termination. Existential termination identifies when a program either fails finitely or produces a successful derivation for a given query with a given computation rule and clause selection rule. Universal termination identifies when all the derivations for a given query of a program are finite. Universal termination, therefore, is only dependent on the computation rule and is independent of the clause selection rule. We will restrict our attention to universal termination. Thus, loop-free logic programs will refer to those logic programs which universally terminate.

The detection of loops in general logic programs is undecidable. However, several attempts have been made to prove termination properties about classes of logic programs. For example, van Gelder (1986) has shown that the class of general logic programs with the bounded term size property (i.e., the length of any literal in a derivation is bounded by some function of the length of the input) and freedom from recursive negation are universally terminating. However, it is undecidable in general whether a logic program has the bounded term size property or not.

Dynamic approaches to loop detection attempt to detect loops at execution time by augmenting the Prolog interpreter with simple loop checks. Bol (1991) has shown that such loop checks are possible for a restricted class of recursive Prolog programs which do not contain function symbols. In Naish's MU-Prolog (Naish, 1986), programs are augmented with wait-declarations which ensure that some input argument decreases and all the other input arguments do not increase before the recursive call. Plümer (1990) has shown that such structural recursive programs can be detected via static analysis and has shown that all structural recursive programs are universally terminating. The following definitions are taken from (Plümer, 1990). We will use the terms in(L) and out(L) to denote the sets of variables occurring on the input and output argument positions of a literal L. The term var(C) will denote the set of variables occurring in the clause C. A moded clause is one which has assigned modes (i.e., input or output) to the argument positions in each of its literals. A moded program is one which contains only moded clauses such that the same modes are assigned to each clause in the definition of p/n for each p/n

in the program.

**Definition.** A literal dependency graph for moded clause  $C = A :- B_1, \dots, B_n$  is a directed graph  $G(V,E)$  over the literals of  $C$  with arcs marked by sets of variables occurring in  $C$ . There is an arc  $\langle L, M \rangle$  in  $G(V,E)$  marked by  $W$  if  $L$  and  $M$  are different literals in  $C$  and the following hold:

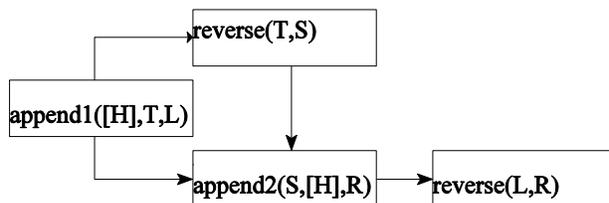
- if  $M \neq A$  and  $W \neq \{ \}$ , for  $W = (\text{out}(L) \cap \text{in}(M)) - \text{in}(A)$
- if  $M = A$  and  $W \neq \{ \}$ , for  $W = (\text{out}(L) \cap \text{out}(M)) - \text{in}(A)$

**Definition.** A literal dependency graph  $G(V,E)$  for moded clause  $C = A :- B_1, \dots, B_n$  is full if for every variable  $v \in \text{var}(C) - \text{in}(A)$  the following holds: there exists some  $i \in \{ 1, \dots, n \}$  and an arc  $\langle B_i, B_j \rangle$  marked with  $W$  containing  $v$  if  $v \in \text{in}(B_j) \cup \text{out}(A)$  for some  $j \in \{ 1, \dots, n \}$ .

Intuitively, a graph  $G(V,E)$  is full if no variable occurs at an input position of a literal in the body or at an output position of the head unless it appears somewhere else at an output position. Consider the following clause:

$$\text{reverse}(L,R) :- \text{append1}([H],T,L), \text{reverse}(T,S), \text{append2}(S,[H],R).$$

It has the following full literal dependency graph for modes  $\text{reverse}(+,-)$ ,  $\text{append1}(-,-,+)$ , and  $\text{append2}(+,+,-)$ :<sup>6</sup>



**Definition.** A moded program is well-moded if the following holds:

1. The literal dependency graph  $G(V,E)$  for every clause  $A :- B_1, \dots, B_n$  is full and defines a partial order  $<_{wf}$  on its nodes (i.e.,  $G(V,E)$  is acyclic).
2. If  $A$  is a unit clause and  $\sigma$  is a substitution such that for all  $v \in \text{in}(A)$ ,  $v\sigma$  is ground, then  $w\sigma$  is ground for  $w \in \text{out}(A)$ .
3. If  $B_i <_{wf} B_j$ , then  $i \leq j$  (i.e., the textual order of the literals in a clause is coherent with the partial order  $<_{wf}$ ).

---

<sup>6</sup>Note that  $\text{append1}/3$  and  $\text{append2}/3$  may refer to the same program, but are referenced here with unique names to reflect their different modes.

**Definition.** Let  $P$  be a well-moded logic program defining  $p/n$  such that  $P$  does not contain mutual recursion. Then  $P$  is structural recursive if there exists a subset  $I$  of the input positions such that for each clause  $p(t_1, \dots, t_n) :- \theta_1, p(u_1, \dots, u_n), \theta_2$  in  $P$ , the following condition holds. For all substitutions  $\sigma$  subject to the bindings produced by  $\theta_1$ , for each position  $i \in I$  it follows that  $t_B \sigma \leq_{wf} t_A \sigma$  and there exists a position  $j \in I$  such that  $t_D \sigma <_{wf} t_C \sigma$  where  $t_A$  is the term in position  $i$  in  $p(t_1, \dots, t_n)$ ,  $t_B$  is the term in position  $i$  in  $p(u_1, \dots, u_n)$ ,  $t_C$  is the term in position  $j$  in  $p(t_1, \dots, t_n)$ ,  $t_D$  is the term in position  $j$  in  $p(u_1, \dots, u_n)$ , and  $<_{wf}$  is some well-founded ordering of terms.

It should be noted that this definition for structural recursion differs slightly from Plümer's definition since it permits positions within arguments (i.e., complex terms within the argument). Thus, this definition includes the following non-standard definition of reverse/2:

```
reverse(L,R) :- ns_reverse([L],R).
ns_reverse([],X),X).
ns_reverse([H|T]|R],X) :- ns_reverse([T|[H|R]],X).
```

where the accumulator is actually combined with the input list. Note that each entire argument of `ns_reverse/2` remains the same size and thus `ns_reverse/2` is not structural recursive by Plümer's definition, but `ns_reverse/2` is structural recursive by the above definition since the first position of the first argument is a strictly decreasing argument.

In the remainder of the chapter, we provide some formal definitions for some common notions. Recall that we denote a set of subgoals with  $\theta$  and we use  $\vartheta \in \theta$  to represent a single subgoal in that set. As such, we can provide the following inductive definition:

**Definition.** A clause  $p(A_1, \dots, A_n) :- \theta$  is a terminating clause if it satisfies the following inductive definition:

$\theta$  contains only built-in predicates<sup>7</sup>

-or-

$\exists \vartheta \in \theta$  such that each clause in the unfolded clause set of  $p(A_1, \dots, A_n) :- \theta$  with respect to  $\vartheta$  is a terminating clause

**Definition.** A clause is a recursive clause if it is not a terminating clause.

**Definition.** A recursive clause  $p(A_1, \dots, A_n) :- \theta$  is indirectly recursive if  $\theta$  does not contain a subgoal of the form  $p(B_1, \dots, B_n)$ . We will refer to such clauses as containing indirect recursion.

**Definition.** A subgoal  $q(A_1, \dots, A_n)$  is a non-recursive subgoal if  $q^*(A_1, \dots, A_n) :- q(A_1, \dots, A_n)$  is a terminating clause.

---

<sup>7</sup>By built-in predicate, we mean those predicates that are part of the Prolog language (e.g., `=/2`, `is/2`, etc.).

**Definition.** A subgoal is a recursive subgoal if it is not a non-recursive subgoal.

**Definition.** A program is recursive if it contains a recursive clause.

**Definition.** A program is non-recursive if all its clauses are terminating clauses.

Consider the "bizarre" railway-shunt reverse/2 program given in Chapter 1:

```
(S1) reverse(L,R) :- rs_reverse(L,[R]).
(S2) rs_reverse([], [X|X]).
(S3) rs_reverse([H|T],S) :- rs_reverse(T,Y),insert(S,H,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).
```

Clause (S1) is a recursive clause since there is no sequence of unfolding steps that can reduce this clause to one containing only built-in predicates. Clause (S2), on the other hand, is a terminating clause since it contains no subgoals. Clause (S3) is another recursive clause. Note that the `rs_reverse(T,Y)` subgoal is a recursive subgoal. The `insert(S,H,Y)` subgoal, on the other hand, is a non-recursive subgoal. Let's look at the definition. In order to show that `insert(S,H,Y)` is non-recursive, we must show that the clause `insert*(S,H,Y) :- insert(S,H,Y)` is a terminating clause using the following program:

```
(Sa) insert*(S,H,Y) :- insert(S,H,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).
```

Unfolding clause (Sa) with respect to the `insert(S,H,Y)` subgoal produces:

```
(Sb) insert*([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).
```

Unfolding clause (Sb) with respect to the `enqueue([A],H,X)` subgoal produces:

```

(Sc) insert*([A|B],H,Y) :- enqueue([],H,X),append([A|X],B,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).

```

Unfolding clause (Sc) with respect to the `enqueue([],H,X)` subgoal produces:

```

(Sd) insert*([A|B],H,Y) :- append([A,H],B,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).

```

Unfolding clause (Sd) with respect to the `append([A,H],B,Y)` subgoal produces:

```

(Se) insert*([A|B],H,[A|Y]) :- append([H],B,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).

```

Unfolding clause (Se) with respect to the `append([H],B,Y)` subgoal produces:

```

(Sf) insert*([A|B],H,[A,H|Y]) :- append([],B,Y).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).

```

Finally, unfolding clause (Sf) with respect to the `append([],B,Y)` subgoal produces:

```

(Sg) insert*([A|B],H,[A,H|B]).
(S4) insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
(S5) enqueue([],E,[E]).
(S6) enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
(S7) append([],L,L).
(S8) append([H|T],L,[H|R]) :- append(T,L,R).

```

The only clause in the final unfolded clause set is clause (Sg). Since it is a terminating clause, the original subgoal `insert(S,H,Y)` is non-recursive. This completes our discussion of the fundamentals of logic programs. Additional definitions are provided as they are needed in Chapter 4.

## Chapter 3

# ADAPT Debugger

This chapter presents the ADAPT (Automated Debugger for an Adaptive Prolog Tutor) debugger. The field of automated program debugging is briefly reviewed and the main features of the existing program debuggers are described. Finally, ADAPT's approach to automated program debugging is detailed and compared to these "state of the art" debuggers.

### 3.1 Automated Program Debugging

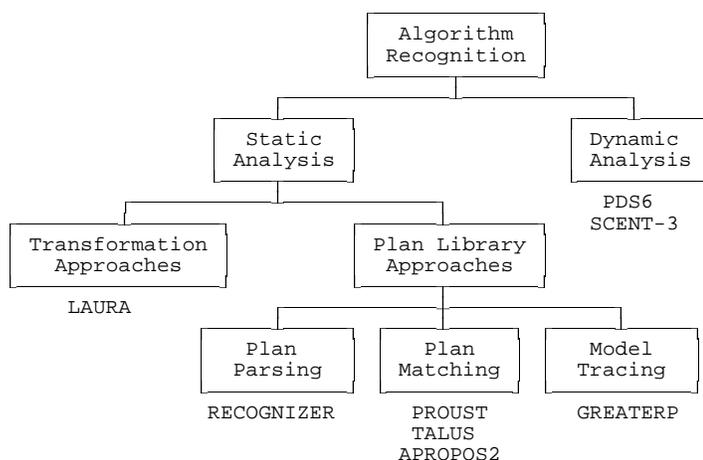
Automated program debugging involves recognizing the algorithm employed by the programmer and detecting any errors that exist in her program. Several attempts have been made to tackle either or both of these tasks for a number of programming languages, including machine language (Koffman & Blount, 1975), Basic (Barr, Beard, & Atkinson, 1976), Logo (Goldstein, 1975; Miller, 1979), Fortran (Adam & Laurent, 1980), Pascal (Soloway, Rubin, Woolf, Bonar, & Johnson, 1983; Johnson, 1986; Bonar & Cunningham, 1988), Lisp (Anderson & Reiser, 1985; Fischer, 1987; Murray, 1988; McCalla, Greer, & SCENT Team, 1988; Wills, 1990; Corbett & Anderson, in press; Reiser, Kimberg, Levitt, & Ranney, in press), and Prolog (Shapiro, 1983; Looi, 1988).

Although acknowledging the importance of the separation of the *algorithm recognition* task from the *bug detection* task, most of the literature (e.g., (Murray, 1988; Looi, 1988)) fails to take this distinction into consideration when comparing and contrasting the various approaches to automated program debugging. For most systems (e.g., LAURA (Adam & Laurent, 1980), the Lisp Tutor (Anderson & Reiser, 1985), and PROUST (Johnson, 1986)) this distinction is unnecessary. However, other systems (e.g., TALUS (Murray, 1988) and APROPOS2 (Looi, 1988)) take different approaches to algorithm recognition and bug detection.

#### 3.1.1 Algorithm Recognition

Algorithm recognition is the task of identifying the underlying algorithm employed by the student in solving the specified program. The major distinction between systems performing algorithm recognition is between those systems that perform their analysis *dynamically* (i.e., viewing the execution of the student's program) versus systems that perform their analysis *statically* (i.e., viewing the structure of the program without executing it) as shown in Figure 1. Systems that perform dynamic analysis include PDS6 (Shapiro, 1983) and SCENT-3 (McCalla, Greer, and

SCENT Team, 1988).<sup>8</sup> The systems which perform static analysis can be further divided into plan library approaches and transformation approaches.



**Figure 1.** Approaches to Algorithm Recognition

*Plan library* approaches decompose the problems into collections of well-defined operations. These collections of operations are stored in a library of plans. There are three major classes of plan library approaches. In *plan parsing* approaches (e.g., RECOGNIZER (Wills, 1990)), plans are represented by grammars and student programs are parsed in terms of the grammars. *Plan matching* approaches (e.g., PROUST, TALUS, and APROPOS2) represent plans using frame-like templates. The student's program is heuristically matched to appropriate templates in the library and the one with the "best" match is identified as the one used by the student. *Model tracing* approaches (e.g., the Lisp Tutor) avoid the task of algorithm recognition by modeling the entire programming process and constraining their students to conform to "expert" programming behavior. In model tracing, each action taken by the student is mapped to a set of production rules. Predefined sets of rules are classified as "ideal" rules which represent actions that a programming "expert" would take, while other rules are classified as "buggy" rules representing actions erroneously taken by "novice" programmers. By *tracing* the actions of the student with collections of these rules, model tracing approaches are able to build a *model* of the student and intervene whenever the student shows less than ideal behavior.

While plan library approaches analyze student programs by comparing them to a set of "correct" implementations, *transformation* approaches represent the "correct" implementation by a single normal form program and attempt to transform the student's program into the canonical form of the normal form program. The only system that uses this technique is the LAURA system for debugging Fortran programs. LAURA represents its normal form programs as graphs and transforms the student's Fortran program into this canonical graph form by using a set of

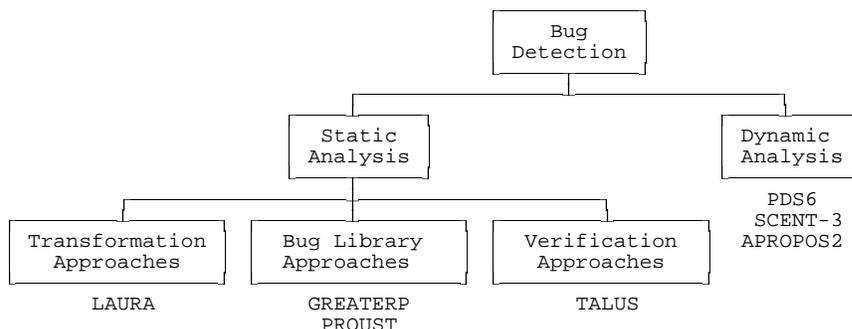
---

<sup>8</sup>Note that SCENT-3 actually performs some *static* analysis of program components with its "strategy judges." Pure dynamic analysis approaches really do not perform algorithm recognition since they are only concerned with *what* the program does rather than *how* the program does it.

equivalence-preserving transformation rules.

### 3.1.2 Bug Detection

Bug detection is the task of locating errors (or bugs) in the student's implementation of the algorithm identified in the algorithm recognition phase. As with algorithm recognition, the major distinction between systems performing bug detection is between *dynamic* and *static* approaches as shown in Figure 2. With the exception of APROPOS2, current automated program debugging systems use the same type of analysis for both algorithm recognition and bug detection. For example, APROPOS2 recognizes algorithms statically by matching them against a library of algorithms and then detects the bugs dynamically by executing the program on a set of I/O pairs associated with the algorithm having the best match.



**Figure 2.** Approaches to Bug Detection

The static approaches to bug detection are subdivided into three categories. *Bug library* approaches are the counterpart to plan library approaches for algorithm recognition. These systems (e.g., the Lisp Tutor and PROUST) have libraries of common bugs usually represented as sets of production rules which are used to compare segments of the student's program which do not match their corresponding segments in the "correct" implementation of the selected algorithm. *Program verification* approaches attempt to apply formal methods to prove the equivalence of corresponding segments of code that do not match (e.g., TALUS attempts to prove the equivalence of corresponding segments of code by invoking the Boyer-Moore Theorem Prover (Boyer & Moore, 1979) and detecting bugs via failed inductive proofs). Finally, *transformation* approaches (e.g., LAURA) highlight bugs by finding segments of code in the student's transformed program which do not match the corresponding segment of code in the normal form program.

## 3.2 Previous Approaches to Automated Debugging

ADAPT is an automated program debugger for Prolog which employs a transformational algorithm recognizer and uses a hierarchical schema-based bug library to detect bugs in incorrect programs. As such, it shares common threads with LAURA, the Lisp Tutor, PROUST, TALUS, and APROPOS2. Each of these systems is briefly reviewed in the following subsections. With the exception of the Lisp Tutor (which spawned several Ph.D. and post-doc research projects), each of these systems was developed as part of a Ph.D. research project. The interested reader is encouraged to read the appropriate dissertations for more details than are presented here.

### 3.2.1 LAURA

LAURA (Adam & Laurent, 1980) is an automated program debugger which attempts to pinpoint semantic errors in Fortran programs. It was developed to complement existing automatic program verification systems. Although such systems were useful in proving the correctness of a program, they were unable to give adequate information about the nature of the errors in an incorrect program.

LAURA's underlying program representation is a language-independent graph which represents the calculus process implied by the Fortran program. Thus, slight deviations in program implementation are eliminated. For example, the Fortran program:

```

      DO 1 I=1,N
1      IF (MAX .<. A(I)) MAX=A(I)
      ...

```

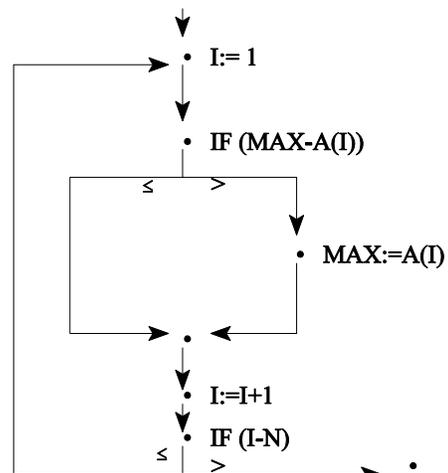
and the Fortran program:

```

      I=1
6      IF (MAX - A(I)) 4,5,5
5      I=I+1
      IF (I-N) 6,6,7
4      MAX=A(I)
      GOTO 5
7      ...

```

both have the same graph:



LAURA begins by standardizing the graph of the student's program. These standardization transformations actually serve to eliminate most of the gains in memory space and execution time, but they enable LAURA to make an accurate comparison of the student's program and the normal form program. Standardization transformations include variable separation, composition, and independent calculus separation. The variable separation transformation would transform the following code segment:

```

READ X
Y = f(X)
X = A + B
Z = g(X)

```

into:

```

READ X
Y = f(X)
WX = A + B
Z = g(WX)

```

eliminating the multiple independent use of the variable X. Composition would transform the code segment:

```

A(1) = f(B)
X = g(A(1))

```

into:

```

X = g(f(B))

```

which eliminates the use of intermediate variables whenever possible. Independent calculus separation transformation would transform the following code segment:

```

DO 1 I=1,N
  A(I) = X
  B(I) = Y
1

```

into:

```

DO 1 I=1,N
  A(I) = X
DO 2 I=1,N
  B(I) = Y
2

```

separating multiple independent processes within the same loop. Once the graph representing the student's Fortran program has been standardized, it is matched with the graph of the normal form program. Matching is a two step process. In the first step, LAURA assumes the student's program is correct and attempts to bind variables and transform portions of the graphs in order to make them

identical. This is done by maintaining a hypothesis list of possible node matches which is eventually emptied if the student's program is correct (or recognizable as correct).<sup>9</sup> In the second step, LAURA assumes the student's program is incorrect and repeats the processing of the first step using a set of correcting transformations which attempt to explain the differences (i.e., bugs) in the student's program.

### 3.2.2 Lisp Tutor

One of the shortcomings of LAURA is that the student may write a correct Fortran program for the problem using an approach that LAURA is incapable of understanding. One solution to this problem is to prevent the student from creating such a program. This is the approach taken by the Lisp Tutor (Anderson & Reiser, 1985). The major goal of the Lisp Tutor was to make the problem solving episodes more effective learning experiences. Anderson and his colleagues felt that students should do as much of the work as possible (i.e., they should learn by doing rather than by simply being told). They also felt that the tutor should provide immediate feedback to the student. Finally, they felt that the tutor should provide the student with the structure of the problem (i.e., provide the student with a template to fill in).

All of these goals are captured by the model tracing methodology which requires fine-grained modeling of the student's actions. Student actions are represented as production rules. There are two sets of rules: ideal rules and buggy rules. For example, an ideal rule for the merging of two lists would be:

```
IF the goal is to combine LIST1 and LIST2 into a single list,
   and LIST1 and LIST2 are both of type list,
THEN use the function APPEND,
   and set subgoals to code LIST1 and LIST2.
```

whereas a buggy rule for the merging of two lists might be:

```
IF the goal is to combine LIST1 and LIST2 into a single list,
   and LIST1 and LIST2 are both of type list,
THEN use the function LIST,
   and set subgoals to code LIST1 and LIST2.
```

The Lisp Tutor consists of several hundred of these ideal and buggy rules. Model tracing enables the Lisp Tutor to diagnose very specific misconceptions and provide immediate feedback to the student, but it has the severe limitation that its students are not permitted any freedom in their problem solving approach.

---

<sup>9</sup>If the student takes a totally different approach to the problem producing a correct Fortran program which has a graph which cannot be transformed into the normal form graph, then LAURA will incorrectly infer that the student's program is incorrect.

### 3.2.3 PROUST

PROUST (Johnson, 1986) is an automated program debugger for Pascal that implements intention-based diagnosis. Rather than forcing a student to maintain the same intentions as "expert" programmers by strict monitoring of the student's actions as with model tracing in the Lisp Tutor (Anderson & Reiser, 1985), intention-based diagnosis attempts to infer the student's intentions by statically analyzing their completed Pascal programs. PROUST diagnoses bugs by inferring the student's intentions and the realizations of these intentions in her Pascal program. Intention-based diagnosis is necessary because "accurate bug identification must be based upon analysis of the programmer's intentions, i.e., the intended function of the program (goals) and the intended implementation of this function (plans)" (Johnson, 1986, p. 65). Furthermore, "any method which focuses on the actual structure and/or function of the program, rather than the intended structure and function, will not work as well on larger or buggier novice programs" (Johnson, 1986, p. 65).

The basic processing of PROUST is as follows. PROUST begins by selecting a goal to be analyzed. Then it retrieves the set of associated plans from the plan database and attempts to match each plan against the goal. Because plans may generate subgoals, this is a recursive process. If the plans fail to match then plan-difference rules are applied to explain the differences.

The instructor provides a problem description for each program assignment. Ideally, the instructor would input the problem description in English (or some other natural language). However, the current implementation of PROUST requires the instructor to input the problem description into an informal specification language which represents each proposition in the English problem statement as either a goal which must be satisfied or a data object that the program must manipulate. Only explicitly stated goals and data objects are included in the problem description. PROUST infers implicit goals (e.g., boundary condition checks) and data objects (e.g., loop counters). Data objects are assigned explicit properties from the problem statement and are associated with an object class which makes it possible for objects to inherit a variety of properties that are not explicitly indicated. For example, the following English problem description:

```
Noah needs to keep track of rainfall in the New Haven area in order
to determine when to launch his ark. Write a Pascal program that will
help him do this. The program should prompt the user to input numbers
from the terminal; each input stands for the amount of rainfall in
New Haven for a day. Note: since rainfall cannot be negative, the
program should reject negative input. Your program should compute the
following statistics from this data:
```

1. the average rainfall per day;
2. the number of rainy days;
3. the number of valid inputs (excluding any invalid data that
 might have been read in);
4. the maximum amount of rain that fell on any one day.

```
The program should read data until the user types 99999; this is the
sentinel value signaling the end of the input. Do not include 99999
in the calculations. Assume that if the input value is non-negative,
and not equal to 99999, then it is valid input data.
```

would be represented by PROUST as:

```

DefProgram Rainfall;
DefObject ?Rainfall:DailyRain ObjectClass ScalarMeasurement;
DefGoal Sentinel-Controlled Input Sequence(?Rainfall:DailyRain, 99999);
DefGoal Loop Input Validation(?Rainfall:DailyRain, ?Rainfall:DailyRain<0);
DefGoal Output(Average(?Rainfall:DailyRain));
DefGoal Count(Average(?Rainfall:DailyRain));
DefGoal Output(Guarded Count(?Rainfall:DailyRain ?Rainfall:DailyRain>0));
DefGoal Output(Maximum(?Rainfall:DailyRain));

```

PROUST maintains a library of *plans*. A plan is a series of steps a programmer follows in solving a problem. Plans are used to map the programmer's intended goals into Pascal code. There are two types of information contained in a plan: set of properties and assertions about the plan and the *plan template*. The properties and assertions about the plan serve two purposes: they determine when a plan can be used and what implications such use may have and they indicate the role each component of the plan plays. These include preconditions and postconditions as well as exception conditions. The plan template can be either a Pascal statement, a subgoal which in turn is implemented as one or more Pascal statements, or a reference to a component of another plan. An example plan is the following Average Plan:

**Variables:**

?Avg, ?Sum, ?Count

**Posterior goals:**

Count(?New, ?Count)

Sum(?New, ?Sum)

Guard Exception(Update: component of goal Average,  
(?Count in goal Count)= 0)

**Exception condition:**

(?Count in goal Count) = 0)

**Template:**

(Component Mainloop:) of goal Read & Process)

Update: ?Avg := (?Sum / ?Count)

An example of a plan-difference rule is the WHILE-for-IF plan-difference rule which could apply to any plan that has conditional statements within loops:

```

IF a while statement is found in place of an if statment,
AND the while statement appears inside another loop,
THEN the bug is a WHILE-for-IF bug,
    probably caused by a confusion about the control flow of
embedded loops.

```

This plan-difference rule would apply to the following code segment:

```

repeat
  ...
  while Rain<0 do
    begin
      ...
    end;
  ...
until Rain=99999;

```

where the student incorrectly coded `while Rain<0 do` in place of `if Rain<0 do`.

One of the major limitations of PROUST's intention-based diagnosis is its reliance on the explicit representation of all possible implementations of a given problem assignment. Two systems were developed to overcome this limitation: TALUS (Murray, 1988) and APROPOS2 (Looi, 1988). Both systems perform an A\* search of the possible algorithms for the assigned problem to find the best matched algorithm for the student's implementation. However, they are not dependent on the explicit representation of every possible implementation of the student's goals and subgoals. If PROUST is unable to accurately infer the algorithm the student is employing, it often provides vague, missing, or incorrect bug diagnoses of the code which sometimes lead to *false alarms* which occur when correct code is interpreted as buggy. By invoking the Boyer-Moore theorem prover (Boyer & Moore, 1979) when it is unable to match a plan to the student's active goal, TALUS is able to extend the class of Lisp programs recognized by the debugger. In a similar fashion, APROPOS2 invokes Shapiro's algorithmic debugger (Shapiro, 1983) to analyze Prolog subgoals that do not match.

### 3.2.4 TALUS

TALUS (Murray, 1988) is an intention-based debugger that uses plan libraries to recognize the algorithm employed by the student. Its algorithm recognition approach employs an A\* search. It differs from PROUST in its approach to bug detection. Unlike PROUST which uses bug libraries in the detection of bugs, TALUS employs a more formal reasoning process to recognize logically equivalent predicates. By evaluating conjectures in the Boyer-Moore logic (Boyer & Moore, 1979), TALUS is able to reason about programming language semantics to establish properties other than equivalence for operators and constructs that occur in student programs. For example, the conjecture:

```

(IMPLIES (AND (NUMBERP X) (NUMBERP Y))
         (EQUAL (PLUS X Y) (PLUS Y X)))

```

is a well-formed formula in the Boyer-Moore logic that asserts the the function PLUS is commutative given that its arguments are both numbers. By proving that this conjecture is a theorem using the Boyer-Moore theorem prover, TALUS can establish that commutativity holds for the function PLUS and that the code fragments (PLUS X Y) and (PLUS Y X) are computationally equivalent. This approach can also be used to prove that a code fragment from the student's program is equivalent to a code fragment from the reference program.

TALUS breaks the debugging task down into four steps. In the first step, the student and reference programs are simplified into a form that is more suitable for analysis. The second step, algorithm recognition, matches the frames representing the simplified functions of the student and reference programs. The third step, bug detection, requires splitting the functions into cases which is needed for an inductive proof of equivalence. Finally, in the last step violated verification conditions are repaired by altering the student function.

Student and reference programs are converted to IF normal form. IF normal form has the following conditions:

1. the only conditional expression is IF
2. no IF expression occurs as part of the test of another IF expression
3. no IF expression occurs inside a function call

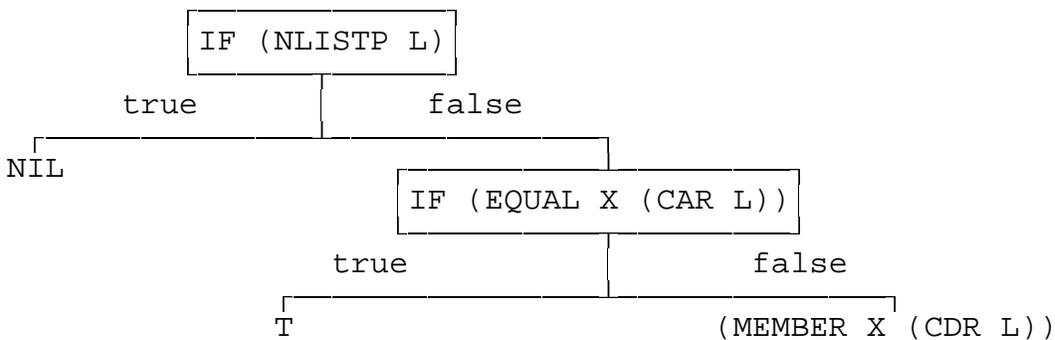
Functions in IF normal form can be represented as binary trees where the nonterminal nodes of the tree are conditional tests and the terminal nodes are function terminations or recursions. For example, the following program for MEMBER:

```
(DEFUN MEMBER (X L)
  (COND ((NLISTP L) NIL)
        ((EQUAL X (CAR L)) T)
        (T (MEMBER X (CDR L)))))
```

has IF normal form of:

```
(DEFUN MEMBER (X L)
  (IF (NLISTP L)
      NIL
      (IF (EQUAL X (CAR L))
          T
          (MEMBER X (CDR L)))))
```

which can be represented by the following binary tree:



The second step in TALUS's debugging process is algorithm recognition. TALUS performs an A\* search to find the best match between the student's program and the set of reference programs.

TALUS represents programs as E-frames. The E-frame for TALUS's reference MEMBER program is:

```

FUNCTION-NAME:      MEMBER
FORMALS:           ( ITEM BAG )
DEFINITION:       ( DEFUN MEMBER ( ITEM BAG )
                   ( IF ( NLISTP BAG )
                       NIL
                       ( IF ( EQUAL ITEM ( CAR BAG )
                           T
                           ( MEMBER ITEM ( CDR BAG )
                               ) ) ) ) ) )
NORMALIZED-CODE:  ( IF ( NLISTP BAG )
                   NIL
                   ( IF ( EQUAL ITEM ( CAR BAG )
                       T
                       ( MEMBER ITEM ( CDR BAG )
                           ) ) ) ) )
TERMINATION-FORMALS: ( ITEM BAG )
OUTPUT-FORMALS:    NIL
VARIABLE-DATA-TYPESS: ( ( ITEM )
                        ( BAG CONS ) )
OUTPUT-DATA-TYPE:  ( BOOLEAN )
CONDITIONS:        ( ( ( NLISTP ) ( BAG ) )
                    ( ( EQUAL ) ( ITEM BAG ) ) ) )
TERMINATIONS:      ( ( ( ( NOT ( NLISTP BAG ) )
                        ( EQUAL ITEM ( CAR BAG ) ) ) ) )
                    ( ( ( NLISTP BAG ) ) NIL ) ) )
RECURSIONS:        ( ( ( ( NOT ( NLISTP BAG ) )
                        ( NOT ( EQUAL ITEM ( CAR BAG ) ) ) ) ) )
                    ( MEMBER ITEM ( CDR BAG ) ) ) ) )
CONSTRUCTIONS:     ( ( ( ( NOT ( NLISTP BAG ) )
                        ( NOT ( EQUAL ITEM ( CAR BAG ) ) ) ) ) )
                    ( MEMBER ITEM ( CDR BAG ) ) ) ) )
VARIABLE-UPDATES:  ( ( ITEM
                    ( ( ( NOT ( NLISTP BAG ) )
                        ( NOT ( EQUAL ITEM ( CAR BAG ) ) ) ) )
                    ) )
                    ITEM ) )
                    ( BAG
                    ( ( ( NOT ( NLISTP BAG ) )
                        ( NOT ( EQUAL ITEM ( CAR BAG ) ) ) ) )
                    ) )
                    ( CDR BAG ) ) ) )
RECURSION-TYPE:    ( LIST-RECURSION )
FUNCTION-TYPE:      RECURSIVE
FUNCTION-ROLE:      MAIN
CONSTRUCTORS-CALLED: NIL
PREDICATES-CALLED:  NIL
FUNCTIONS-CALLED:   ( MEMTREE )
SIDE-EFFECTORS:     NIL

```

```
DB-FETCH-FNS :      NIL
PROGNS :          0
```

TALUS's E-frame representation facilitates "a robust algorithm representation process by allowing partial matching to occur on the *semantic* features of abstract enumerations and the role of functions in solving tasks, rather than on code structure" (Murray, 1988, pp. 75-76). In selecting the best algorithm match, TALUS produces a mapping between functions in the student's program to functions in the reference program.

This mapping is used in the bug detection step of the debugging process. Since the student and reference program have been converted into IF normal form, case splitting enables TALUS to detect missing and extra conditionals. After correcting any missing or extra conditionals, TALUS uses the verification template:

```
(IMPLIES (common condition)
         (EQUAL (student code fragment) (reference code fragment)))
```

to prove that the student's program segment is equivalent to the closest matching reference program segment. If the Boyer-Moore theorem prover is unable to establish that the conjecture is a theorem, then TALUS attempts to correct the student's program by patching the inductive proof.

### 3.2.5 APROPOS2

APROPOS2 (Looi, 1988) is another intention-based debugger that uses plan libraries to recognize the algorithm employed by the student. Like TALUS, it was designed to overcome PROUST's susceptibility to false alarms. Unlike TALUS, it employs program testing rather than program verification to resolve program mismatches. Program testing is effective in uncovering bugs, but testing alone cannot ensure that a program is free of bugs. Thus, APROPOS2 may incorrectly assume a student's program is correct when it contains bugs.

A reference P-frame contains abstract computational features of a Prolog program. The P-frame for APROPOS2's reference naive reverse/2 program is:

```
P-FRAME:
TASK NAME: reverse/2
ALGORITHM NAME: naive_reverse
LIKELY PREDICATE NAMES: reverse,rev,rv
NUMBER OF ARGUMENTS: 2
INVOCATION TYPE OF PREDICATE: both arguments are lists
INVOCATION MODE OF PREDICATE: reverse(+,-)
RECURSION ARGUMENT: list iteration in the 1st argument
PROGRAMMING TECHNIQUES: naive-recursion
NUMBER OF CLAUSES: 2
CLAUSE 1:
  TYPE: base
  HEAD GOAL: reverse([],[])
  PREFIX SUBGOALS: nil
  RECURSIVE SUBGOALS: nil
```

SUFFIX SUBGOALS: nil  
 COMMENTARY: "This base case says that the reverse of the empty list  
 list is the empty list."

CLAUSE 2:

TYPE: recursive  
 HEAD GOAL: reverse([H|T],Res)  
 PREFIX SUBGOALS: nil  
 RECURSIVE SUBGOALS: reverse(T,Sofar)  
 SUFFIX SUBGOALS: append(Sofar,[H],Res)  
 COMMENTARY: "This recursive case says that the reverse of a non-empty  
 list can be found by reversing the tail of the list and  
 then appending a list consisting only of the first  
 element of the original to the end of the reverse of the  
 tail."

CLAUSE 2:

TYPE: recursive  
 HEAD GOAL: reverse([H|T],Res)  
 PREFIX SUBGOALS: append(Sofar,[H],Res)  
 RECURSIVE SUBGOALS: reverse(T,Sofar)  
 SUFFIX SUBGOALS: nil  
 COMMENTARY: "This recursive case says that the reverse of a non-empty  
 list can be found by reversing the tail of the list and  
 appending a list consisting only of the first element  
 of the original to the end of the reverse of the tail.  
 However, you should put the ',append/3,' subgoal after  
 the recursive subgoal for efficiency reasons. If it  
 comes before the recursive subgoal, then ',Sofar,' and  
 ',Res,' will be uninstantiated variables when this goal  
 is attempted. It will try many values for ',Sofar,' and  
 ',Res,' before it gets the right one."

BUGGY-CLAUSE 2:

TYPE: recursive  
 HEAD GOAL: reverse([H|T],Res)  
 PREFIX SUBGOALS: nil  
 RECURSIVE SUBGOALS: reverse(T,Sofar)  
 SUFFIX SUBGOALS: append(Sofar,H,Res)  
 COMMENTARY: "This recursive case says that the reverse of a non-empty  
 list can be found by reversing the tail of the list and  
 then appending a list consisting only of the first  
 element of the original to the end of the reverse of the  
 tail. For the append goal to work, it needs arguments  
 which are of type list. In this instance, H is an  
 element which is to be tagged to the end of another list  
 ',Sofar,'. So we need to write append(Sofar,[H],Res)."

CLAUSE-ORDERING: no constraint

TEST-CASES:

```
[reverse([],Res),Res=[]]
[reverse([a,b],Res),Res=[b,a]]
[reverse([a,b,c,d,e],Res),Res=[e,d,c,b,a]]
[append([],[a],Res),Res=[a]]
[append([b],[a],Res),Res=[b,a]]
[append([e,d,c,b],[a],Res),Res=[e,d,c,b,a]]
```

APROPOS2 begins its analysis by creating a P-frame for the student's program. This is done by applying program transformations which are simply one step unfolding. Given the type and mode of each of the main predicate's arguments, APROPOS2 infers the type and mode of each of

the remaining arguments. Basic dataflow analysis is performed to identify singleton variables (i.e., variables which occur only once in a clause) and obvious looping structures. Associated with this analysis is misspelling analysis which attempts to infer typographical errors that the student may have made which lead to singleton variables. Finally, APROPOS2 attempts to detect any programming techniques (Brna, Bundy, Dodd, Eisenstadt, Looi, Pain, Robertson, Smith, & van Someren, 1991) that the student has employed.

Once the student's Prolog program has been converted into a P-frame, it is compared to APROPOS2's database of reference P-frames using an A\* search. This search is performed top-down:

1. match predicate definitions
2. match clauses
3. match subgoals
4. match arguments
5. match terms

Then the Code Critique Phase is entered where an attempt is made to determine whether mismatched components of the student's program are correct or not. The first step is permuting the order of the arguments in the student's subgoals. If this is unsuccessful in producing a match with the reference program then dynamic analysis is performed. For dynamic analysis, APROPOS2 runs the student program on each of TEST-CASES in the reference P-frame and assumes the student's program is correct if it produces the correct bindings for each of the test cases. If at least one of the test cases fails then APROPOS2 assumes that the clauses in the student's program are not in the correct order so the clauses in the student's program are permuted. Finally, if the programs still fail to match then APROPOS2 assumes the student's program is incorrect.

### **3.3 ADAPT's Approach to Automated Program Debugging**

The schema-based Prolog tutor was originally conceived from the feeling that the ideal learning environment for students learning to program (or perform any problem solving task) is one in which they are initially given problems to solve with almost all the structure provided and then incrementally given problems to solve with less and less structure provided (Gegg-Harrison, 1991). Unlike the Lisp Tutor, where the template (or schema) is provided in order to permit the student to "focus on the more conceptually difficult aspects of Lisp" (Anderson & Reiser, 1985, p. 162), our tutor provides templates with the explicit goal of teaching problem solving techniques.

In addition to its instructional role, schemata can also be exploited in the evaluation (or program debugging) process. ADAPT makes use of program schemata in both the algorithm recognition and the bug detection phases of the program debugging task. By using normal form programs as the basis of its algorithm recognition phase, ADAPT is able to recognize a large precisely defined class of Prolog programs without the need for a large plan library. By separating bug types into three categories and hierarchically arranging schema-level bugs, ADAPT provides enhanced bug detection over traditional bug library approaches to bug detection. The current version of the ADAPT debugger and the associated schema-based Prolog tutor is written in Prolog.

It consists of approximately 5000 lines of code. The basic algorithm employed by ADAPT is given below and examples are given in the following subsections. In the following description, "NF" represents the dynamic normal form program (i.e., it is like a program variable in an imperative programming language) which is potentially changed in each step.

### **ADAPT Debugging Algorithm<sup>10</sup>**

1. Find the main predicate:
  - . ensure program is structural recursive (i.e., loop-free)
  - . identify missing/extra predicate definitions and suggest potential typos (when applicable)
2. Eliminate indirect recursion in target program where possible by unfolding/folding.
3. If target program still has an indirectly recursive clause then apply accumulator transformation to NF.
4. If target program decomposes the input from the back then apply back processing transformation to NF.
5. If target program has more base case clauses than NF then unfold recursive step clauses in NF until both programs have the same number of base case clauses.
6. If target program removes more than one element in its recursive clause then unfold the NF's recursive call subgoal with respect to either the original NF or the back processing NF (i.e., program produced by applying the back processing transformation to the original NF).
7. Apply subgoal permutation, clause permutation, and the list constructor transformations to NF if necessary.
8. Apply subgoal matching to decompose and compose subgoal components of target program and NF (including associativity rewrite rule).
9. Apply appropriate bug rules:
  - a. .Apply all bug rules associated with the assigned program and all of its ancestors in the schema hierarchy

---

<sup>10</sup>The accumulator, back processing, clause permutation, subgoal permutation, and list constructor transformations are formally defined in Chapter 4. The rationale behind the order of the transformation application is also given in Chapter 4.

- b. Apply all bug rules associated with any techniques used (e.g., accumulated composition of the output or backward decomposition of the input)

### 3.3.1 ADAPT's Approach to Algorithm Recognition

ADAPT's algorithm recognizer begins with a single normal form Prolog program (e.g., "naive" reverse/2), generates a set of representative programs, and then transforms the most appropriate one into a structure that best matches the student's program. ADAPT begins its analysis by finding the student's main predicate. The main predicate is the one which is not an auxiliary to any of the other predicates. Because ADAPT finds the main predicate rather than searching for common names, it is able to recognize the following program (variation #94 in Appendix B):

```
reverse([],L,L).
reverse([H|T],X,[H|L]) :- reverse(T,X,L).
append([],[]).
append([H|T],L) :- append(T,M),reverse(M,[H],L).
```

where the predicate names "reverse" and "append" have been switched. While searching for the main predicate, ADAPT also finds missing and extra predicates. Extra predicates are predicates that have definitions, but do not appear in the body of any other predicate. Note that the main predicate appears to ADAPT as an extra predicate. So ADAPT identifies a predicate as extra only if there is more than one potential main predicate. Often times extra predicate definitions occur in programs that also contain missing predicate definitions, resulting from the student making a typographical error either in the predicate name or the number of arguments in the subgoal invocation. For example, in the following program (variation #24 in Appendix B):

```
reverse([],[]).
reverse([H|T],Res) :- reverse(T,Sofar),append(Sofar,H,Res).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append([L1,L2,L3]).
```

the student has combined the arguments in the append([L1,L2,L3]) subgoal into a single argument. It is also possible for a student to simply omit a predicate definition. For example, in the following program (variation #21 in Appendix B):

```
reverse([],[]).
reverse([H|T],Res) :- reverse(T,Sofar),append(Sofar,H,Res).
```

the student has omitted the definition for append/3. It is important to note that unlike APROPOS2 which assumes that the student's subgoal append(Sofar,H,Res) is incorrect since the normal form program has subgoal append(Sofar,[H],Res), ADAPT simply asks the student to enter a definition for append/3 enabling the student to enter:

```
append([],A,[A]).
```

```
append([A|B],C,[A|D]) :- append(B,C,D).
```

which combines with the previous two clauses to form a correct program.

In addition to locating missing/extra predicates while searching for the main predicate, ADAPT also detects infinite loops. It does this by forcing the student's program to be structural recursive (Plümer, 1990). ADAPT can detect the infinite loop in the following program (variation #77 in Appendix B):

```
reverse(L,R) :- reversex(L,R).
reverse([],X-X).
reverse([H|T],Y-X) :- Z=[H|X],reverse(T,Y-Z).
reversex(A,B) :- reverse(A,[],B).
reverse(List,Accum,Result) :- reverse(List,Result-Accum).
```

which has mutually recursive predicates defined since the student intended the first clause to be an initializing clause for the actual recursive program defined in the second and third clauses. A special case of this mutual recursion without the indirection is highlighted in the following program (variation #78 in Appendix B):

```
reverse(L,R) :- reverse(L,R-[]).
reverse([],X-X).
reverse([H|T],Y-X) :- Z=[H|X],reverse(T,Y-Z).
```

where the infinite looping behavior is more apparent. A more standard example of an infinite loop is in the following non-structural recursive program (variation #18 in Appendix B):

```
app([],L,L).
app([H|L1],L2,[H|L3]) :- app(L1,L2,L3).
reverse([],[]).
reverse([H|T],Res) :- app(Sofar,[H],Res),reverse(T,Sofar).
```

which loops since the first and last arguments in the `app(Sofar,[H],Res)` are unbound (i.e., `app/3` is not structural recursive for mode `app(-,+,-)`).<sup>11</sup>

Once ADAPT has located the main predicate, it attempts to create the best normal form to compare to the student's program. Each input list can be decomposed from either the front or the back. Likewise, each output can be composed from either the front or the back. There are two types of programs that the student might produce: programs that remove a single element on each pass and programs that remove more than one element on each pass.<sup>12</sup>

If the student's program removes a single element on each pass then ADAPT merely compares the directionality of the output composition in the student's program to the normal form

---

<sup>11</sup>ADAPT defines the notion of loop-free in terms of universal termination. Specifically, ADAPT says that a program is correct only if it is structural recursive.

<sup>12</sup>ADAPT's design permits it to handle program which remove more than one element each pass (i.e., hybrid programs). However, they are not included in the Characterization Theorem given in the next chapter.

program. If they are different then ADAPT applies the accumulator transformation to the normal form program. For example, ADAPT is able to recognize the following program (variation #8 in Appendix B):

```
reverse(A,B) :- rev(A,[B]).
rev([], [B|B]).
rev([Head|Tail],[B|Temp]) :- rev(Tail,[B,Head|Temp]).
```

by transforming the naive reverse/2 normal form into its accumulator representative program:

```
reverse(L,R) :- reverse(L,[],R).
reverse([],L,L).
reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

Now ADAPT must attempt to transform the normal form program into the same form (i.e., same subgoal ordering, same argument ordering, etc.) as the student's program. For this particular program, that requires ADAPT to transform the initializing clause `reverse(L,R) :- reverse(L,[],R)` into one that imbeds its accumulator in the second argument position. This is done by replacing the original initializing clause with a new initializing clause `trans_reverse(L,R) :- reverse(L,[R])` and a reordering clause `reverse(L,[R]) :- reverse(L,[],R)` producing:

```
trans_reverse(L,R) :- reverse(L,[R]).
reverse(L,[R]) :- reverse(L,[],R).
reverse([],L,L).
reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

Next the accumulator is located and generalized in the original initializing clause `trans_reverse(L,R) :- reverse(L,[R])`:

```
trans_reverse(L,R) :- reverse(L,[R]).
reverse(L,[R|G]) :- reverse(L,G,R).
reverse([],L,L).
reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

Now ADAPT can unfold the `reverse(L,G,R)` in the clause `reverse(L,[R|G]) :- reverse(L,G,R)` producing:

```
trans_reverse(L,R) :- reverse(L,[R]).
reverse([], [L|L]).
reverse([H|T],[X|R]) :- reverse(T,[H|X],R).
```

Folding the clause `reverse(L,[R|G]) :- reverse(L,G,R)` into the newly generated recursive clause `reverse([H|T],[X|R]) :- reverse(T,[H|X],R)` produces the desired program:

```
trans_reverse(L,R) :- reverse(L,[R]).
reverse([], [L|L]).
reverse([H|T],[X|R]) :- reverse(T,[X,H|R]).
```

which is identical to the student's program modulo variable and predicate renaming. The ADAPT merely compares the directionality of the input list decomposition in the student's program to the normal form program. If they are different then ADAPT applies the back processing transformation to the normal form program. For example, ADAPT is able to recognize the following program (variation #9 in Appendix B with the missing definition for `append/3` added):

```
reverse([], []).
reverse(L, [H|RT]) :- append(T, [H], L), reverse(T, RT).
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

by transforming the naive `reverse/2` normal form into its back processing representative program (which happens to be identical modulo variable renaming to this student's program).

The accumulator and back processing transformations enable ADAPT to recognize a large class of programs with a single normal form, however, ADAPT's robustness in algorithm recognition is due to its ability to reduce programs via unfolding. One use of unfolding by ADAPT is in the recognition of programs for tasks that permit implementations that remove multiple elements on each pass. If the student's program removes more than one element on each pass then ADAPT must create a hybrid normal form.

ADAPT creates hybrid normal forms via unfolding. There are two basic forms of hybrid normal forms. One set of hybrid programs merely processes more than one input from the same end of the input list. For example, the following accumulator implementation of `reverse/2`:

```
reverse(L, R) :- reverse(L, [], R).
reverse([], L, L).
reverse([X], L, [X|L]).
reverse([H, I|T], X, R) :- reverse(T, [I, H|X], R).
```

processes two elements from the front of the input list on each pass. This program was produced from the standard accumulator implementation of `reverse/2` by merely unfolding the `reverse(T, [H|X], R)` subgoal in the recursive clause. Note that this also works for programs which remove their elements from the back of the input list. For example, ADAPT can transform the following inverse naive implementation of `reverse/2`:

```
reverse([], []).
reverse(L, [H|R]) :- append(T, [H], L), reverse(T, R).
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

into a program that removes two elements from the back of the input list by unfolding the `reverse(T, R)` subgoal in the recursive clause producing the following program:

```

reverse([], []).
reverse([X],[X]).
reverse(L,[I,H|R]) :- append(T,[H,I],L),reverse(T,R).
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

```

Merely unfolding the `reverse(T,R)` subgoal actually does not produce this program. The key to producing this program comes from the fact that `append/3` is associative (i.e., the pair of subgoals `append(A,B,X),append(X,C,D)` produces the same answer for `D` as the pair of subgoals `append(B,C,Y),append(A,Y,D)`). Unfolding the `reverse(T,R)` subgoal actually produces the following program:

```

reverse([], []).
reverse([X],[X]).
reverse(L,[I,H|R]) :- append(M,[H],L),append(T,[I],M),reverse(T,R).
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

```

Since `append/3` is associative, we can rewrite `append(M,[H],L),append(T,[I],M)` as `append([H],[I],X),append(T,X,L)` producing following program:

```

reverse([], []).
reverse([X],[X]).
reverse(L,[I,H|R]) :- append([H],[I],X),append(T,X,L),reverse(T,R).
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

```

Unfolding `append([H],[I],X)` produces following program:

```

reverse([], []).
reverse([X],[X]).
reverse(L,[I,H|R]) :- append([], [I], Y), append(T, [H|Y], L), reverse(T, R).
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).

```

Finally, the desired program is obtained by unfolding `append([], [I], Y)` producing the following program:

```

reverse([], []).
reverse([X],[X]).
reverse(L,[I,H|R]) :- append(T,[H,I],L),reverse(T,R).
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

```

The other set of hybrid programs processes the input list from both ends. We have already seen examples of this with `switch_reverse/2`. Let's look at this process in detail. We start with both the naive and the inverse naive definitions for `reverse/2`:

```

(A1) reverse([], []).
(A2) reverse([H|T], L) :- reverse(T, M), append(M, [H], L).
(A3) append([], L, L).
(A4) append([H|T], L, [H|R]) :- append(T, L, R).

(B1) reverse([], []).
(B2) reverse(L, [H|R]) :- append(T, [H], L), reverse(T, R).
(B3) append([], L, L).
(B4) append([H|T], L, [H|R]) :- append(T, L, R).

```

We take the recursive clause in the first definition (A2) and unfold its `reverse(T,M)` subgoal with respect to the second definition (B1) - (B4) producing the following program:

```

reverse([], []).
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
reverse([X], [X]).
reverse([H|T], [G|L]) :- append(X, [G], T), reverse(X, M), append(M, [H], L).

```

when the no longer needed second reverse/2 definition is removed.

The most predominant use of unfolding by ADAPT is in the comparison of subgoals with arguments that are not in the same order or in different structures. For example, ADAPT is able to recognize the following program (variation #21 in Appendix B with the missing definition for `append/3` added):

```

reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, H, Res).
append([], E, [E]).
append([H|T], E, [H|R]) :- append(T, E, R).

```

The best normal form implementation (which in this case is naive reverse/2) is selected:

```

reverse([], []).
reverse([H|T], R) :- reverse(T, S), append(S, [H], R).
append([], E, [E]).
append([H|T], E, [H|R]) :- append(T, E, R).

```

ADAPT makes its selection of the "best" normal form implementation by generating (if necessary) a normal form implementation that decomposes its input in the same direction (front or back) and constructs its output in the same fashion (accumulator or non-accumulator). The current implementation of ADAPT uses the front processing implementation (i.e., the one that decomposes its input list from the front) if it is unable to determine the direction of the input decomposition and uses the non-accumulator implementation if it is unable to find an accumulator. It is possible to extend ADAPT to try multiple normal form implementations in this case, however, the current implementation does not support this feature.

Analysis of the student's program begins by binding the variables  $H_{NF} \leftarrow H_{student}$ ,  $T_{NF} \leftarrow T_{student}$ ,  $R_{NF} \leftarrow Res_{student}$ ,  $S_{NF} \leftarrow Sofar_{student}$ . Then ADAPT performs a term-by-term matching between the programs and matches the `append(S,[H],R)` subgoal in the normal form program to the

append(Sofar,H,Res) subgoal in the student's program. In order to compare these subgoals, ADAPT must standardize them with respect to their arguments. This is done by replacing each subgoal by a common subgoal and adding the necessary standardizing clause to each program producing the following extended student program:

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), p(Sofar, H, Res).
p(A, B, C) :- enqueue(A, B, C).
enqueue(X, [], [X]).
enqueue(X, [H|T], [H|R]) :- enqueue(X, T, R).
```

and the following extended normal form program:

```
reverse([], []).
reverse([H|T], R) :- reverse(T, S), p(S, H, R).
p(S, H, R) :- append(S, [H], R).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

We can transform each of these programs via unfolding/folding producing the following identical subprogram for both the normal form and the student's program:

```
p([], E, [E]).
p([H|T], E, [H|R]) :- p(T, E, R).
```

so the student's program is recognized as correct even though its append/3 subgoal invocation has a different argument structure than the normal form program.

Unfolding is extremely powerful with respect to program reduction enabling ADAPT to recognize drastically different program implementations. One example of this is the following member/2 implementation:

```
member(L, E) :- append(_, [E|_], L).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

which can be transformed into the following standard member/2 implementation:

```
member([E|_], E).
member([H|T], E) :- member(T, E).
```

Program reduction via unfolding is applicable on a wide range of programs. For example, there are several ways to define the predicate last/2 (which finds the last element in the given input list). One way is to use the predicate append/3:

```
last(L, E) :- append(_, [E], L).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

Unfolding the `append(_, [E], L)` subgoal in the first clause produces:

```
last([E], E).
last([H|T], E) :- append(_, [E], T).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

And then folding the original clause `last(L, E) :- append(_, [E], L)` into the new recursive clause produces the standard `last/2` implementation:

```
last([E], E).
last([H|T], E) :- last(T, [E]).
```

Although capable of recognizing a large class of correct implementations of standard recursive form programs, ADAPT must also be able to deal with incorrect programs. This is discussed in the next section.

### 3.3.2 ADAPT's Approach to Bug Detection

ADAPT's bug detector uses a hierarchical bug library to classify bugs found in incorrect programs. The bug detector begins with the transformed normal form program produced by the algorithm recognizer and attempts a term-by-term matching between this transformed normal form program and the student's program. Obviously, if there are no mismatches then the student's program is correct. If there are mismatches, however, ADAPT attempts to explain them by searching the hierarchical bug library.

ADAPT supports a bug library that is divided into three separate sublibraries for task-related, implementation-related, and general language-related bug rules. Task-related bug rules apply to bugs that are specific to the task (or program) or a set of tasks (or a schema). These bug rules are maintained hierarchically using the same schema hierarchies as the tutoring component. Implementation-related bug rules apply to bugs that are specific to programming techniques (e.g., accumulated composition of the output or backward decomposition of the input). Finally, general language-related bug rules apply to general bugs that are common to all Prolog programs.

The basic process of bug detection is to gather all appropriate bug rules for the student's program and then execute them to see if the student's program contains any of these common bugs. General language-related bugs are actually detected during the algorithm recognition phase and as such are not represented explicitly with rules. Task-related and implementation-related bug rules are gathered as follows. Task-related bug rules are obtained by finding the assigned program in the schema hierarchy and gathering all bug rules associated with that program and any of its ancestors.<sup>13</sup> Implementation-related bug rules are obtained by gathering all bug rules associated with any programming techniques that the student used in her program.

---

<sup>13</sup>Schema hierarchies are described in detail in (Gegg-Harrison, 1989). The use of schema hierarchies within our schema-based Prolog tutor is given in Chapter 5.

Let's look at some examples of bug rules and their applicability to actual student programs. In the following program (variation #3 in Appendix B):

```
reverse([],L,L).
reverse([],[]).
reverse(X,Y) :- reverse(X,[],Y).
reverse([H|T],Y,Z) :- reverse(T,[H|Y],Z).
```

ADAPT recognizes that the second clause is actually a redundant base case when it discovers that the student's program has more clauses than the transformed normal form program. Note that ADAPT unfolds the normal form program to produce the appropriate base case and then matches it to the student's extra base case to ensure that the base case is redundant rather than being an incorrect extra base case as in the following program (variation #50 in Appendix B):

```
reverse([],[]).
reverse([],Y).
reverse([H|T],[X]) :- reverse(T,[X|H]).
```

Missing definitions are also classified as general language-related bugs. When a missing definition is discovered as in the following program (variation #9 in Appendix B):

```
reverse([],[]).
reverse(L,[H|RT]) :- append(T,[H],L),reverse(T,RT).
```

the student is asked to enter a definition for the missing definition (append/3 in this case) before any analysis is performed. Likewise, if the student's program is not structural recursive as in the following program (variation #18 in Appendix B):

```
app([],L,L).
app([H|L1],L2,[H|L3]) :- app(L1,L2,L3).
reverse([],[]).
reverse([H|T],Res) :- app(Sofar,[H],Res),reverse(T,Sofar).
```

the student is informed that app/3 is not structural recursive and is asked to re-enter a structural recursive program. Note that ADAPT has no trouble recognizing the errors with programs that contain multiple predicates that are not structural recursive as can be seen in the following program (variation #30 in Appendix B):

```
reverse([],[]).
reverse([Head|Tail],X) :- append(Y,[Head],X),reverse(Y,Tail).
append([],List,List).
append([H|T],List2,Result) :- Result=[H|X],append(T,List2,X).
```

where neither reverse/2 nor append/3 are structural recursive. Missing clauses also constitute general language-dependent mistakes. Note, however, that the detection of a missing base case in the following program (variation #55 in Appendix B):

```
reverse([H|T],Y) :- reverse(T,[H|Y]).
```

does not prevent the analysis of the program which may (as it does in this case) contain other errors. Another common misconception held by novice Prolog programmers (especially those with prior experience to programming) is a reassignment bug. Because the use of variables in Prolog is different from other conventional languages permitting destructive assignment (where variables can contain different assignments throughout the execution of the program), many novice Prolog programmers attempt to rebind a bound variable to a new value as in the following program (variation #44 in Appendix B):

```
reverse([],L).
reverse([H],Ans) :- Ans=[Ans|H].
reverse([H|T],Ans) :- Ans=[Ans|H],reverse(T,Ans).
```

The last class of general language-related bugs are term mismatches. For example, if a student codes a variable for a constant argument or codes the wrong variable or constant then this is detected during the term-by-term matching between the student's program and the transformed normal form program. For extremely buggy programs, this information can be almost meaningless since almost every argument would be flagged as a mismatch. However, for many buggy programs this information actually pinpoints the error in the student's program. For example, in the following reverse/2 (variation #25 in Appendix B):

```
reverse([],[]).
reverse([H|T],Res) :- reverse(T,Sofar),append(Sofar,H,Res).
append([],L,L).
append([H|L1],L2,[H,L3]) :- append(L1,L2,L3).
```

the student is attempting to append a single element to the end of a list. The problem is that the subgoal append(Sofar,H,Res) assumes that the second argument is a single element while the definition of append/3 assumes the second argument is a list of elements. ADAPT always assumes the top-most predicate is correct (which is not always a correct assumption) so it finds a mismatch in the second argument of the append/3 base case definition rather than in the second argument in the subgoal append(Sofar,H,Res). Whether ADAPT's assumption is correct or not is irrelevant since ADAPT has pinpointed the argument in error which is sufficient advice to enable the student to correct her program.

Technique-level and schema-level bug rules use this mismatch information in order to detect more specific types of errors. For example, in the following reverse/2 program (variation #2 in Appendix B):

```
reverse(X,Y) :- reverse(X,_,Y).
reverse([],L,L).
reverse([H|T],Y,Z) :- reverse(T,[H|Y],Z).
```

the term-by-term matching will determine that there is an (anonymous) variable for a constant in the second argument of the first subgoal of reverse/2. But since this mismatch occurs in the accumulator argument of the initializing clause (which ADAPT knows because the student has

employed the accumulator technique), ADAPT is able to recognize that the error is actually an initialization error. Likewise, schema-level bug rules use mismatch information to provide more specific error descriptions. For example, in the following `reverse/2` program (variation #36 in Appendix B):

```
reverse([], []).  
reverse([H|T1],[T2|H]) :- reverse(T1,T2).
```

the student has attempted to append an element to the tail of the list using the list constructor. This shows up as a constant for a variable (i.e., `[T2|H]` in place of `L`) and a missing subgoal (i.e., `append(T2,[H],L)`). The "invalid append" bug rule tells ADAPT that whenever these two mismatches occur within a `reverse/2` program then the student probably has a misconception with the use of list constructors. Note that the "invalid append" bug rule is also a back processing technique-level bug rule since it identifies a bug which could occur in any global list processing program which decomposes its input from the back.

In addition to providing the student with advice on how to fix an erroneous program, accurate bug detection is essential to the student modeling component of an automated tutoring system. Accurate student modeling enables the tutoring system to adapt the instruction to fit the needs of its students by providing them with appropriate remediation. This is discussed in the last chapter.

## Chapter 4

# ADAPT Debugger

In this chapter, we will formally define a class of programs that contains most reasonable implementations for a subset of the single pass recursive list processing programs. The goal of this chapter is the development of the Characterization Theorem which characterizes the domain of the ADAPT debugger. We begin with some preliminary definitions. It is important to note that we are restricting our attention to programs which are free of mutual recursion. This restriction implies a partial ordering on the predicates of the program defined by the acyclic predicate dependency graph.

We need to define the notion of an associative predicate for predicates with three list argument positions in which two of the list arguments are combined in some way to produce the third list argument. For example, the predicate `append/3` is associative:

```
append( [], L, L ).
append( [H|T], X, [H|M] ) :- append( T, X, M ).
```

There are two ways of appending three lists A, B, and C together. We can either append B to the end of A and then append C to end of this list or we can append C to the end of B and then append this list to the end of A (i.e., `append(A,B,X),append(X,C,D)` or `append(B,C,Y),append(A,Y,D)`). We also need to define the notions of left and right identities for associative predicates. For example, the empty list `[]` is the left and right identity for the predicate `append/3`. These notions are formalized in the following definitions.

**Definition.** Let  $P$  be the definition of  $p/3$ . Let  $Q$  be the program obtained by adding the clause  $q(A, B, C, D) :- p(A, B, X), p(X, C, D)$  to  $P$  and let  $Q^*$  be the program obtained by adding the clause  $q(A, B, C, D) :- p(B, C, Y), p(A, Y, D)$  to  $P$ . Then  $p/3$  is an associative predicate if  $Q$  and  $Q^*$  are equivalent for  $q/4$ .

**Definition.** Let  $p/3$  be an arbitrary predicate. If  $p(X, \epsilon, X)$  succeeds for all  $X$  that are unifiable with `[]` or `[H|T]` then  $\epsilon$  is a right identity for  $p/3$ . Likewise, if  $p(\epsilon, X, X)$  succeeds for all  $X$  that are unifiable with `[]` or `[H|T]` then  $\epsilon$  is a left identity for  $p/3$ . We say that  $\epsilon$  is an identity if epsilon is both a left identity and a right identity.

**Definition.** A subgoal  $p(A, B, C)$  is an associative subgoal if  $p/3$  is an associative predicate.

Another example of an associative predicate is `conjoin/3` (which appends two lists together leaving a separator between them):

```
conjoin( [], L, [$_|L] ).
conjoin( [H|T], X, [H|M] ) :- conjoin( T, X, M ).
```

For any associative subgoal, the following rewrite rule can be invoked to replace the occurrence of one subgoal pair with another without altering the semantics of the program (i.e., the programs are equivalent):

**Associative Rewrite Rule:**

$$p(A, B, X), p(X, C, D) \text{ iff } p(B, C, Y), p(A, Y, D)$$

For example, in the following reverse/2 program:

```
(C1) reverse([], []).
(C2) reverse([X], [X]).
(C3) reverse([G, H|T], L) :- reverse(T, X), append([H], [G], M), append(X, M, L).
(C4) append([], L, L).
(C5) append([H|T], X, [H|R]) :- append(T, X, R).
```

the subgoal pair `append([H],[G],M),append(X,M,L)` in clause (C3) can be rewritten as `append(X,[H],M),append(M,[G],L)` since `append/3` is an associative predicate:

```
(C1) reverse([], []).
(C2) reverse([X], [X]).
(C3) reverse([G, H|T], L) :- reverse(T, X), append(X, [H], M), append(M, [G], L).
(C4) append([], L, L).
(C5) append([H|T], X, [H|R]) :- append(T, X, R).
```

In order for a single program implementation to serve as a normal form implementation, a general transformation scheme must be defined which can transform it into the other representative program implementations. Since these additional representative programs actually employ general programming techniques, it is possible to define a transformation scheme that is applicable to large classes of programs. In the remainder of this chapter, we present an equivalence-preserving transformation scheme which enables the generation of all standard recursive form programs from a single standard recursive *normal form* program for the task. Let's look at some examples of what is required of this transformation scheme. Consider the reverse/2 program which has the following normal form program:

```
(A1) reverse([], []).
(A2) reverse([H|T], R) :- reverse(T, S), append(S, [H], R).
(A3) append([], L, L).
(A4) append([H|T], X, [H|R]) :- append(T, X, R).
```

Let's look at the steps that would have to be taken to transform this program into the "accumulator" version. First of all, an "initializing clause" must be introduced which creates a new argument for the accumulator which is initialized to the empty list:

```
(B1) reverse(L, R) :- reverse(L, [], R).
(A1) reverse([], []).
(A2) reverse([H|T], R) :- reverse(T, S), append(S, [H], R).
```

```
(A3) append([ ],L,L).
(A4) append([H|T],X,[H|R]) :- append(T,X,R).
```

Then the base case clause (A1) must be changed to copy the accumulator over to the output variable:

```
(B1) reverse(L,R) :- reverse(L,[ ],R).
(B2) reverse([ ],R,R).
(A2) reverse([H|T],R) :- reverse(T,S),append(S,[H],R).
(A3) append([ ],L,L).
(A4) append([H|T],X,[H|R]) :- append(T,X,R).
```

The next step is to transform the recursive step clause (A2) to build its output in the opposite direction. This can be done by appending the element to the front of the accumulator rather than the back of the partial result:

```
(B1) reverse(L,R) :- reverse(L,[ ],R).
(B2) reverse([ ],R,R).
(B3) reverse([H|T],X,R) :- append([H],X,Y),reverse(T,Y,R).
(A3) append([ ],L,L).
(A4) append([H|T],X,[H|R]) :- append(T,X,R).
```

Finally, by unfolding the `append([H],X,Y)` subgoal away, the well-known "accumulator" `reverse/2` implementation is obtained:

```
(B1) reverse(L,R) :- reverse(L,[ ],R).
(B2) reverse([ ],R,R).
(B3) reverse([H|T],X,R) :- reverse(T,[H|X],R).
```

Another approach to reversing the elements of a list is to recursively remove the last element of the input list and append it to the front of the output list resulting from reversing the remainder of the input list. Since such a program will be processing the list from the back of the list instead of the front of the list, the output list must be constructed from the front instead of the back. Note that the base case clause (A1) remains the same regardless of the directionality of the input list decomposition. We consider the transformation process:

```
(A1) reverse([ ],[ ]).
(A2) reverse([H|T],R) :- reverse(T,S),append(S,[H],R).
(A3) append([ ],L,L).
(A4) append([H|T],X,[H|R]) :- append(T,X,R).
```

For the recursive step clause (A2), two transformations must be made. First of all, the input list must be decomposed into the last element and the remainder of the list. This can be accomplished by invoking `append(T,[H],L)`.<sup>14</sup> Secondly, the output construction process must be "reversed" since

---

<sup>14</sup>Note that for `reverse/2`, the definition of `append/3` was already present. In general, a definition for `append/3` will have to be introduced.

the new program processes the input list in the opposite order:

```
(C1) reverse([], []).
(C2) reverse(L,R) :- append(T,[H],L),reverse(T,S),append([H],S,R).
(C3) append([],L,L).
(C4) append([H|T],X,[H|R]) :- append(T,X,R).
```

Finally, by unfolding the `append([H],S,R)` subgoal away, the well-known "inverse naive" `reverse/2` implementation is obtained:

```
(C1) reverse([], []).
(C2) reverse(L,[H|R]) :- append(T,[H],L),reverse(T,S).
(C3) append([],L,L).
(C4) append([H|T],X,[H|R]) :- append(T,X,R).
```

Notice that both transformations can be applied to produce the following "inverse accumulator" version of `reverse/2`:

```
(D1) reverse(L,R) :- reverse(L,[],R).
(D2) reverse([],R,R).
(D3) reverse(L,X,R) :- append(T,[H],L),append(X,[H],Y),reverse(T,Y,R).
(D4) append([],L,L).
(D5) append([H|T],X,[H|R]) :- append(T,X,R).
```

## 4.1 Program Transformations

In this section, we formally define the transformations necessary to produce the set of representative standard recursive form programs. These transformations can be generalized to other programs (e.g., programs with more than two arguments), but we provide the specific case which is sufficient for the standard recursive normal form programs (e.g., "naive" `reverse/2`) in an effort to simplify the discussion. Before presenting the transformations, we formalize the notion of normal form in the following definition.

**Definition.** A program is a standard recursive normal form program for mode  $r(+, -)$  if it has the following form:

```
r([],β).
r([H|T],L) :- r(T,M),s(M,f(H),L).
```

where  $r/2$  and  $s/3$  are arbitrary predicate symbols,  $s/3$  is an associative predicate with identity  $\beta$  and  $f$  is an optional function (including  $f(H) = [H]$ ).

The class of standard recursive normal form programs includes "naive" `reverse/2` and `level/2` (which takes as input a list of lists and levels out the first level of lists producing a list of the elements of the inner lists):

```

level([], []).
level([H|T], L) :- level(T, M), combine(M, H, L).
combine(L, [], L).
combine(X, [H|T], [H|R]) :- combine(X, T, R).

```

Notice that if the infix predicate `is/2` is redefined to be the separate predicates `is+/3` (for `A is B+C`) and `is*/3` (for `A is B*C`) and the notion of identity is modified appropriately (i.e., `X` must be unifiable with any legal integer) then `sum/2` (which calculates the sum of all the elements in a list):

```

sum([], 0).
sum([H|T], L) :- sum(T, M), is+(M, H, L).

```

and `product/2` (which calculates the product of all the elements in a list):

```

product([], 1).
product([H|T], L) :- product(T, M), is*(M, H, L).

```

are also standard recursive normal form programs. Now we are ready to present the transformations.

**Accumulator Transformation.** For each standard recursive normal form program for mode `p(+, -)`, the following accumulator implementation of the original program exists:

1. Add the following initializing clause:

```
p(I, R) :- p(I, β, R).
```

2. Replace each terminating clause:

```
p([], β).
```

with the clause:

```
p([], R, R).
```

3. Replace the recursive clause:

```
p([H|T], R) :- p(T, S), q(S, f(H), R).
```

with the clause:

```
p([H|T], S, R) :- q(f(H), S, X), p(T, X, R).
```

Note that `q/3` is an associative predicate with identity `β`. We say that the resultant program of this transformation is an **accumulator normal form program**.

**Theorem.** If  $P^*$  is the program obtained by applying the Accumulator Transformation to  $P$  (which is a standard recursive normal form program defining  $p/n$ ) then  $P^*$  is equivalent to  $P$  for  $p/n$ .

**Proof:** We provide an inductive proof using the length of the input list.

*Basis:* Consider the query  $p([], R)$ . For the original program  $P$ , the query  $p([], R)$  will only unify with the base case clause:

$$p([], \beta).$$

which succeeds binding  $R \leftarrow \beta$ . For the accumulator program  $P^*$ , the query  $p([], R)$  will only unify with the clause:

$$p(I, R) :- p(I, \beta, R).$$

which replaces the query with the goal  $p([], \beta, R)$  which will only unify with the clause:

$$p([], R, R).$$

which succeeds binding  $R \leftarrow \beta$ . Since both  $P$  and  $P^*$  produce either:

- exactly one answer to the query  $p([], R)$  if  $R$  unifies with  $\beta$

-or-

- no answers to the query  $p([], R)$  if  $R$  will not unify with  $\beta$

it follows that  $P$  and  $P^*$  are equivalent for  $p/n$  with respect to queries with length 0 input lists.

*Inductive Hypothesis:* Assume the  $P$  and  $P^*$  are equivalent for  $p/n$  with respect to queries with length  $i$  input lists. Consider the query  $p([B_1, \dots, B_i], R)$ . Since it occurs uniformly throughout, we drop the function application  $f(H)$  leaving simply  $H$ . Given this, the subgoals produced by the original program  $P$ :

$$q(\beta, B_i, R_{i-1}), q(R_{i-1}, B_{i-1}, R_{i-2}), \dots, q(R_2, B_2, R_1), q(R_1, B_1, R)$$

are equivalent to the subgoals produced by the accumulator program  $P^*$ :

$$q(B_1, \beta, X_1), q(B_2, X_1, X_2), \dots, q(B_{i-1}, X_{i-2}, X_{i-1}), q(B_i, X_{i-1}, R)$$

In order for  $P$  to be equivalent to  $P^*$  for  $p/n$ , they must both produce the same binding for the output argument  $R$  for the query  $p([B_1, \dots, B_i], R)$ . Let's represent each  $q(\alpha_1, \alpha_2, \alpha_3)$  subgoal by the substituting  $(\alpha_1 \alpha_2)$  for all occurrences of  $\alpha_3$ . This representation highlights  $q/3$ 's associative property and is intended to make the following proof more readable. Applying this representation to the subgoals produced for  $P$  and  $P^*$  gives us the following representation for

R in P:

$$(((\dots((\beta B_i) B_{i-1})\dots) B_2) B_1)$$

and the following representation for R in P\*:

$$(B_i (B_{i-1} (\dots (B_2 (B_1 \beta))\dots)))$$

*Inductive Step:* We must show that programs are equivalent for queries with length  $i+1$  input lists. Consider the query  $p([B_1, \dots, B_{i+1}], R)$ . Note that since  $p/2$  is a standard recursive normal form program, each query and subsequent subgoals will only unify with at most one clause in the program. The same is true of the accumulator program since it only has one base case clause (for the empty list) and one recursive step clause (for the non-empty list). Thus, each query either succeeds exactly once or it fails. Therefore, both directions of the equivalence property are covered by showing that R must have the same answer for both P and P\*.

For the query  $p([B_1, \dots, B_{i+1}], R)$ , the subgoals produced by the original program P are:

$$(\beta, B_{i+1}, R_i), q(R_i, B_i, R_{i-1}), \dots, q(R_2, B_2, R_1), q(R_1, B_1, R)$$

and the subgoals produced by the accumulator program P\* are:

$$q(B_1, \beta, X_1), q(B_2, X_1, X_2), \dots, q(B_i, X_{i-1}, X_i), q(B_{i+1}, X_i, R)$$

This gives us the following representation for R in P:

$$(((\dots((\beta B_{i+1}) B_i)\dots) B_2) B_1)$$

and the following representation for R in P\*:

$$(B_{i+1} (B_i (\dots (B_2 (B_1 \beta))\dots)))$$

We must show that:

$$(((\dots((\beta B_{i+1}) B_i)\dots) B_2) B_1) = (B_{i+1} (B_i (\dots (B_2 (B_1 \beta))\dots)))$$

Since  $\beta$  is an identity of  $q/3$ , we know that  $(\beta B_{i+1})$  is equivalent to  $(B_{i+1} \beta)$ . Thus, we can rewrite the equation as:

$$(((\dots((B_{i+1} \beta) B_i)\dots) B_2) B_1) = (B_{i+1} (B_i (\dots (B_2 (B_1 \beta))\dots)))$$

Since  $q/3$  is associative, we know that  $((B_{i+1} \beta) B_i)$  is equivalent to  $(B_{i+1} (\beta B_i))$ . Thus, we can rewrite this equation as:

$$(((\dots(B_{i+1} (\beta B_i))\dots) B_2) B_1) = (B_{i+1} (B_i (\dots (B_2 (B_1 \beta))\dots)))$$

Using the fact that  $\alpha/3$  is associative again, we know that  $((B_{i+1} (\beta B_i)) B_{i-1})$  is equivalent to  $((B_{i+1} ((\beta B_i) B_{i-1}))$ . Rewriting the equation using the associative property of  $\alpha/3$  in this manner  $i-1$  times produces the following equation:

$$(B_{i+1} (((\dots((\beta B_i)B_{i-1})\dots) B_2) B_1) = (B_{i+1} (B_i (\dots(B_2 (B_1 \beta))\dots)))$$

Using the inductive hypothesis which tells us that:

$$(((\dots((\beta B_i) B_{i-1})\dots) B_2) B_1) = (B_i (B_{i-1} (\dots(B_2 (B_1 \beta))\dots)))$$

we can rewrite the equation into the following equality:

$$(B_{i+1} (B_i (\dots(B_2 (B_1 \beta))\dots))) = (B_{i+1} (B_i (\dots(B_2 (B_1 \beta))\dots)))$$

So the accumulator transformation produces an equivalent program.  $\square$

The Accumulator Transformation is a commonly known programming technique and has appeared throughout the literature (Clark & Tärnlund, 1977; Hansson & Tärnlund, 1982; Zhang & Grant, 1988) as a method of improving program efficiency. It also appears in most Prolog texts (Clocksin & Mellish, 1987; Sterling & Shapiro, 1986; O'Keefe, 1990). Another transformation that is possible on programs which use associative subgoals is the Back Processing Transformation. Because this transformation does not improve efficiency (in fact, except for the rare case of *reverse/2* where the efficiency remains the same, this transformation actually degrades the performance of the program), it has not appeared in the literature. In the present application, however, it is desirable to capture as many correct implementations as possible even if they turn out to be very inefficient implementations for a given program.

**Back Processing Transformation.** For each standard recursive normal form program for mode  $p(+, -)$  and for each accumulator normal form program for mode  $p(+, -)$ , the following back processing implementation of the original program exists:

1. Add the following decomposition program:

```
append([ ], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

where *append/3* is an arbitrary predicate symbol that does not already occur in the program.

2. Keep the terminating clause and the initializing clause (if one is present).

3. Apply the appropriate replacement to the recursive clause:

- If the program's recursive clause has the form:

$$p([H|T], R) :- p(T, S), q(S, f(H), R).$$

then replace it with the clause:

$$p(L, R) :- \text{append}(Z, [A], L), p(Z, S), q(f(A), S, R).$$

- If the program's recursive clause has the form:

$$p([H|T], S, R) :- q(f(H), S, X), p(T, X, R).$$

then replace it with the clause:

$$p(L, S, R) :- \text{append}(Z, [A], L), q(S, f(A), X), p(Z, X, R).$$

Note that  $q/3$  is an associative predicate with identity  $\beta$ .

In order to show that the Back Processing Transformation produces an equivalent program, we must first show that our decomposition program  $\text{append}/3$  correctly decomposes the input list.

**Decomposition Lemma.** The subgoal  $\text{append}(A, [Z], L)$  with mode  $\text{append}(-, -, +)$  removes the last element  $Z$  from the list  $L$  leaving the remainder of the list in  $A$  given the program:

$$\begin{aligned} &\text{append}([ ], L, L). \\ &\text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L). \end{aligned}$$

**Proof:** We provide an inductive proof using the length of list  $L$ .

*Basis:* Let  $L$  be a single element list (i.e.,  $L = [X]$ ). Thus, we are considering the query  $\text{append}(A, [Z], [X])$ . If we unify this subgoal with the second clause of the  $\text{append}/3$  program then the subgoal is replaced with the subgoal  $\text{append}(T1, [Z], [ ])$  which will not unify with any of the clauses in the  $\text{append}/3$  program. The only other possibility is to unify the original subgoal  $\text{append}(A, [Z], [X])$  with the first clause in the  $\text{append}/3$  program which results in the mapping  $A \leftarrow [ ]$  and  $Z \leftarrow X$  which is the desired bindings.

*Inductive Hypothesis:* Assume  $\text{append}(A, [Z], L)$  removes the last element  $Z$  from list  $L$  leaving the remainder of the list in  $A$  for lists  $L$  of length  $i > 1$ .

*Inductive Step:* Let  $L$  be a list of length  $i+1$ . Consider the query  $\text{append}(A, [Z], L)$  (where the length of  $L$  is  $i+1$ ). Obviously,  $\text{append}(A, [Z], L)$  does not unify with the first clause since that would require binding a single element list to a list with more than one element which is not

possible. So  $\text{append}(A, [Z], L)$  must unify with the second clause which replaces the query with the subgoal  $\text{append}(T1, [Z], L1)$  and makes the bindings  $A \leftarrow [H1 | T1]$  and  $L \leftarrow [H1 | L1]$ . Now since the length of  $L1$  is  $i$  we know by the inductive hypothesis that  $Z$  is the last element of  $L1$  and that  $T1$  is the remainder of the list. Then it follows that copying the first element of list  $L$  over as the first element in list  $A$  produces the desired bindings.  $\square$

**Theorem.** If  $P^*$  is the program obtained by applying the Back Processing Transformation to  $P$  (which is a standard recursive normal form program or an accumulator normal form program defining  $p/n$ ) then  $P^*$  is equivalent to  $P$  for  $p/n$ .

**Proof:** We provide an inductive proof using the length of the input list. As with the Accumulator Transformation proof, each query and subsequent subgoals will only unify with at most one clause in both  $P$  and  $P^*$ . Thus, each query either succeeds exactly once or it fails. Therefore, both directions of the equivalence property are covered by showing that  $R$  must have the same answer for both  $P$  and  $P^*$ .

*Basis:* For length 0 input lists, the query unifies with the base case clause. Note that the base case clause is identical in both programs. Thus, for both the original program  $P$  and the back processing program  $P^*$ , the query  $p([], R)$  succeeds exactly once iff  $R$  unifies with  $\beta$ . Otherwise, the query  $p([], R)$  fails in both  $P$  and  $P^*$ . So the  $P$  and  $P^*$  are equivalent for  $p/n$  with respect to queries with length 0 input lists.

*Inductive Hypothesis:* Assume that  $P$  and  $P^*$  are equivalent for  $p/n$  with respect to queries with length  $i$  input lists. Consider the query  $p([B_1, \dots, B_i], R)$ . The Decomposition Lemma tells us that the subgoal sequence  $\text{append}(Z_1, [A_1], [B_1, \dots, B_i]), \text{append}(Z_2, [A_2], Z_1), \dots, \text{append}(Z_i, [A_i], Z_{i-1})$  will bind  $A_{j+1}$  to  $B_{i-j}$ . Thus, the elements of the input list are processed in reverse order. There are two cases to consider: standard recursive normal form programs and accumulator normal form programs. Since it occurs uniformly throughout, we drop the function application  $f(H)$  leaving simply  $H$ .

Case 1 - standard recursive normal form programs

The subgoals produced by the original program  $P$ :

$$q(\beta, B_i, R_{i-1}, q(R_{i-1}, B_{i-1}, R_{i-2}), \dots, q(R_2, B_2, R_1), q(R_1, B_1, R))$$

are equivalent to the subgoals produced by the back processing program  $P^*$ :

$$q(B_1, \beta, R_{i-1}, q(B_2, R_{i-1}, R_{i-2}), \dots, q(B_{i-1}, R_2, R_1), q(B_i, R_1, R))$$

In order for  $P$  to be equivalent to  $P^*$  for  $p/n$ , they must both produce the same binding for the output argument  $R$  for the query  $p([B_1, \dots, B_i], R)$ . For readability, we will represent each  $q(\alpha_1, \alpha_2, \alpha_3)$  subgoal by the substituting  $(\alpha_1 \alpha_2)$  for all occurrences of  $\alpha_3$  as we did in the Accumulator Transformation proof. Applying this representation to the subgoals produced for  $P$  and  $P^*$  gives us

the following representation for R in P:

$$(((\dots ((\beta B_1) B_{i-1}\dots) B_2) B_1))$$

and the following representation for R in P\*:

$$(B_i (B_{i-1}\dots (B_2 (B_1 \beta))\dots))$$

Case 2 - accumulator normal form programs

The subgoals produced by the original program P:

$$q(B_1, \beta, X_1), q(B_2, X_1, X_2), \dots, q(B_{i-1}, X_{i-2}, X_{i-1}), q(B_i, X_{i-1}, R)$$

are equivalent to the subgoals produced by the back processing program P\*:

$$q(\beta, B_i, X_{i-1}), q(X_{i-1}, B_{i-1}, X_{i-2}), \dots, q(X_2, B_2, X_1), q(X_1, B_1, R)$$

In order for P to be equivalent to P\* for p/n, they must both produce the same binding for the output argument R for the query  $p([B_1, \dots, B_i], R)$ . The subgoals for P can be represented as:

$$(B_i (B_{i-1}\dots (B_2 (B_1 \beta))\dots))$$

and the subgoals for P\* can be represented as:

$$(((\dots ((\beta B_1) B_{i-1}\dots) B_2) B_1))$$

Note that the representation of the subgoals for the original standard recursive normal form program are identical to that of the back processing accumulator normal form program. Likewise, the representation of the subgoals for the original accumulator normal form program are identical to that of the back processing standard recursive normal form program. Given this symmetry, the remainder of the proof will consider only the standard recursive normal form case since a similar argument will handle the accumulator normal form case.

*Inductive Step:* We must show that programs are equivalent for queries with length  $i+1$  input lists. Consider the query  $p([B_1, \dots, B_{i+1}], R)$ . Note that since p/2 is a standard recursive normal form program, each query and subsequent subgoals will only unify with at most one clause in the program. The same is true of the accumulator program since it only has one base case clause (for the empty list) and one recursive step clause (for the non-empty list). Thus, each query either succeeds exactly once or it fails. Therefore, both directions of the equivalence property are covered by showing that R must have the same answer for both P and P\*.

The subgoals produced by the original standard recursive normal form program P are:

$$q(\beta, B_{i+1} R_i), q(R_i, B_i, R_{i-1}), \dots, q(R_2, B_2, R_1), q(R_1, B_1, R)$$

and the subgoals produced by the back processing standard recursive normal form program P\* are:

$$q(B_1, \beta, R_i), q(B_2, R_i, R_{i-1}), \dots, q(B_i, R_2, R_1), q(B_{i+1} R_1, R)$$

This gives us the following representation for R in P:

$$(((\dots ((\beta B_{i+1} B_i) \dots) B_2) B_1)$$

and the following representation for R in P\*:

$$(B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots))$$

We must show that:

$$(((\dots ((\beta B_{i+1} B_i) \dots) B_2) B_1) = (B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots)))$$

Since  $\beta$  is an identity of  $q/3$ , we know that  $(\beta B_{i+1}$  is equivalent to  $(B_{i+1} \beta)$ . Thus, we can rewrite the equation as:

$$(((\dots ((B_{i+1} \beta) B_i) \dots) B_2) B_1) = (B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots)))$$

Since  $q/3$  is associative, we know that  $((B_{i+1} \beta) B_i)$  is equivalent to  $(B_{i+1} \beta B_i)$ . Thus, we can rewrite this equation as:

$$(((\dots (B_{i+1} \beta B_i) \dots) B_2) B_1) = (B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots)))$$

Using the fact that  $q/3$  is associative again, we know that  $((B_{i+1} \beta B_i) B_{i-1})$  is equivalent to  $(B_{i+1} (\beta B_i) B_{i-1})$ . Rewriting the equation using the associative property of  $q/3$  in this manner  $i$  minus 1 times produces the following equation:

$$(B_{i+1} (((\dots ((\beta B_i) B_{i-1} \dots) B_2) B_1)) = (B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots)))$$

Using the inductive hypothesis which tells us that:

$$(((\dots ((\beta B_i) B_{i-1} \dots) B_2) B_1) = (B_i (B_{i-1} \dots (B_2 (B_1 \beta)) \dots)))$$

we can rewrite the equation into the following equality:

$$(B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots)) = (B_{i+1} B_i (\dots (B_2 (B_1 \beta)) \dots))$$

So the back processing transformation produces an equivalent program.  $\square$

We also need transformations which allow us to permute the order of the subgoals within a clause and clauses within the program.

**Clause Permutation Transformation.** For program P, any pair of clauses may be permuted.

**Subgoal Permutation Transformation.** Let P be a program which is structural recursive for some mode. Then for any clause C in P of the form  $p(X_1, \dots, X_n) :- \theta_1, q(Y_1, \dots, Y_i), r(Z_1, \dots, Z_j), \theta_2$ , the clause can be replaced in P with the following clause:  $p(X_1, \dots, X_n) :- \theta_1, r(Z_1, \dots, Z_j), q(Y_1, \dots, Y_i), \theta_2$  assuming P remains structural recursive for the given mode.

The clause permutation and subgoal permutation transformations do not change the declarative semantics of the program. For the class of structural recursive programs, the procedural semantics is also not altered since the definition of the subgoal permutation transformation requires the program to remain structural recursive and universal termination is not affected by clause order. One final transformation is also necessary for list decomposition and composition. The list constructor transformation enables the trivial transformation of either:

- . the argument [H|T] to L (where L does not appear elsewhere in the clause) and adding the subgoal invocation  $\text{append}([H],T,L)$  with a corresponding normal form program definition for  $\text{append}/3$
- . the argument [H|T] to [A|B] (where A and B do not appear elsewhere in the clause) and adding the subgoal invocation  $\text{append}([H],T,[A|B])$  with a corresponding normal form program definition for  $\text{append}/3$
- . the subgoal invocation  $\text{append}(M,[H],R)$  to  $\text{append}(M,[H],[A|B])$  for invocations of the normal form program definition for  $\text{append}/3$  (where A and B do not appear elsewhere in the clause)

**List Constructor Transformation - format 1.** For program P defining p/n which contains the literal  $p(X_1, \dots, X_{i-1}, [H|T], X_{i+1}, \dots, X_n)$  which does not contain the variables L,A,B, the following list constructor transformation can be applied to the original program:

1. Add the following decomposition program (if necessary):

$$\begin{aligned} & \text{append}([ ], L, L). \\ & \text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L). \end{aligned}$$

2. If the literal occurs as a subgoal in the body of a clause then replace it with the following subgoals:

$$p(X_1, \dots, X_{i-1}, \omega, X_{i+1}, \dots, X_n), \text{append}([H], T, \omega)$$

3. If the literal occurs in the head of a clause then replace it with the following literal:

$$p(X_1, \dots, X_{i-1}, \omega, X_{i+1}, \dots, X_n)$$

and add the following subgoal at the beginning of the body of the clause:

$$\text{append}([H], T, \omega)$$

where  $\omega$  is either L or [A|B].

**List Constructor Transformation - format 2.** For program P which contains a clause of the form  $p(X_1, \dots, X_n) :- \theta_1, \text{append}(M, [H], R), \theta_2$  and the normal form definition of append/3:

$$\begin{aligned} &\text{append}([], L, L). \\ &\text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L). \end{aligned}$$

the following list constructor transformation can be applied to the original program:

Replace all occurrences of R with [A|B] in the clause  $p(X_1, \dots, X_n) :- \theta_1, \text{append}(M, [H], R), \theta_2$  where A and B do not appear elsewhere in the clause.

We will say list constructor transformation when referring to both transformations. If we want to specifically refer to format 1 then we will say LC1 transformation. Likewise, if we want to refer to format 2 then we will say LC2 transformation.

**Theorem.** If  $P^*$  is the program obtained by applying the LC1 Transformation to P (which is a program defining p/n) then  $P^*$  is equivalent to P for p/n.

**Proof:** Unfolding the  $\text{append}([H], T, \omega)$  subgoal twice produces the original program.  $\square$

**Theorem.** If  $P^*$  is the program obtained by applying the LC2 Transformation to P (which is a program defining p/n) then  $P^*$  is equivalent to P for p/n.

**Proof:** There are only 2 possibilities for list arguments: the empty list [] and the non-empty list [A|B]. We must show that it is not possible for the third argument in the subgoal  $\text{append}(M, [H], R)$  to be the empty list []. Assume that R is []. Then the subgoal  $\text{append}(M, [H], [])$  will fail given the following definition of append/3:

$$\begin{aligned} &\text{append}([], L, L). \\ &\text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L). \end{aligned}$$

which is required for the LC2 transformation. Since R cannot be the empty list [], we can conclude that R can be replaced with the non-empty list [A|B].  $\square$

## 4.2 Decomposing/Composing Programs

In this section, we formally define the class of decomposing/composing programs which will serve as auxiliary programs for the standard recursive form programs which are presented in the next section.

**Definition.** A restructuring permutation of a sequence of terms  $t_1, \dots, t_n$  is defined inductively as follows:

1. A permutation of  $t_1, \dots, t_n$  is a restructuring permutation of  $t_1, \dots, t_n$
2. If  $u_1, \dots, u_m$  is a restructuring permutation of  $t_1, \dots, t_n$  then so are:

$$u_1, \dots, u_{i-1} [ u_i ], u_{i+1}, \dots, u_m$$

-and-

$$u_1, \dots, u_{i-2} [ u_{i-1} u_i ], u_{i+1}, \dots, u_m$$

-and-

$$u_1, \dots, u_{i-2} [ u_{i-1} u_i ], u_{i+1}, \dots, u_m$$

-and-

$$u_1, \dots, u_{i-2} f( u_{i-1} u_i ), u_{i+1}, \dots, u_m$$

where  $f$  is an arbitrary function symbol and  $[ u_i ]$ ,  $[ u_{i-1} u_i ]$ , and  $[ u_{i-1} u_i ]$  are Prolog lists

3. Nothing else is a restructuring permutation of  $t_1, \dots, t_n$

**Definition.** The class of decomposing/composing programs is defined inductively as follows:

1.  $P$  is a decomposing/composing program with respect to  $\pi$  if it has one of the following forms:

$$s(\pi([ ], L, L)) .$$

$$s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)) .$$

$$s(\pi([ ], E, [E])) .$$

$$s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)) .$$

(where  $\pi$  is a restructuring permutation and  $s$  is an arbitrary predicate symbol).

2. If P is a decomposing/composing program with respect to  $\pi$  then Q is a decomposing/composing program with respect to  $\pi$  if Q is created from P by either:
  - a. permuting the clauses within P
  - b. permuting the subgoals within any clause in P such that Q remains structural recursive<sup>15</sup>
  - c. applying the LC1 transformation to any literal in P
3. If P is a decomposing/composing program with respect to  $\pi_2$  and p is a main predicate of P then Q is a decomposing/composing program with respect to  $\pi_1$  if Q is formed from P by adding a clause of the form  $q(\pi_1(A,B,C)) :- p(\pi_2(A,B,C))$  where  $\pi_1$  is some restructuring permutation of the variables A, B, and C and q does not already appear in P
4. Nothing else is a decomposing/composing program

Let's look at some examples of decomposing/composing programs. Standard append/3:

```
append( [], L, L ).
append( [H|T], X, [H|L] ) :- append( T, X, L ).
```

is obviously a decomposing/composing program. Other examples of decomposing/composing programs include:

```
enqueue( [H|T], X, Y ) :- enqueue( T, X, L ), Y=[H|L].
enqueue( [], E, [E] ).
```

and

```
enqueue( [ [], X ], X ).
enqueue( [ [A|B], Y ], [A|Z] ) :- enqueue( [B, Y], Z ).
```

and

```
enqueue( C, D, E ) :- append( D, [C], E ).
append( [], L, L ).
append( [H|T], X, [H|L] ) :- append( T, X, L ).
```

and

```
enq2( [A|B], C ) :- append( A, B, C ).
append( [], L, L ).
append( [H|T], X, [H|L] ) :- append( T, X, L ).
```

---

<sup>15</sup>Note that the LC1 transformation potentially adds a subgoal to the recursive clause so it is possible to produce a decomposing/composing program with a recursive clause which contains subgoals that can be permuted.

Decomposing/composing programs serve as auxiliary programs to standard recursive form programs. For standard recursive form programs, we want to restrict the subgoal invocation of decomposing/composing programs to those invocations that result in a single element being removed/added to some input list. Intuitively, single element decomposing/composing subgoals have the form  $s(\pi(A_1, A_2, A_3))$  where  $A_1$  is an input list that is either:

- . being decomposed into  $A_2$  and  $A_3$  where  $A_3$  is the first/last element of  $A_1$  (or a list containing the first/last element of  $A_1$ ) and  $A_2$  is the remainder of list  $A_1$

-or-

- . being composed from  $A_2$  and  $A_3$  where  $A_3$  is a single element (or a list containing a single element) which becomes the first/last element of  $A_1$  and  $A_2$  is the remainder of list  $A_1$

**Definition.** A subgoal  $s(\mathcal{D}(A_1, A_2, A_3))$  is a single element decomposing/composing subgoal if  $\mathcal{D}(A_1, A_2, A_3)$  is either:

$$\pi(A_3, A_2, A_1)$$

-or-

$$\pi(A_2, A_3, A_1)$$

-or-

$$\pi([A_3], A_2, A_1)$$

-or-

$$\pi([A_2], A_3, A_1)$$

where  $\pi$  is a restructuring permutation and the complete definition of  $s$  is a decomposing/composing program with respect to  $\pi$ .

The following theorem shows that it is possible to have auxiliary subgoals with their arguments in different orders and structures than the normal form program.

**Program Reduction Theorem for append/3** Let  $P$  be a program containing a single clause<sup>16</sup> of the form  $p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2$  such that  $s(\pi(A, B, C))$  is a single element decomposing/composing subgoal,  $A, B,$  and  $C$  are arbitrary Prolog variables, and  $t_i$  are arbitrary Prolog terms. If the clause  $p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2$  is replaced by a clause of the form  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2$ , and one of the following sets of clauses:

---

<sup>16</sup>This theorem actually holds for the more general case of a program with any number of clauses of the form  $p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2$ . However, it is stated for the single clause case since that is all that is needed for the Characterization Theorem for reverse/2.

$$\begin{aligned} & \text{append}([ ], L, L) . \\ & \text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L) . \end{aligned}$$

-or-

$$\begin{aligned} & \text{append}([ ], E, [E]) . \\ & \text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L) . \end{aligned}$$

is added to P defining  $P^*$  then there exists a transformation  $\tau$  such that  $P \tau$  is equivalent to  $P^*$  for p/n.

**Proof.** We provide an inductive proof based on the complexity of s.

**Basis.** The predicate s has complexity of 0. :p. There are two cases to consider: s is already in one of following forms or s can be obtained by permuting the clauses in one of the following forms:

$$\begin{aligned} & s(\pi([ ], L, L)) . \\ & s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)) . \end{aligned}$$

-or-

$$\begin{aligned} & s(\pi([ ], E, [E])) . \\ & s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)) . \end{aligned}$$

These cases correspond to #1 and #2a in the definition of decomposing/composing program. Since there are no auxiliary subgoals in complexity 0 predicate definitions, we do not need to consider #2b in the basis case. Cases #2c and #3 both produce programs which have complexity greater than 0. Thus, we only need to consider cases #1 and #2a in the basis case. For case #2a, an inverse transformation exists. This transformation merely permutes the clauses back to the form of case #1 in the decomposing/composing program definition. Setting  $\tau$  to this transformation makes  $P \tau$  equivalent to  $P^*$  for p/n. Consider case #1. Program P must have one of the following forms:

$$\begin{aligned} & p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2 . \\ & s(\pi([ ], L, L)) . \\ & s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)) . \end{aligned}$$

-or-

$$\begin{aligned} & p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2 . \\ & s(\pi([ ], E, [E])) . \\ & s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)) . \end{aligned}$$

This follows directly from the definition of decomposing/composing programs. Let's start with the first form for P:

(A1)  $p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2.$   
 (A2)  $s(\pi([], L, L)).$   
 (A3)  $s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)).$

We start by adding the clause  $\text{append}(A, B, C) :- s(\pi(A, B, C))$  and replacing the clause (A1) with  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2$  creating the following program:

(A1r)  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2.$   
 (A2)  $s(\pi([], L, L)).$   
 (A3)  $s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)).$   
 (A4)  $\text{append}(A, B, C) :- s(\pi(A, B, C)).$

Clearly, this program is equivalent to P for p/n since we can simply unfold the  $\text{append}(A, B, C)$  subgoal in clause (A1r) producing P. Alternatively, however, we can unfold the  $s(\pi(A, B, C))$  subgoal in clause (A4) producing:

(A1r)  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2.$   
 (A2)  $s(\pi([], L, L)).$   
 (A3)  $s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)).$   
 (A5)  $\text{append}([], L, L).$   
 (A6)  $\text{append}([H|T], X, [H|L]) :- s(\pi(T, X, L)).$

Now we can fold the clause (A4) into clause (A6) producing the following program:

(A1r)  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2.$   
 (A2)  $s(\pi([], L, L)).$   
 (A3)  $s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)).$   
 (A5)  $\text{append}([], L, L).$   
 (A7)  $\text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L).$

If we remove the no longer needed definition of  $s$  in clauses (A2) - (A3) then we produce the desired program  $P^*$ :

(A1r)  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2.$   
 (A5)  $\text{append}([], L, L).$   
 (A7)  $\text{append}([H|T], X, [H|L]) :- \text{append}(T, X, L).$

Now consider the second form for P:

(B1)  $p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2.$   
 (B2)  $s(\pi([], E, [E])).$   
 (B3)  $s(\pi([H|T], X, [H|L])) :- s(\pi(T, X, L)).$

We start by adding the clause  $\text{append}(A, B, C) :- s(\pi(A, B, C))$  and replacing the clause (B1) with  $p(t_1, \dots, t_n) :- \theta_1, \text{append}(A, B, C), \theta_2$  creating the following program:

```

(B1r) p(t1, ..., tn) :- θ1, append(A, B, C), θ2.
(B2)  s(π([], E, [E])).
(B3)  s(π([H|T], X, [H|L])) :- s(π(T, X, L)).
(B4)  append(A, B, C) :- s(π(A, B, C)).

```

Clearly, this program is equivalent to P for p/n since we can simply unfold the `append(A, B, C)` subgoal in clause (B1r) producing P. Alternatively, however, we can unfold the `s(π(A, B, C))` subgoal in clause (B4) producing:

```

(B1r) p(t1, ..., tn) :- θ1, append(A, B, C), θ2.
(B2)  s(π([], E, [E])).
(B3)  s(π([H|T], X, [H|L])) :- s(π(T, X, L)).
(B5)  append([], E, [E]).
(B6)  append([H|T], X, [H|L]) :- s(π(T, X, L)).

```

Now we can fold the clause (B4) into clause (B6) producing the following program:

```

(B1r) p(t1, ..., tn) :- θ1, append(A, B, C), θ2.
(B2)  s(π([], E, [E])).
(B3)  s(π([H|T], X, [H|L])) :- s(π(T, X, L)).
(B5)  append([], E, [E]).
(B7)  append([H|T], X, [H|L]) :- append(T, X, L).

```

If we remove the no longer needed definition of `s` in clauses (B2) - (B3) then we produce the desired program P\*:

```

(B1r) p(t1, ..., tn) :- θ1, append(A, B, C), θ2.
(B5)  append([], E, [E]).
(B7)  append([H|T], X, [H|L]) :- append(T, X, L).

```

Induction. Assume the theorem holds when `s` has complexity of `i` minus 1. There are four cases to consider for predicate dependency graphs of depth `i > 1` corresponding to #2a, #2b, #2c, and #3 in the definition of decomposing/composing programs.

For #2a, #2b, and #2c in the definition of decomposing/composing programs, there exist transformations which perform the inverse transformations. Thus, we can assume that we can permute the clauses of the target program P\* to match the order of the clauses of the normal form program P by applying the clause permutation transformation. We can also assume the subgoals of each of the clauses of the target program P\* match the order of the subgoals of each of the clauses of the normal form program P by applying the subgoal permutation transformation. The additional subgoals created by applying the LC1 transformation have the form `append([H], T, ω)` (where `ω` is either `L` or `[A|B]`) given the following definition of `append/3`:

```

append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).

```

Unfolding the `append([H], T, ω)` subgoal twice results in the elimination of the subgoal, replacing

all occurrences of  $\omega$  with [H|T].

Now we must consider programs of the form created by #3 in the definition of decomposing/composing programs. In this case, program P must have the following form:

$$\begin{aligned} (C1) \quad & p(t_1, \dots, t_n) :- \theta_1, s(\pi(A, B, C)), \theta_2. \\ (C2) \quad & s(\pi(A, B, C)) :- q(\pi^*(A, B, C)). \end{aligned}$$

along with a definition of q which has a complexity of i minus 1. Unfolding the  $s(\pi(A, B, C))$  subgoal in the clause (C1) produces:

$$\begin{aligned} (C2) \quad & s(\pi(A, B, C)) :- q(\pi^*(A, B, C)). \\ (C3) \quad & p(t_1, \dots, t_n) :- \theta_1, q(\pi^*(A, B, C)), \theta_n. \end{aligned}$$

Since the complexity of q is i-1, the inductive hypothesis shows us that we can produce a program in the desired form.  $\square$

### 4.3 Standard Recursive Form Programs

In this section, we are ready to formally define the class of standard recursive programs. This class contains most reasonable implementations for a subset of the single pass recursive list processing programs.

**Definition.** The class of standard recursive form programs is defined inductively as follows:

1. P is a standard recursive form program if it is structural recursive and it has one of the following forms:

$$\begin{aligned} & r(t_1, t_2). \\ & r(t_3, t_4) :- s_1(\pi_1(t_3, t_5, X)), r(t_5, t_6), s_2(\pi_2(t_4, t_6, X)). \end{aligned}$$

$$\begin{aligned} & r(L, R) :- q(\pi_1(L, [], R)). \\ & q(\pi_1(t_1, R, R)). \\ & q(\pi_1(t_2, t_3, R)) :- s_1(\pi_2(t_2, t_4, X)), s_2(\pi_3(t_3, t_5, X)), q(\pi_1(t_4, t_5, R)). \end{aligned}$$

where  $s_i$  is an optional single element decomposing/composing subgoal,  $\pi_i$  is a restructuring permutation, r and q are arbitrary predicate symbols,  $s_i$  is an arbitrary predicate symbol that is different from r and q,  $t_i$  is an arbitrary Prolog term, and L, R, and X are arbitrary Prolog variables.

2. P is a standard recursive form program if it can be obtained by unfolding some subgoal in one of the clauses of a standard recursive form program
3. Nothing else is a standard recursive form program

Let's look at some examples of standard recursive form programs. The "naive" reverse/2 is one example:

```
reverse([ ], [ ]).
reverse([H|T], R) :- reverse(T, M), append(M, [H], R).
append([ ], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

its accumulator implementation:

```
reverse(L, R) :- reverse(L, [ ], R).
reverse([ ], L, L).
reverse([H|T], X, R) :- reverse(T, [H|X], R).
```

and its inverse "naive" implementation:

```
reverse([ ], [ ]).
reverse(L, [H|T]) :- append(M, [H], L), reverse(M, T).
append([ ], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

Each of these programs is equivalent in the sense that each one of them produces the same answer on any given query.

It is important to note that the class of standard recursive form programs also includes incorrect reverse/2 programs. For example, the following incorrect "naive" reverse/2 program:

```
reverse([ ], [ ]).
reverse([H|T], R) :- reverse(T, M), append(M, H, R).
append([ ], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

and the following incorrect accumulator program:

```
reverse(L, R) :- reverse(L, [ ], R).
reverse([ ], L, L).
reverse([H|T], X, R) :- enqueue(X, H, Y), reverse(T, Y, R).
enqueue([ ], E, [E]).
enqueue([H|T], X, [H|L]) :- enqueue(T, X, L).
```

and the following incorrect inverse "naive" implementation:

```
reverse([ ], [ ]).
reverse(L, R) :- append(M, [H], L), reverse(M, T), append(T, [H], R).
append([ ], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

are all standard recursive form programs. The following standard recursive normal form program

for level/2:

```
level([], []).
level([H|T], L) :- level(T, M), append(H, M, L).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

is also in the class of standard recursive form programs. This program also has an accumulator implementation:

```
level(L, R) :- level(L, [], R).
level([], L, L).
level([H|L], X, R) :- append(X, H, Y), level(T, Y, R).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

and an inverse naive implementation:

```
level([], []).
level(L, R) :- append(T, [H], L), level(T, M), append(M, H, R).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

We now define the notion of transformation that we will be applying to our standard recursive normal form programs and also to the target program which must be in standard recursive form. With the exception of accumulator and back processing transformations, The domain of these transformation is the set of all programs.

**Definition.** A transformation  $\tau$  applied to program P defining p/n is either:

1. application of the accumulator transformation to predicate p/n in P if P is in standard recursive normal form.
2. application of the back processing transformation to predicate p/n in P if P is in standard recursive normal form or accumulator normal form.
3. application of the clause permutation transformation to some pair of clauses in P
4. application of the subgoal permutation transformation to some pair of subgoals in some clause in P
5. application of the list constructor transformation to some literal in some clause in P
6. unfolding some subgoal in some clause in P
7. folding some clause  $C_1$  into some clause  $C_2 = p(t_1, \dots, t_n) :- \theta$  that was produced by

unfolding clause  $C_1$  such that the predicate symbol of the head of clause  $C_1$  does not occur elsewhere in  $P$

8. application of the associative rewrite rule to an associative subgoal
9. variable renaming (i.e., changing all occurrences of a given variable to another variable which does not exist in  $P$ )

The application of transformation  $\tau$  to program  $P$  is denoted  $P \tau$ .

**Definition.** A transformation sequence applied to program  $P$  is a sequence of transformations  $\tau_1, \dots, \tau_n$  applied to  $P$  in the following order  $((\dots ((P \tau_1) \tau_2) \dots) \tau_n)$ .

The following transformation scheme uses these transformations to produce the set of representative standard recursive form programs. This is high-level description of the algorithm employed by the ADAPT debugger described in Chapter 3. The ADAPT debugger begins with a single normal form Prolog program (e.g., "naive" reverse/2), generates a set of representative programs, and then transforms the most appropriate one into a structure that best matches with the student's program. The accumulator transformation must be applied before the back processing transformation since the domain of the accumulator transformation is restricted to standard recursive normal form programs, while the domain of the back processing transformation includes both standard recursive normal form programs and those programs produced by applying the accumulator transformation to a standard recursive normal form program. It is important to note that steps 4-5 enable ADAPT to actually accept programs that are outside of the class of standard recursive form programs. In the following description, "NF" represents the dynamic normal form program (i.e., it is like a program variable in an imperative programming language) which is potentially changed in each step.

### Transformation Scheme.

1. Eliminate indirect recursion in target program.
2. If target program still has an indirectly recursive clause then apply accumulator transformation to NF.
3. If target program decomposes the input from the back then apply back processing transformation to NF.
4. If target program has more base case clauses than NF then unfold recursive step clauses in NF until both programs have the same number of base case clauses.
5. If target program removes more than one element in its recursive clause then unfold the NF's recursive call subgoal with respect to either the original NF or the back processing NF (i.e., the program produced by applying the back processing transformation to the

original NF).

6. Apply subgoal permutation, clause permutation, and the list constructor transformation to NF if necessary.
7. Apply subgoal matching to the decompose and compose subgoal components of target program and NF (including the associativity rewrite rule).

### Characterization Theorem for **reverse/2**

For every standard recursive form **reverse/2** program with mode **reverse(+,-)** which is equivalent to "naive" **reverse/2** (for **reverse/2**), there exist transformation sequences  $\tau_1$  and  $\tau_2$  such that  $\tau_1$  applied to this target **reverse/2** program is identical to  $\tau_2$  applied to "naive" **reverse/2**:

```
reverse( [], [] ).
reverse( [H|T], R) :- reverse(T, M), append(M, [H], R).
append( [], L, L).
append( [H|T], X, [H|L]) :- append(T, X, L).
```

**Proof.** We must consider all possible **reverse/2** programs. Note that the definition of standard recursive form programs restricts the class of potential target **reverse/2** programs to the following two forms:

```
reverse(t1, t2) .
reverse(t3, t4) :- s1(π1(t3, t5, X)), reverse(t5, t6), s2(π2(t4, t6, X)) .

reverse(L, R) :- r(π1(L, [], R)) .
r(π1(t1, R, R)) .
r(π1(t2, t3, R)) :- s1(π2(t2, t4, X)), s3(π3(t3, t5, X)), r(π1(t4, t5, R)) .
```

where  $s_i$  is an optional single element decomposing/composing subgoal,  $\pi_i$  is a restructuring permutation, **reverse** and **r** are arbitrary predicate symbols,  $s_i$  is an arbitrary predicate symbol that is different from **reverse** and **r**,  $t_i$  is an arbitrary term, and **L**, **R**, and **X** are arbitrary Prolog variables.

Note that the target program must be structural recursive. We use the notation  $|X|$  to denote the length of the list **X**.  $|X|$  defines the ordering that we will be using to ensure that the target program is structural recursive.<sup>17</sup> It is important to note that we are referring to dynamic lengths of lists as opposed to static lengths of lists. Thus, when we say that  $|X| < |Y|$  we mean that the when both **X** and **Y** are instantiated with ground substitutions that the corresponding instance of **X** has a shorter length than the corresponding instance of **Y**.

---

<sup>17</sup>Note that we have factored out the restructuring permutation  $\pi_i$  from the argument list for each of the single element decomposing/composing subgoals. As such, our notion of structural recursive is the same as that of (Plümer, 1990).

Now we consider each of the legal forms of standard recursive form reverse/2 programs.

Form 1 - The program has the form:

$$\begin{aligned} & \text{reverse}(t_1, t_2) . \\ & \text{reverse}(t_3, t_4) \text{ :- } s_1(\pi_1(t_3, t_5, X)), \text{reverse}(t_5, t_6), s_2(\pi_2(t_4, t_6, X)) . \end{aligned}$$

where  $s_i$  is an optional single element decomposing/composing subgoal,  $\pi_i$  is a restructuring permutation, reverse is an arbitrary predicate symbol,  $s_i$  is an arbitrary predicate symbol that is different from reverse,  $t_i$  is an arbitrary term, and  $X$  is an arbitrary Prolog variable. Now let's consider all possibilities for each of the  $t_i$ .

Term  $t_1$  - Assume  $t_1$  is not []. Then the target program does not handle the empty list case (i.e., the query  $\text{reverse}([], X)$  will fail) or the target program produces multiple solutions for some case (i.e., some query  $\text{reverse}([A_1, \dots, A_n], X)$  can be resatisfied at least once). This follows since  $t_1$  must be either of the form  $[A_1, \dots, A_n | B]$ , a single variable  $V$ , or of the form  $f(t)$ . Let's consider each of these cases. Assume  $t_1$  is of the form  $f(t)$ . Then  $t_1$  will not unify with any valid query. Thus,  $t_1$  cannot be of the form  $f(t)$ . Assume  $t_1$  is a single variable  $V$ . Then the base case unifies with all queries (e.g.,  $\text{reverse}([], X)$  and  $\text{reverse}([a, b], X)$ ). But then the target program returns the same answer  $t_2$  for both the empty list and the two element list which is clearly incorrect since there are two possibilities to consider:  $t_2 = t_1$  (in which case the  $\text{reverse}([a, b], X)$  query produces an incorrect answer) or  $t_2 \neq t_1$  (in which case both queries produce the same answer). Thus,  $t_1$  cannot be a single variable. Assume  $t_1$  has the form  $[A_1, \dots, A_n | B]$ . Then the target program does not handle the empty list case unless the recursive clause handles it. But there is no way for the recursive clause to handle the empty list case since the length of  $t_5$  must be less than the length of  $t_3$  (which would be 0 if  $t_3$  was []) in order for the target program to be structural recursive. Obviously, it not possible for a term to represent a list of length less than 0. Thus,  $t_1$  must be [] in order for the target program to be correct.

Term  $t_2$  - Assume  $t_2$  is not []. We can assume that  $t_1$  is [] if the target program is correct. If  $t_2$  is not [] then the target program is incorrect for the query  $\text{reverse}([], X)$ . The term  $t_2$  must be [] in order for the target program to be correct.

Term  $t_3$  - Consider the possible cases for  $t_3$ . It can be either a single variable  $V$ , the empty list [], a list of the form  $[A_1, \dots, A_n | B]$ , or a term of the form  $f(t)$ . If  $t_3$  is a term of the form  $f(t)$  then the recursive clause will not unify with any query of the form  $\text{reverse}([A_1, \dots, A_n], X)$ . Obviously, this query will not unify with a valid base case so if  $t_3$  has the form  $f(t)$  then the target program does not handle queries of the form  $\text{reverse}([A_1, \dots, A_n], X)$ . Thus, we can conclude that  $t_3$  does not have the form  $f(t)$ . If  $t_3$  is the empty list [] then there is no way for the recursive clause to be structural recursive since  $t_5$  must have length less than or equal to the length of  $t_3$  (which is 0). Now consider the case when  $t_3$  is a list of the form  $[A_1, \dots, A_n | B]$ . There are two subcases to consider:  $n = 1$  and  $n > 1$ . If  $n = 1$  then  $t_3 = [A | B]$  can be unified with its corresponding term  $[H | T]$  in the head of the recursive clause of the "naive" reverse/2 program producing the substitution  $\{ A \leftarrow H, B \leftarrow T \}$ . Note that this substitution is only possible if  $A$  and  $B$  are both variables, but they must be distinct

variables. Assume either A or B is not a variable (e.g.,  $A = f(a)$  or  $B = f(b)$ ). Then the query  $\text{reverse}([c,d],X)$  would fail since  $f(a)$  is not unifiable with  $c$  and  $f(b)$  is not unifiable with  $d$ . Now assume that A and B are the same variable. Then the clause would not handle the query  $\text{reverse}([a],X)$  since  $[a]$  will not unify with  $[A|A]$ . Thus, such a program would be incorrect unless A and B are distinct variables. If  $n > 1$  then  $t_3$  will unify only with lists containing more than one element so the query  $\text{reverse}([a],X)$  would not unify with the recursive clause. Since the base case must have the form  $\text{reverse}([],[])$ , the query  $\text{reverse}([a],X)$  would fail. Thus,  $n$  must be 1 if  $t_3$  has the form  $[A_1, \dots, A_n|B]$ . In this case,  $t_3 = [A|B]$ . Finally, consider the case when  $t_3$  is a single variable V. Applying the list constructor transformation to  $[H|T]$  in the "naive" reverse/2 program produces:

```
(a) reverse([], []).
reverse(L,R):-append([H],T,L),reverse(T,M),append(M,[H],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

and

```
(b) reverse([], []).
reverse([A|B],R):-append([H],T,[A|B]),reverse(T,M),append(M,[H],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

Now  $t_3 = V$  can be unified with the corresponding term L in the newly generated normal form reverse/2 program producing the substitution  $\{ V \leftarrow L \}$ . The term  $t_3$  must be either V or  $[A|B]$  in order for the target program to be correct.

Term  $t_4$  - Consider the possible cases for  $t_4$ . It can be either a single variable V, the empty list [], a list of the form  $[A_1, \dots, A_n|B]$ , or a term of the form  $f(t)$ . If  $t_4$  is a term of the form  $f(t)$  then the recursive clause will produce an incorrect answer to any valid query. The correct answer to the query  $\text{reverse}([a],X)$  is  $\{ X \leftarrow [a] \}$  which does not unify with a term of form  $f(t)$ . Thus,  $t_4$  cannot have the form  $f(t)$ . If  $t_4$  is the empty list then the query  $\text{reverse}([A_1, \dots, A_n],X)$  will produce an incorrect answer. Now consider the case when  $t_4$  is a list of the form  $[A_1, \dots, A_n|B]$ . There are two subcases to consider:  $n = 1$  and  $n > 1$ . If  $n > 1$  then  $t_4$  will unify only with lists containing more than one element. Now consider the query  $\text{reverse}([a],X)$ . Since the base case must have the form  $\text{reverse}([],[])$ , the query  $\text{reverse}([a],X)$  will not unify with the base case clause. If the target program is correct then this query must unify with the recursive clause. But if  $t_4$  has the form  $[A_1, \dots, A_n|B]$  then there is no way for the program to produce the correct answer to this query (i.e.,  $\{ X \leftarrow [a] \}$ ) since  $[A_1, \dots, A_n|B]$  will not unify with  $[a]$  for  $n > 1$ . Thus,  $n$  must be 1 if  $t_4$  has the form  $[A_1, \dots, A_n|B]$ . In this case,  $t_4 = [A|B]$ . We can produce a corresponding term in the head of the recursive clause of a normal form reverse/2 program by applying the back processing transformation to the "naive" reverse/2 program producing the following normal form program:

```
(c) reverse([], []).
reverse(L,R):-append(T,[H],L),reverse(T,M),append([H],M,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

and then unfolding the `append([H],M,R)` subgoal in the recursive clause twice which produces the following normal form program:

```
(d) reverse([], []).
reverse(L, [H|M]) :- append(T, [H], L), reverse(T, M).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

Now  $t_4 = [A|B]$  can be unified with the corresponding term `[H|M]` in the newly generated normal form `reverse/2` program producing the substitution  $\{ A \leftarrow H, B \leftarrow M \}$ . Note that this substitution is only possible if  $A$  and  $B$  are both variables, but they must be distinct variables. Assume either  $A$  or  $B$  is not a variable (e.g.,  $A = f(a)$  or  $B = f(b)$ ). Then the query `reverse([c,d],X)` would fail since  $f(a)$  is not unifiable with  $c$  and  $f(b)$  is not unifiable with  $[d]$ . Now assume that  $A$  and  $B$  are the same variable. Then the clause would not handle the query `reverse([a],X)` since  $[a]$  will not unify with `[A|A]`. Thus, such a program would be incorrect unless  $A$  and  $B$  are different variables. The term  $t_4$  must be either  $V$  or `[A|B]` in order for the target program to be correct.

**Term  $t_5$**  - Consider the possible cases for  $t_5$ . It can be either a single variable  $V$ , the empty list `[]`, a list of the form `[A1, ..., An|B]`, or a term of the form  $f(t)$ . If  $t_5$  is a term of the form  $f(t)$  then the recursive clause will either fail or infinitely loop since  $f(t)$  will not unify `[]` which is required in order for the subgoal `reverse(t5, t6)` to terminate. Thus,  $t_5$  cannot have the form  $f(t)$ . If  $t_5$  is the empty list then the recursive call subgoal `reverse(t5, t6)` will unify with the first clause of the `reverse/2` definition forcing  $t_6$  to be unified with the empty list `[]`, or the recursive call subgoal `reverse(t5, t6)` will fail since  $t_3$  cannot be bound to the empty list `[]` as shown in the discussion of Term  $t_3$ . If the recursive call subgoal `reverse(t5, t6)` unifies with the base case clause (i.e.,  $t_6$  is unified with the empty list) then the query `reverse([a,b],Y)` will fail since there is no way create the list `[b,a]` from the lists `[a,b]` and `[]` using the single element decomposing/composing subgoal  $s_2(\pi_2(t_4, t_6, X))$ . Thus,  $t_5$  cannot be the empty list. Now consider the case when  $t_5$  is a list of the form `[A1, ..., An|B]`. If  $t_5$  has the form `[A1, ..., An|B]` then the length of  $t_5$  is greater than 0 so the program does not handle the query `reverse([a],X)` because in this case the length of  $t_3$  is 1, so the length of  $t_5$  must be less than 1 in order for the program to be structural recursive. Thus, we can conclude that  $t_5$  must be a single variable  $V$ .

**Term  $t_6$**  - Consider the possible cases for  $t_6$ . It can be either a single variable  $V$ , the empty list `[]`, a list of the form `[A1, ..., An|B]`, or a term of the form  $f(t)$ . If  $t_6$  is a term of the form  $f(t)$  then the recursive clause will always fail since  $f(t)$  will not unify with any correct answers which are in the form of a list. Thus,  $t_6$  cannot have the form  $f(t)$ . We also know that  $t_6$  cannot be the empty list `[]`. If  $t_6$  is the empty list then  $t_5$  must be unified with `[]` in order for the recursive call subgoal `reverse(t5, t6)` to produce a correct answer. This can only happen if the recursive call subgoal `reverse(t5, t6)` unifies with the base case clause. If the recursive call subgoal `reverse(t5, t6)` unifies with the base case clause (i.e.,  $t_5$  is unified with the empty list) then the query `reverse([a,b],Y)` will fail since there is no way create the list `[b,a]` from the lists `[a,b]` and `[]` using the single element decomposing/composing subgoal  $s_2(\pi_2(t_4, t_6, X))$ . Thus,  $t_6$  cannot be the empty list. Now consider the case when  $t_6$  is a list of the form `[A1, ..., An|B]`. If  $t_6$  has the form `[A1, ..., An|B]` then the length of  $t_6$  is greater than 0 so the program does not handle the query

$\text{reverse}([a],X)$  because in this case the length of  $t_3$  is 1, but the length of  $t_5$  must be less than 1 in order for the program to be structural recursive. Thus, the length of  $t_6$  would be greater than the length of  $t_5$  which is not possible. So we can conclude that  $t_6$  must be a single variable  $V$ .

Now we have looked at all the valid values for each  $t$  sub  $i$ . From this, we can conclude that the only target programs that need to be considered have one of the following forms:<sup>18</sup>

$$(1) \text{ reverse}(L,R) :- s_1(\pi_1(L,T,H)), \text{reverse}(T,M), s_2(\pi_2(R,M,H)).$$

$$(2) \text{ reverse}(L,[C|D]) :- s_1(\pi_1(L,T,H)), \text{reverse}(T,M), s_2(\pi_2([C|D],M,H)).$$

$$(3) \text{ reverse}([A|B],R) :- s_1(\pi_1([A|B],T,H)), \text{reverse}(T,M), s_2(\pi_2(R,M,H)).$$

$$(4) \text{ reverse}([A|B],[C|D]) :- s_1(\pi_1([A|B],T,H)), \text{reverse}(T,M), s_2(\pi_2([C|D],M,H)).$$

The Program Reduction Theorem for `append/3` shows us that each of these target program forms can be reduced to one that is an invocation of `append/3`. This enables us to assume that  $s_i$  is `append/3`. For standard `append/3`, the arguments of a query `append(u1, u2, u3)` are as follows:  $u_3$  is either the list that is being decomposed into  $u_1$  and  $u_2$ , or  $u_3$  is the list that is being composed from  $u_1$  and  $u_2$ . In each case, since  $s_i$  is a single element decomposing/composing subgoal we know that either  $u_1$  or  $u_2$  represents a list containing a single element that is either the first or last element of  $u_3$  and the other term is the remainder of list  $u_3$ . Assume that we want to map  $s_1(\pi_1(L,T,H))$  from the first two forms to `append(u1, u2, u3)`. Since the target program must be structural recursive, we know that  $|T| < |L|$ . Since  $s_1(\pi_1(L,T,H))$  is a single element decomposing/composing subgoal, we know that  $H$  is the single element that appears as either the first or last element of  $L$ . Thus,  $u_3 = L$  in  $s_1(\pi_1(L,T,H))$  in the first two forms. Assume we want to map  $s_1(\pi_1([A|B],T,H))$  from the last two forms to `append(u1, u2, u3)`. Since the target program must be structural recursive, we know that  $|T| < |[A|B]|$ . Since  $s_1(\pi_1([A|B],T,H))$  is a single element decomposing/composing subgoal, we know that  $H$  is the single element that appears as either the first or last element of  $[A|B]$ . Thus,  $u_3 = [A|B]$  in  $s_1(\pi_1([A|B],T,H))$  in the last two forms.

Now we consider each of the  $s_2$  subgoals individually. For the first form,  $|L| = |R|$  and  $|T| = |M|$  by the definition of `reverse/2` and  $|T| < |L|$  since the program is structural recursive. Therefore, we can conclude that  $|M| < |R|$ . Thus,  $u_3 = R$  in  $s_2(\pi_2(R,M,H))$  in the first form. For the second form,  $|L| = |[C|D]|$  and  $|T| = |M|$  by the definition of `reverse/2` and  $|T| < |L|$  since the program is structural recursive. Therefore, we can conclude that  $|M| < |[C|D]|$ . Thus,  $u_3 = [C|D]$  in  $s_2(\pi_2([C|D],M,H))$  in the second form. For the third form,  $|L| = |[A|B]|$  and  $|T| = |M|$  by the definition of `reverse/2` and  $|T| < |[A|B]|$  since the program is structural recursive. Therefore, we can conclude that  $|M| < |R|$ . Thus,  $u_3 = R$  in  $s_2(\pi_2(R,M,H))$  in the third form. For the fourth form,  $|[A|B]| = |[C|D]|$  and  $|T| = |M|$  by the definition of `reverse/2` and  $|T| < |[A|B]|$  since the program is structural recursive. Therefore, we can conclude that  $|M| < |[C|D]|$ . Thus,  $u_3 = [C|D]$  in  $s_2(\pi_2([C|D],M,H))$  in the fourth form.

---

<sup>18</sup>We use the term form italicized to represent the recursive clause of the target program. Any valid target program also contains a valid base clause (which must be the unit clause `reverse([],[])`) in order for it to be correct.

Now we argue that the variables H, T, M, L, and R must be distinct with respect to each other. By the definition of reverse/2, we know that L and R must be distinct with respect to each other and T and M must be distinct with respect to each other. By the definition of append/3, we know that H, T, and L must be distinct with respect to each other and H, M, and R must be distinct with respect to each other. Thus, it follows that H, T, M, L, and R must be distinct with respect to each other.

We must also consider whether or not A, B, H, and T must be distinct with respect to each other. We will consider only the cases when they appear in the decomposing subgoal and leave the other cases for the individual discussion of each of the four forms. These variables appear in one of the two following forms: `append([H],T,[A|B])` and `append(T,[H],[A|B])`. In the first of these forms (`append([H],T,[A|B])`), H and A can be the same variable and T and B can be the same variable. It is also possible for A, B, H, and T to all be distinct variables. For all other mappings, the query `append(X,Y,[a,b])` would fail. Likewise, in the second form (`append(T,[H],[A|B])`), the query `append(X,Y,[a,b])` would fail unless A, B, H, and T are distinct variables. An identical argument shows the same restrictions for variables C, D, H, and M.

Now, we must consider the set of cases for each of the possible forms given above. Let's consider each of them in turn. Consider the first form:

(1) `reverse(L,R):-s1( $\pi_1$ (L,T,H)),reverse(T,M),s2( $\pi_2$ (R,M,H))`.

We can immediately see that it is not possible to omit the  $s_1(\pi_1(L,T,H))$  subgoal and keep the program structural recursive since T would be unbound. It is also not possible to omit the  $s_2(\pi_2(R,M,H))$  subgoal since that would mean that the clause's output argument R would remain unbound which is clearly an incorrect program. This leaves us with 4 possibilities for the first form:

(1a) `reverse(L,R):-append([H],T,L),reverse(T,M),append(M,[H],R)`.

(1b) `reverse(L,R):-append(T,[H],L),reverse(T,M),append([H],M,R)`.

(1c) `reverse(L,R):-append([H],T,L),reverse(T,M),append([H],M,R)`.

(1d) `reverse(L,R):-append(T,[H],L),reverse(T,M),append(M,[H],R)`.

Form (1a) is identical to normal form (a) so it is obviously correct. Form (1b) is identical to normal form (c) so it also is correct. Forms (1c) and (1d) are incorrect, however, since they produce the incorrect answer `X = [a,b]` to the query `reverse([a,b],X)`.

Now consider the second form:

(2) `reverse(L,[C|D]):-s1( $\pi_1$ (L,T,H)),reverse(T,M),s2( $\pi_2$ ([C|D],M,H))`.

We can immediately see that it is not possible to omit the  $s_1 (\pi_1 (L,T,H))$  subgoal and keep the program structural recursive since T would be unbound. This leaves us with 6 possibilities for the second form:

(2a) `reverse(L,[C|D]):-append(T,[H],L),reverse(T,M).`

(2b) `reverse(L,[C|D]):-append([H],T,L),reverse(T,M).`

(2c) `reverse(L,[C|D]):-append([H],T,L),reverse(T,M),append(M,[H],[C|D]).`

(2d) `reverse(L,[C|D]):-append(T,[H],L),reverse(T,M),append(M,[H],[C|D]).`

(2e) `reverse(L,[C|D]):-append([H],T,L),reverse(T,M),append([H],M,[C|D]).`

(2f) `reverse(L,[C|D]):-append(T,[H],L),reverse(T,M),append([H],M,[C|D]).`

The only way for forms (2a) and (2b) to be correct is by making the variables C and D the same as some other variables occurring in the clause. Otherwise, the answer to the query `reverse([a],X)` would be incorrect since X would remain unbound. Mapping C identical to H and D identical to T makes form (2a) identical to normal form (d) so it is obviously correct. Any other mapping for C and D creates an incorrect program. Likewise, any mapping for C and D in form (2b) creates an incorrect program. Form (2c) is a correct program. This can be shown by applying the LC2 transformation to the `append(M,[H],R)` subgoal in `reverse/2`'s recursive clause normal form (a) producing:

```
(e) reverse([], []).
    reverse(L, [C|D]) :- append([H], T, L), reverse(T, M), append(M, [H], [C|D]).
    append([], L, L).
    append([H|T], X, [H|L]) :- append(T, X, L).
```

which is identical to form (2c). Forms (2d) and (2e) are incorrect, however, since they produce the incorrect answer `X = [a,b]` to the query `reverse([a,b],X)`. Finally, form (2f) is correct. This can be shown by applying the LC1 transformation to `[H|M]` in normal form (d) which produces two programs. One the programs is identical to normal form (c) and the other:

```
(f) reverse([], []).
    reverse(L, [C|D]) :- append(T, [H], L), reverse(T, M), append([H], M, [C|D]).
    append([], L, L).
    append([H|T], X, [H|L]) :- append(T, X, L).
```

is identical to form (2f).

Now let's consider the third form:

(3) `reverse([A|B],R):-s1 ( $\pi_1$  ([A|B],T,H)),reverse(T,M),s2 ( $\pi_2$  (R,M,H)).`

We immediately see that it is also not possible to omit the  $s_2 (\pi_2 (R,M,H))$  subgoal since that would

mean that the clause's output argument R would remain unbound which is clearly an incorrect program. This leaves us with 6 possibilities for the third form:

(3a) `reverse([A|B],R):-reverse(T,M),append(M,[H],R).`

(3b) `reverse([A|B],R):-reverse(T,M),append([H],M,R).`

(3c) `reverse([A|B],R):-append([H],T,[A|B]),reverse(T,M),append(M,[H],R).`

(3d) `reverse([A|B],R):-append(T,[H],[A|B]),reverse(T,M),append(M,[H],R).`

(3e) `reverse([A|B],R):-append([H],T,[A|B]),reverse(T,M),append([H],M,R).`

(3f) `reverse([A|B],R):-append(T,[H],[A|B]),reverse(T,M),append([H],M,R).`

The only way for forms (3a) and (3b) to be correct is by making the variables A and B the same as some other variables occurring in the clause. The variable T must be the same as either A or B in order for the program to be structural recursive. If T is the same as A then the query `reverse([a],X)` would fail so T must be the same as B. If A is still not mapped to some other variable then the answer to the query `reverse([a],X)` would be incorrect. Mapping A identical to H makes form (3a) identical to "naive" `reverse/2` so it is obviously correct. Any other mapping for A creates an incorrect program. Likewise, any mapping for A in form (3b) creates an incorrect program. Form (3c) is identical to normal form (b) so it is correct. Form (3d) and (3e) are incorrect, however, since they produce the incorrect answer `X = [a,b]` to the query `reverse([a,b],X)`. Form (3f) is a correct program. This can be shown by applying the LC2 transformation to the `append(T,[H],L)` subgoal in `reverse/2`'s recursive clause normal form (c) producing:

```
(g) reverse([], []).
    reverse([A|B], R) :- append(T, [H], [A|B]), reverse(T, M), append([H], M, R).
    append([], L, L).
    append([H|T], X, [H|L]) :- append(T, X, L).
```

which is identical to form (3f).

Now let's consider the fourth and final form:

(4) `reverse([A|B],[C|D]):-s1(π1([A|B],T,H)),reverse(T,M),s2(π2([C|D],M,H)).`

We have 9 possibilities for the fourth form:

(4a) `reverse([A|B],[C|D]):-reverse(T,M).`

(4b) `reverse([A|B],[C|D]):-reverse(T,M),append([H],M,[C|D]).`

(4c) `reverse([A|B],[C|D]):-reverse(T,M),append(M,[H],[C|D]).`

(4d) `reverse([A|B],[C|D]):-append(T,[H],[A|B]),reverse(T,M).`

(4e) `reverse([A|B],[C|D]):-append([H],T,[A|B]),reverse(T,M).`

(4f) `reverse([A|B],[C|D]):-append([H],T,[A|B]),reverse(T,M),append(M,[H],[C|D]).`

(4g) `reverse([A|B],[C|D]):-append(T,[H],[A|B]),reverse(T,M),append(M,[H],[C|D]).`

(4h) `reverse([A|B],[C|D]):-append([H],T,[A|B]),reverse(T,M),append([H],M,[C|D]).`

(4i) `reverse([A|B],[C|D]):-append(T,[H],[A|B]),reverse(T,M),append([H],M,[C|D]).`

The only way for forms (4a) through (4c) to be correct is by making the variables A and B the same as some other variables occurring in the clause. The variable T must be the same as either A or B in order for the program to be structural recursive. If T is the same as A then the query `reverse([a],X)` would fail so T must be the same as B. If A is still not mapped to some other variable then the answer to the query `reverse([a],X)` would be incorrect. Likewise, the only way for forms (4a), (4d), and (4e) to be correct is by making the variables C and D the same as some other variables occurring in the clause. There is no mapping for A, C, and D that create correct programs for forms (4a), (4b), and (4e). We can make form (4c) correct if we map A to H and map B to T. Otherwise, these programs produce incorrect answers to the query `reverse([a,b],X)`. We can show that form (4c) is a correct program by applying the LC2 transformation to the `append(M,[H],R)` subgoal in "naive" `reverse/2`'s recursive clause producing:

```
(h) reverse([], []).
    reverse([H|T],[C|D]):-reverse(T,M),append(M,[H],[C|D]).
    append([],L,L).
    append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (4c). We can make form (4d) correct if we map C to H and map D to M. Otherwise, these programs produce incorrect answers to the query `reverse([a,b],X)`. We can show that form (4d) is a correct program by applying the LC2 transformation to the `append(T,[H],L)` subgoal in `reverse/2`'s recursive clause normal form (d) producing:

```
(i) reverse([], []).
    reverse([A|B],[H|M]):-append(T,[H],[A|B]),reverse(T,M).
    append([],L,L).
    append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (4d). We can show that form (4f) is a correct program by applying the LC2 transformation to the `append(M,[H],R)` subgoal in `reverse/2`'s recursive clause normal form (b) producing:

```
(j) reverse([], []).
    reverse([A|B],[C|D]):-append([H],T,[A|B]),reverse(T,M),append(M,[H],[C|D]).
    append([],L,L).
    append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (4f). Forms (4g) and (4h) are incorrect, however, since they both produce incorrect answers to the query `reverse([a,b],X)`. We can show that form (4i) is a correct program by applying the LC2 transformation to the `append(T,[H],L)` subgoal in `reverse/2`'s recursive clause normal form (f) producing:

```
(k) reverse([], []).
reverse([A|B], [C|D]) :- append(T, [H], [A|B]), reverse(T, M), append([H], M, [C|D]).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

Form 2 - The program has the form:

```
reverse(L, R) :- r(π1(L, [], R)).
r(π1(t1, R, R)).
r(π1(t2, t3, R)) :- s1(π2(t2, t4, X)), s3(π3(t3, t5, X)), r(π1(t4, t5, R)).
```

where  $s_i$  is an optional single element decomposing/composing subgoal,  $\pi_i$  is a restructuring permutation, `reverse` and `r` are arbitrary predicate symbols,  $s_i$  is an arbitrary predicate symbol that is different from `reverse` and `r`,  $t_i$  is an arbitrary term, and `X` is an arbitrary Prolog variable. Now let's consider all possibilities for each of the  $t_i$ . Note that the query `reverse(u1, u2)` is replaced by the subgoal `r(π1(u1, [], u2))`. We will drop the  $\pi_1$  for readability since it occurs uniformly throughout and refer to the "query" `r(u1, [], u2)` whenever we are actually meaning the original query `reverse(u1, u2)`. The "query" `r(u1, [], u2)` can be generalized to `r(u1, X, u2)`. Before we begin considering the legal values for each of the terms, we must first consider an accumulator `reverse/2` program that can be obtained by applying the accumulator transformation to the "naive" `reverse/2` program producing:

```
(l) reverse(L, R) :- reverse(L, [], R).
reverse([], X, X).
reverse([H|T], X, R) :- append([H], X, Y), reverse(T, Y, R).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

Unfolding the `append([H],X,Y)` subgoal in the recursive clause twice and eliminating the no longer necessary definition of `append/3` produces the following normal form program:

```
(m) reverse(L, R) :- reverse(L, [], R).
reverse([], X, X).
reverse([H|T], X, R) :- reverse(T, [H|X], R).
```

which is the "standard" accumulator `reverse/2` implementation. Applying the LC1 transformation to `[H|T]` in this program produces:

```
(n) reverse(L, R) :- reverse(L, [], R).
reverse([], X, X).
reverse(L, X, R) :- append([H], T, L), reverse(T, [H|X], R).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

and

```
(o) reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([A|B],X,R):-append([H],T,[A|B]),reverse(T,[H|X],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

Term  $t_1$  - Assume  $t_1$  is not  $[]$ . Then the target program does not handle the empty list case (i.e., the query  $r([],[],X)$  will fail) or the target program produces multiple solutions for some case (i.e., some query  $r([A_1, \dots, A_n],[],X)$  can be resatisfied at least once). This follows since  $t_1$  must be either of the form  $[A_1, \dots, A_n | B]$ , a single variable  $V$ , or of the form  $f(t)$ . Let's consider each of these cases. Assume  $t_1$  is of the form  $f(t)$ . Then  $t_1$  will not unify with any valid query. Thus,  $t_1$  cannot be of the form  $f(t)$ . Assume  $t_1$  is a single variable  $V$ . Then the base case unifies with all queries (e.g.,  $r([],[],X)$  and  $r([a,b],[],X)$ ). Note that since the schema requires the second and third argument positions to be unified with each other, the answer to each of these queries is the same. The answer for both queries is  $[]$  which is correct for the first query and incorrect for the second query. Thus,  $t_1$  cannot be a single variable. Now assume that  $t_1$  is of the form  $[A_1, \dots, A_n | B]$ . If we assume that the base case clause is responsible for the empty list case then the target program does not handle the query  $r([],[],X)$ . If we assume that the base case clause is not responsible for the empty list case then the empty list case must be handled by the recursive clause. But there is no way for the recursive clause to handle the empty list case since the length of  $t_4$  must be less than the length of  $t_2$  (which would be 0 if  $t_2$  was  $[]$ ) in order for the target program to be structural recursive. Obviously, it not possible for a term to represent a list of length less than 0. Since there are no other possibilities for  $t_1$ , we can conclude that  $t_1$  must be  $[]$  in order for the target program to be correct.

Term  $t_2$  - Consider the possible cases for  $t_2$ . It can be either a single variable  $V$ , the empty list  $[]$ , a list of the form  $[A_1, \dots, A_n | B]$ , or a term of the form  $f(t)$ . If  $t_2$  is a term of the form  $f(t)$  then the recursive clause will not unify with any query of the form  $r([A_1, \dots, A_n],X,Y)$ . Obviously, this query will not unify with a valid base case so if  $t_2$  has the form  $f(t)$  then the target program does not handle queries of the form  $r([A_1, \dots, A_n],[],X)$  so we can conclude that  $t_2$  does not have the form  $f(t)$ . If  $t_2$  is the empty list  $[]$  then there is no way for the recursive clause to be structural recursive since  $t_4$  must have length less than the length of  $t_2$  (which is 0). Now consider the case when  $t_2$  is a list of the form  $[A_1, \dots, A_n | B]$ . There are two subcases to consider:  $n = 1$  and  $n > 1$ . If  $n = 1$  then  $t_2 = [A|B]$  can be unified with its corresponding term  $[H|T]$  in the head of the recursive clause of the normal form (I) producing the substitution  $\{A \leftarrow H, B \leftarrow T\}$ . Note that this substitution is only possible if  $A$  and  $B$  are both variables, but they must be distinct variables. Assume either  $A$  or  $B$  is not a variable (e.g.,  $A = f(a)$  or  $B = f(b)$ ). Then the query  $reverse([c,d],X)$  would fail  $f(a)$  is not unifiable with  $c$  and  $f(b)$  is not unifiable with  $[d]$ . Now assume that  $A$  and  $B$  are the same variable. Then the clause would not handle the query  $r([a],[],X)$  since  $[a]$  will not unify with  $[A|A]$ . Thus, such a program would be incorrect unless  $A$  and  $B$  are different variables. If  $n > 1$  then  $t_2$  will only unify with lists containing more than one element so the query  $r([a],[],X)$  would not unify with the recursive clause. Since the base case must have the form  $r([],Z,Z)$ , the query  $r([a],[],X)$  would fail. Thus,  $n$  must be 1 if  $t_2$  has the form  $[A_1, \dots, A_n | B]$ . Finally, consider the case when  $t_2$  is a single variable  $V$ . Applying the list constructor transformation to  $[H|T]$  in normal form (I) produces:

```
(p) reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse(L,X,R):-append([H],T,L),append([H],X,Y),reverse(T,Y,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

and

```
(q) reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([A|B],X,R):-append([H],T,[A|B]),append([H],X,Y),reverse(T,Y,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

Now  $t_2 = V$  can be unified with the corresponding term  $L$  in normal form (p) producing the substitution  $\{ V \leftarrow L \}$ . The term  $t_2$  must be either  $V$  or  $[A|B]$  in order for the target program to be correct.

Term  $t_3$  - Consider the possible cases for  $t_3$ . We argue that  $t_3$  must be a single variable  $V$ . It is possible to have the queries  $r([a],[],X)$  or  $r([a],[b],X)$ . Since  $t_3$  must be able to unify with the empty list  $[]$  and non-empty lists (e.g.,  $[a]$ ), neither the empty list  $[]$ , an argument of the form  $[A_1, \dots, A_n | B]$ , nor an argument of the form  $f(t)$  would unify with both of these queries. Thus, we can conclude that  $t_3$  must be a single variable  $V$ .

Term  $t_4$  - Consider the possible cases for  $t_4$ . It can be either a single variable  $V$ , the empty list  $[]$ , a list of the form  $[A_1, \dots, A_n | B]$ , or a term of the form  $f(t)$ . If  $t_4$  is a term of the form  $f(t)$  then the recursive clause will always fail since  $f(t)$  will not unify with any correct answer from the  $s_1$  ( $\pi_2$  ( $t_2, t_4, X$ )) subgoal. Thus,  $t_4$  cannot have the form  $f(t)$ . If  $t_4$  is the empty list then the recursive call subgoal  $r(t_4, t_5, R)$  will unify with the first clause of the  $r/3$  definition forcing  $t_5$  to be unified with  $R$ , or the recursive call subgoal  $r(t_4, t_5, R)$  will fail since  $t_2$  cannot be bound to the empty list  $[]$  as shown in the discussion of Term  $t_2$ . If it unifies with the base case clause then  $R$  is unified with  $t_5$ . It is possible to have the queries  $r([a],[],Y)$  and  $r([a],[b],[],Y)$ , but if  $t_4$  is the empty list then both of these queries would have the same answer. Thus,  $t_4$  cannot be the empty list. Now consider the case when  $t_4$  is a list of the form  $[A_1, \dots, A_n | B]$ . If  $t_4$  has the form  $[A_1, \dots, A_n | B]$  then the length of  $t_4$  is greater than 0 so the program does not handle the query  $r([a],[],X)$  because in this case the length of  $t_2$  is 1, so the length of  $t_4$  must be less than 1 in order for the program to be structural recursive. Thus, we can conclude that  $t_4$  must be a single variable  $V$ .

Term  $t_5$  - Consider the possible cases for  $t_5$ . It can be either a single variable  $V$ , the empty list  $[]$ , a list of the form  $[A_1, \dots, A_n | B]$ , or a term of the form  $f(t)$ . Consider the query  $r([a],[],X)$ . This query will not unify with the base case clause since  $t_1$  must be the empty list in order for the target program to be correct. In order for the target program to be structural recursive,  $|t_4| < |t_2|$ . In this case,  $t_4$  (which must be a single variable as shown in the discussion for Term  $t_4$ ) must be bound to the empty list  $[]$  in the recursive call subgoal  $r(t_4, t_5, R)$ . This subgoal unifies with the base case clause producing the substitution  $\{ R \leftarrow t_5 \}$ . In this case, the correct answer is  $\{ R \leftarrow [a] \}$  so we can conclude that  $t_5$  cannot be  $f(t)$ , the empty list  $[]$ , or a list of the form  $[A_1, \dots, A_n | B]$  for  $n > 1$ . The term  $t_5$  must be either  $V$  or  $[A|B]$  (where  $A$  and  $B$  are distinct variables as was shown for Term  $t_2$ ).

in order for the target program to be correct.

Now we have looked at all the valid values for each  $t$  sub  $i$ . We can further restrict the possibilities for target programs by replacing the clause:

$$\text{reverse}(L,R) :- r(\pi_1(L,[],R)).$$

with:

$$\text{reverse}(L,R) :- q(L,[],R).$$

and adding the clause:

$$q(A,B,C) :- r(\pi_1(A,B,C)).$$

to the target program. Clearly, this new target program is equivalent to the original target program for `reverse/2` since we can simply unfold the subgoal  $q(L,[],R)$  in the clause  $\text{reverse}(L,R) :- q(L,[],R)$  producing the original target program. Note, however, that if we unfold the  $r(\pi_1(A,B,C))$  subgoal in the clause  $q(A,B,C) :- r(\pi_1(A,B,C))$  then we would produce the following set of clauses:

$$\begin{aligned} & q(t_1, R, R). \\ & q(t_2, t_3, R) :- s_1(\pi_2(t_2, t_4, X)), s_2(\pi_3(t_3, t_5, X)), r(\pi_1(t_4, t_5, R)). \end{aligned}$$

Finally, folding the clause  $q(A,B,C) :- r(\pi_1(A,B,C))$  into the clause  $q(t_2, t_3, R) :- s_1(\pi_2(t_2, t_4, X)), s_2(\pi_3(t_3, t_5, X)), r(\pi_1(t_4, t_5, R))$  produces:

$$\begin{aligned} & q(t_1, R, R). \\ & q(t_2, t_3, R) :- s_1(\pi_2(t_2, t_4, X)), s_2(\pi_3(t_3, t_5, X)), q(t_4, t_5, R). \end{aligned}$$

Using this target program form, we can conclude that the only target programs that need to be considered have one of the following forms (where we have replaced the predicate  $q/3$  with  $r/3$  for consistency):

- (1)  $r(L,A,R) :- s_1(\pi_2(L,T,H)), s_2(\pi_3(A,B,H)), r(T,B,R).$
- (2)  $r(L,A,R) :- s_1(\pi_2(L,T,H)), s_2(\pi_3(A,[E|F],H)), r(T,[E|F],R).$
- (3)  $r([C|D],A,R) :- s_1(\pi_2([C|D],T,H)), s_2(\pi_3(A,B,H)), r(T,B,R).$
- (4)  $r([C|D],A,R) :- s_1(\pi_2([C|D],T,H)), s_2(\pi_3(A,[E|F],H)), r(T,[E|F],R).$

The Program Reduction Theorem for `append/3` shows us that each of these target program forms can be reduced to one that is an invocation of `append/3`. This enables us to assume that  $s_1$  is `append/3`. For standard `append/3`, the arguments of a query `append(u1, u2, u3)` are as follows:  $u_3$

is either the list that is being decomposed into  $u_1$  and  $u_2$ , or  $u_3$  is the list that is being composed from  $u_1$  and  $u_2$ . In each case, since  $s_i$  is a single element decomposing/composing subgoal we know that either  $u_1$  or  $u_2$  represents a list containing a single element that is either the first or last element of  $u_3$  and the other term is the remainder of list  $u_3$ . Assume that we want to map  $s_1(\pi_1(L,T,H))$  from the first two forms to  $\text{append}(u_1, u_2, u_3)$ . Since the target program must be structural recursive, we know that  $|T| < |L|$ . Since  $s_1(\pi_1(L,T,H))$  is a single element decomposing/composing subgoal, we know that  $H$  is the single element that appears as either the first or last element of  $L$ . Thus,  $u_3 = L$  in  $s_1(\pi_1(L,T,H))$  in the first two forms. Assume we want to map  $s_1(\pi_1([C|D],T,H))$  from the last two forms to  $\text{append}(u_1, u_2, u_3)$ . Since the target program must be structural recursive, we know that  $|T| < |[C|D]|$ . Since  $s_1(\pi_1([C|D],T,H))$  is a single element decomposing/composing subgoal, we know that  $H$  is the single element that appears as either the first or last element of  $[C|D]$ . Thus,  $u_3 = [C|D]$  in  $s_1(\pi_1([C|D],T,H))$  in the last two forms.

Assume that we want to map  $s_2(\pi_3(A,B,H))$  from the first and third forms to  $\text{append}(u_1, u_2, u_3)$ . Since we know that  $H$  is a single element,  $A$  is a list (which is bound to  $[]$  on the initial query), and  $s_2$  is a valid single element decomposing/composing subgoal, we know that  $B$  must be composed from  $H$  and  $A$ . This follows since there is no way to decompose  $A$  in the initial case when it is bound to the empty list  $[]$ . Thus,  $u_3 = B$  in the first and third forms. Now assume that we want to map  $s_2(\pi_3(A,[E|F],H))$  from the second and fourth forms to  $\text{append}(u_1, u_2, u_3)$ . Since we know that  $H$  is a single element,  $A$  is a list (which is bound to  $[]$  on the initial query), and  $s_2$  is a valid single element decomposing/composing subgoal, we know that  $[E|F]$  must be composed from  $H$  and  $A$ . Thus,  $u_3 = [E|F]$  in the second and fourth forms.

The recursive clause of all of our normal form programs of the second standard recursive form removes a single element from the front of the input list. It is possible to produce an accumulator reverse/2 program which removes elements from the back of the input list by applying the back processing transformation to normal form (1) producing the following normal form program:

```
(r) reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse(L,X,R):-append(T,[H],L),append(X,[H],Y),reverse(T,Y,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

With this normal form, we are ready to consider the cases for each of the possible forms given above. Let's consider each of them in turn. Consider the first form:

(1)  $r(L,A,R):-s_1(\pi_2(L,T,H)),s_2(\pi_3(A,B,H)),r(T,B,R)$ .

Recall that in order for a program to be structural recursive, each of its recursive clauses must have at least one input argument position that is strictly decreasing. In the case of accumulator reverse/3, the first two argument positions are the inputs and the third argument position is an output. Since the second argument position (which holds the accumulator) is initially bound to the empty list  $[]$ , it is not possible for the second argument position to be strictly decreasing. If the  $s_1(\pi_2(L,T,H))$  subgoal is omitted from the first form then the variable  $T$  in the recursive call subgoal  $r(T,B,R)$  would be unbound. If  $T$  is unbound in the recursive call subgoal  $r(T,B,R)$  then the first argument

position is not strictly decreasing. Thus, we can conclude that is not possible to omit the  $s_1 (\pi_2 (L,T,H))$  subgoal from the first form. It is also not possible to omit the  $s_2 (\pi_3 (A,B,H))$  subgoal from the first form. If we unfold the first form (with the  $s_2 (\pi_3 (A,B,H))$  subgoal omitted) with respect to the recursive call subgoal  $r(T,B,R)$  then we would get the following set of clauses (with the base case clause added to the first form):

$$\begin{aligned} & r([],R,R). \\ & r(L,R,R) :- s_1 (\pi_2 (L,[],H)). \\ & r(L,A,R) :- s_1 (\pi_2 (L, T_1, H_1)), s_1 (\pi_2 (T_1, T_2, H_2)), r(T_2, B, R). \end{aligned}$$

Note that  $r/3$ 's output argument (i.e., the third argument position) is left unbound in the second clause. The answer to the query  $r([a],[],X)$  would be  $\{ \}$  which is clearly incorrect. Thus, both  $s_1 (\pi_2 (L,T,H))$  and  $s_2 (\pi_3 (A,B,H))$  must be present in the first form. This leaves us with 4 possibilities for the first form:

(1a)  $r(L,A,R):-append([H],T,L),append(A,[H],B),r(T,B,R).$

(1b)  $r(L,A,R):-append(T,[H],L),append([H],A,B),r(T,B,R).$

(1c)  $r(L,A,R):-append([H],T,L),append([H],A,B),r(T,B,R).$

(1d)  $r(L,A,R):-append(T,[H],L),append(A,[H],B),r(T,B,R).$

Forms (1a) and (1b) are incorrect since they produce the incorrect answer  $X \leftarrow [a,b]$  to the query  $r([a,b],[],X)$ . Form (1c) is identical to normal form (p) so it is obviously correct. Form (1d) is identical to normal form (r) so it is also correct.

Now consider the second form:

$$(2) r(L,A,R):-s_1 (\pi_2 (L,T,H)),s_2 (\pi_3 (A,[E|F],H)),r(T,[E|F],R).$$

We can immediately see that it is not possible to omit the  $s_1 (\pi_2 (L,T,H))$  subgoal and keep the program structural recursive since  $T$  would be unbound. This leaves us with 6 possibilities for the second form:

(2a)  $r(L,A,R):-append([H],T,L),r(T,[E|F],R).$

(2b)  $r(L,A,R):-append(T,[H],L),r(T,[E|F],R).$

(2c)  $r(L,A,R):-append([H],T,L),append(A,[H],[E|F]),r(T,[E|F],R).$

(2d)  $r(L,A,R):-append(T,[H],L),append(A,[H],[E|F]),r(T,[E|F],R).$

(2e)  $r(L,A,R):-append([H],T,L),append([H],A,[E|F]),r(T,[E|F],R).$

(2f)  $r(L,A,R):-append(T,[H],L),append([H],A,[E|F]),r(T,[E|F],R).$

The only way for forms (2a) and (2b) to be correct is by making the variables E and F the same as some other variables occurring in the clause. The variable A must be the same as F and the variable H must be the same as E in order for the program to produce the correct answer for query  $reverse([a],[],X)$ . This mapping makes form (2a) identical to normal form (n) so it is obviously correct. Any other mapping creates an incorrect program. Likewise, any mapping for form (2b) creates an incorrect program. Forms (2c) and (2d) are incorrect. They both fail on the query  $r([a,b],[],X)$  since it is impossible to unify  $[b,a]$  with  $[a,b]$ . Form (2e) is correct. This can be shown by applying the LC1 transformation to  $[H|X]$  in normal form (n) produces two programs. One of the programs is identical to normal form (p) and the other program has the form:

```
(s) reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse(L,X,R):-append([H],T,L),append([H],X,[C|D]),reverse(T,[C|D],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (2e). Finally, form (2f) is incorrect since it produces the incorrect answer  $X = [a,b]$  to the query  $r([a,b],[],X)$ .

Now let's consider the third form:

(3)  $r([C|D],A,R):-s_1(\pi_2([C|D],T,H)),s_2(\pi_3(A,B,H)),r(T,B,R).$

We immediately see that it is also not possible to omit the  $s_2(\pi_3(R,M,H))$  subgoal as was shown for the first form. This leaves us with 6 possibilities for the third form:

(3a)  $r([C|D],A,R):-append([H],A,B),r(T,B,R).$

(3b)  $r([C|D],A,R):-append(A,[H],A,B),r(T,B,R).$

(3c)  $r([C|D],A,R):-append([H],T,[C|D]),append(A,[H],B),r(T,B,R).$

(3d)  $r([C|D],A,R):-append(T,[H],[C|D]),append(A,[H],B),r(T,B,R).$

(3e)  $r([C|D],A,R):-append([H],T,[C|D]),append([H],A,B),r(T,B,R).$

(3f)  $r([C|D],A,R):-append(T,[H],[C|D]),append([H],A,B),r(T,B,R).$

The only way for forms (3a) and (3b) to be correct is by making the variables C and D the same as some other variables occurring in the clause. The variable T must be the same as either C or D in order for the program to be structural recursive. If T is the same as C then the query  $reverse([a],[],X)$  would fail so T must be the same as D. If C is still not mapped to some other variable then the answer to the query  $reverse([a],[],X)$  would be incorrect. Mapping C identical to H makes form (3a) identical to normal form (l) so it is obviously correct. Any other mapping for

C creates an incorrect program. Likewise, any mapping for C in form (3b) creates an incorrect program. Forms (3c) and (3f) are incorrect since they both produce the incorrect answer  $X - [a,b]$  to the query  $r([a,b],[],X)$ . We can show that form (3d) is a correct program by applying the LC2 transformation to the  $append(T,[H],L)$  subgoal in  $reverse/2$ 's recursive clause normal form (r) producing:

```
(t)  reverse(L,R):-reverse(L,[],R).
      reverse([],X,X).
      reverse([A|B],X,R):-append(T,[H],[A|B]),append(X,[H],Y),reverse(T,Y,R).
      append([],L,L).
      append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (3d). Form (3e) is also correct since it is identical to normal form (q).

Now let's consider the fourth and final form:

(4)  $r([C|D],A,R):-s_1(\pi_2([C|D],T,H),s_2(\pi_3(A,[E|F],H)),r(T,[E|F],R))$ .

We have 9 possibilities for the fourth form:

(4a)  $r([C|D],A,R):-r(T,[E|F],R)$ .

(4b)  $r([C|D],A,R):-append(A,[H],[C|D]),r(T,[E|F],R)$ .

(4c)  $r([C|D],A,R):-append([H],A,[C|D]),r(T,[E|F],R)$ .

(4d)  $r([C|D],A,R):-append([H],T,[C|D]),r(T,[E|F],R)$ .

(4e)  $r([C|D],A,R):-append(T,[H],[C|D]),r(T,[E|F],R)$ .

(4f)  $r([C|D],A,R):-append([H],T,[C|D]),append(A,[H],[E|F]),r(T,[E|F],R)$ .

(4g)  $r([C|D],A,R):-append(T,[H],[C|D]),append(A,[H],[E|F]),r(T,[E|F],R)$ .

(4h)  $r([C|D],A,R):-append([H],T,[C|D]),append([H],A,[E|F]),r(T,[E|F],R)$ .

(4i)  $r([C|D],A,R):-append(T,[H],[C|D]),append([H],A,[E|F]),r(T,[E|F],R)$ .

The only way for forms (4a) through (4c) to be correct is by making the variables C and D the same as some other variables occurring in the clause. The variable T must be the same as either C or D in order for the program to be structural recursive. If T is the same as C then the query  $reverse([a],[],X)$  would fail so T must be the same as D. If C is still not mapped to some other variable then the answer to the query  $reverse([a],[],X)$  would be incorrect. Mapping C identical to E and F to A makes form (4a) identical to normal form (m) so it is obviously correct. Any other mapping for C creates an incorrect program. Likewise, any mapping for C in form (4b) creates an incorrect program. Applying the LC1 to  $[H|X]$  to normal form (m) produces a program that is

identical to normal form (l) and the following program:

```
(u) reverse(L,R):-reverse(L,[],R).
    reverse([],X,X).
    reverse([H|T],X,R):-append([H],X,[C|D]),reverse(T,[C|D],R).
    append([],L,L).
    append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (4c) as long as C is mapped to A and F is mapped to A. Any other mapping for C creates an incorrect program. Form (4d) is also correct since it is identical to normal form (o) as long as F is mapped to A. Forms (4e), (4f), and (4i) are incorrect since they each produce the incorrect answer  $X \leftarrow [a,b]$  to the query  $r([a,b],[],X)$ . We can show that form (4g) is a correct program by applying the LC2 transformation to the  $\text{append}(X,[H],Y)$  subgoal in  $\text{reverse}/2$ 's recursive clause normal form (t) producing:

```
(v) reverse(L,R):-reverse(L,[],R).
    reverse([],X,X).
    reverse([A|B],X,R):-append(T,[H],[A|B]),append(X,[H],[C|D]),reverse(T,[C|D],R).
    append([],L,L).
    append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (4g). Applying the LC1 to  $[H|X]$  to normal form (o) produces a program that is identical to normal form (q) and the following program:

```
(w) reverse(L,R):-reverse(L,[],R).
    reverse([],X,X).
    reverse([A|B],X,R):-append([H],T,[A|B]),append([H],M,[C|D]),reverse(T,[C|D],R).
    append([],L,L).
    append([H|T],X,[H|L]):-append(T,X,L).
```

which is identical to form (4h).

We have exhausted the possibilities for target programs and have shown that every correct target program can be transformed into one that is identical to "naive"  $\text{reverse}/2$  or one of the normal forms (a) - (w).  $\square$

## 4.4 Constructing Intelligent Tutoring Systems

Characterizing the domain of recognizable programs is a vital step in the construction of an intelligent tutoring system (ITS) for programming. The construction of the diagnosis component of an intelligent tutoring system requires three steps:

1. Conduct empirical studies to determine the range of potential solutions that students might produce.
2. Construct the diagnosis component of the ITS.
3. Conduct formal analysis to characterize the domain that the diagnosis component can

recognize.

Although empirical studies are necessary for the construction of an ITS, they are insufficient to actually evaluate the effectiveness of an ITS. For APROPOS2, Looi (1988) conducted empirical results gathering student solutions to the reverse/2 problem. His results are given in Appendix B along with an additional set of reverse/2 programs that have been tested on the ADAPT debugger. In studying these results, one can see that students produce programs that:

1. remove a single element on each pass from either the front or the back of the input list
2. contain a single recursive clause
3. do not contain mutual recursion
4. use at most one subgoal to decompose the input list (compose the output list)
5. use increasing arguments (e.g., list arguments which have additional elements on each successive pass) only for result accumulators

These are the same basic restrictions<sup>19</sup> that the class of standard recursive form programs imposes on the student. In order to justify the robustness of this class, 10 of the more than 40 unique correct implementations of reverse/2 that are provided in Appendix B are given below.

Standard "naive" reverse/2 (ADAPT's normal form program) is obviously within the class of standard recursive form programs (variation #17 in Appendix B):

```
reverse([ ], [ ]).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, [H], Res).
append([ ], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

The standard accumulator reverse/2 is also in the class of standard recursive form programs (variation #1 in Appendix B):

```
reverse(X, Y) :- reverse(X, [ ], Y).
reverse([ ], L, L).
reverse([H|T], Y, Z) :- reverse(T, [H|Y], Z).
```

The standard inverse "naive" reverse/2 is another member of the class of standard recursive form programs (variation #9 in Appendix B with a definition for append/3 added):

---

<sup>19</sup>Some of the students programs do contain multiple base case clauses. Having multiple base case clauses is inconsistent with removing a single element on each pass since such programs produce duplicate answers (which is considered incorrect by ADAPT). Thus, restricting the class of programs to those with a single base case clause does not eliminate any correct implementations.

```

reverse([], []).
reverse(L, [H|RT]) :- append(T, [H], L), reverse(T, RT).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

```

The class of standard recursive form programs also includes the following "railway-shunt" reverse/2 which is an accumulator implementation where the accumulator is embedded within the second argument (variation #8 in Appendix B):

```

reverse(A, B) :- rev(A, [B]).
rev([], [B|B]).
rev([Head|Tail], [B|Temp]) :- rev(Tail, [B, Head|Temp]).

```

Another common accumulator version uses difference lists. The difference list A-B represents a list that consists of the list A with the "suffix" list B removed. For example, [a,b,c,d,e]-[c,d,e] represents the list [a,b]. The following is standard recursive form difference list implementation (variation #113 in Appendix B):

```

reverse(L, R) :- reversex(L, R-[]).
reversex([], X-X).
reversex([H|T], X-Y) :- reversex(T, X-[H|Y]).

```

It is also possible to produce an accumulator reverse/2 implementation using any arbitrary function (e.g., f) to combine the accumulator and the result (variation #114 in Appendix B):

```

reverse(L, R) :- reversex(L, f(R, [])).
reversex([], f(X, X)).
reversex([H|T], f(X, Y)) :- reversex(T, f(X, [H|Y])).

```

The class of standard recursive form reverse/2 programs also includes the following railway-shunt version (variation #115 in Appendix B):

```

reverse(L, R) :- rs_reverse(L, [R]).
rs_reverse([], [X|X]).
rs_reverse([H|T], S) :- rs_reverse(T, Y), insert(S, H, Y).
insert([A|B], H, Y) :- enqueue([A], H, X), append(X, B, Y).
enqueue([], E, [E]).
enqueue([H|T], E, [H|R]) :- enqueue(T, E, R).
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).

```

One of the advantages of ADAPT over APROPOS2 is its ability to recognize programs whose arguments have differing forms and permutations. The various versions of the accumulator reverse/2 are one example. Another example is the use of slightly different functions to decompose the input and compose the output as can be seen in the following "naive" versions of reverse/2 (variation #86 in Appendix B):

```
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
reverse([],[]).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
append([],X,X).
```

and the following reverse/2 which adds an additional level of indirection to its output composition program (variation #87 in Appendix B):

```
reverse([H|T],L):-reverse(T,M),enqueue(H,M,L).
reverse([],[]).
enqueue(C,D,E):-append(D,[C],E).
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
```

Permitting the student to use subgoals with different argument forms and permutations enables the following inverse "naive" reverse/2 program which removes an element from the back of the input list using the proedicate last/3 (variation #102 in Appendix B):

```
reverse([],[]).
reverse(L,[H|T]):-last(L,H,M),reverse(M,T).
last([X],X,[]).
last([H|T],X,[H|L]):-last(T,X,L).
```

Additional examples of both correct and incorrect reverse/2 are given in Appendix B. In addition to standard recursive form programs, ADAPT also supports some reverse/2 programs which remove more than one element on each pass. For example, ADAPT can recognize the following inverse naive reverse/2 program:

```
reverse([],[]).
reverse([X],[X]).
reverse(L,[H,G|RT]):-append(T,[G,H],L),reverse(T,RT).
append([],L,L).
append([H|L1],L2,[H|L3]):-append(L1,L2,L3).
```

which removes two elements on each pass. ADAPT is also capable of distinguishing between programs which contain a redundant base case clause (i.e., the base case clause is correct, but results in additional answers) from those which have additional incorrect base case clauses. For example, the second clause of the following program is redundant<sup>20</sup> (variation #38 in Appendix B):

```
reverse([],[]).
reverse([N],[N]).
reverse([A|B],[C|A]):-reverse(B,C).
```

---

<sup>20</sup>There are other errors in these programs. See Appendix B for ADAPT's analysis of these programs.

while the second clause in the following program is incorrect<sup>14</sup> (variation #50 in Appendix B):

```
reverse([], []).
reverse([], Y).
reverse([H|T], [X]) :- reverse(T, [X|H]).
```

ADAPT even recognizes reverse/2 programs that remove elements from both the front and the back of the list on each pass:

```
reverse([], []).
reverse([X], [X]).
reverse([H|L], [G|M]) :- append(T, [G], L), reverse(T, X), append(X, [H], M).
append([], L, L).
append([H|T], X, [H|R]) :- append(T, X, R).
```

The Characterization Theorem is stated for reverse/2 programs. It is believed that similar characterizations can be produced for other programs in the class, including level/2, insertion\_sort/2, length/2, sum/2, and product/2. The focus of our research has been on the development of the ADAPT debugger. However, it is merely a component of our schema-based Prolog tutor. The tutoring component is the described in the next chapter.

---

## Chapter 5

# Teaching Prolog Programming

One of the original goals of intelligent tutoring systems (ITSs) was to extend the power and accuracy of the adaptive instruction available in traditional computer-aided instruction (CAI) systems by examining more than just the student's answers to the problems she was assigned (Sleeman & Brown, 1982). By equipping them with a *student model*, ITSs are able to dynamically adapt their instruction through instructional planning. Instructional planning has the goal of configuring the most efficient sequence of instructional operations to communicate a body of knowledge to the student. There are two major aspects of instructional planning: presentation determination and subject matter selection. Determining how the subject material should be presented relies on accurate modeling of the student's preferences and learning styles. Subject matter selection requires knowledge of the student's abilities (i.e., prerequisite or background knowledge) and her capabilities (i.e., her readiness to learn the new material).

Although the incorporation of a robust student model into an ITS has enabled enhanced instruction over traditional CAI, existing ITSs still fall short of their goal of truly adaptive instruction because they plan instruction based on their students' ability to perform with known concepts rather than their students' readiness to learn new concepts. Based on Vygotsky's developmental theory and his concept of the "zone of proximal development" (Vygotsky, 1978), the schema-based instructional technique employed by our Prolog tutor provides an ideal framework for implementing the guided learning environment necessary to measure the student's knowledge zone.

Traditionally, mental development level has been determined by an individual's performance on a test requiring some sort of individual problem solving. Lev Vygotsky, an influential Russian developmental psychologist in the 1930s, was bothered by the use of this form of testing to determine mental development level since it "oriented learning towards yesterday's development, toward developmental stages already completed" (Vygotsky, 1978, p. 89). As an alternative, he proposed an extension to the traditional testing paradigm which included both independent problem solving and guided (or assisted) problem solving. While independent problem solving provides a good indicator of actual developmental level, he proposed the use of guided problem solving as an indicator of potential developmental level.

Guided problem solving provided a mechanism for measuring what he labelled the "zone of proximal development" which is "the difference between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers" (Vygotsky, 1978, p. 86). The "zone of proximal development" provides a means for distinguishing fully developed (or mature) concepts from developing (or immature) concepts. Empirical evidence (Brown & Ferrara, 1985; Campione, Brown, Ferrara, & Bryant, 1984) has shown the effectiveness of guided problem solving as an instructional technique.

An extension to Vygotsky's basic framework is the method of collaborative problem solving (or peer collaboration) in which two or more individuals of essentially equal abilities work together to solve problems (Doise, Mugny, & Perret-Clermont, 1975,1976; Forman & Cazden, 1985; Perlmutter, Berhrend, Kuo, & Muller, 1989). It has also been argued that collaborative problem solving promotes mental development by creating an environment which produces critical cognitive conflicts (Damon, 1984). Piaget (1970) claimed that cognitive conflicts arise when a child's beliefs are in contradiction with actual experiences. When a child finds herself in disagreement with a collaborating peer, she is forced to examine her point of view and reassess its validity. Thus, collaborative problem solving provides an effective mechanism for motivating children to reassess the validity of their views which in turn forces them to reformulate their beliefs.

Recently, this instructional technique has been incorporated into an intelligent tutoring system (Chan & Baskin, 1988). Their Learning Companion System (LCS) incorporates the use of an automated learning companion to assist the student in solving indefinite integration problems. LCS can be used in a number of modes. In the first mode while the student is becoming familiar with the basic rules of integration, the learning companion and the student work independently and competitively on the same problem. When more complex rules are being introduced (e.g., the substitution method and integration by parts), the student is permitted to watch and offer advise while the learning companion works on the problem. Finally, while practicing miscellaneous exercises the student and learning companion work together collaboratively to solve the problems. The use of collaborative problem solving within the ITS framework has also been proposed by Self and his colleagues (Dillenbourg & Self, in press; Gilmore & Self, 1988). They propose the use of a collaborative partner which is constructed via machine learning techniques as the replacement of an explicit student model.

We have proposed the knowledge zone as an essential dimension on which to distinguish ITSs that model the student's potential capabilities from ITSs that merely model the student's actual abilities (Gegg-Harrison, 1990). Schema-based instruction analyzes the student's performance given various levels of assistance to obtain a rough approximation of its students' knowledge zone and uses it to better tailor its instruction to the capability level of its students (Gegg-Harrison, 1992).

## 5.1 Basic Prolog Schemata

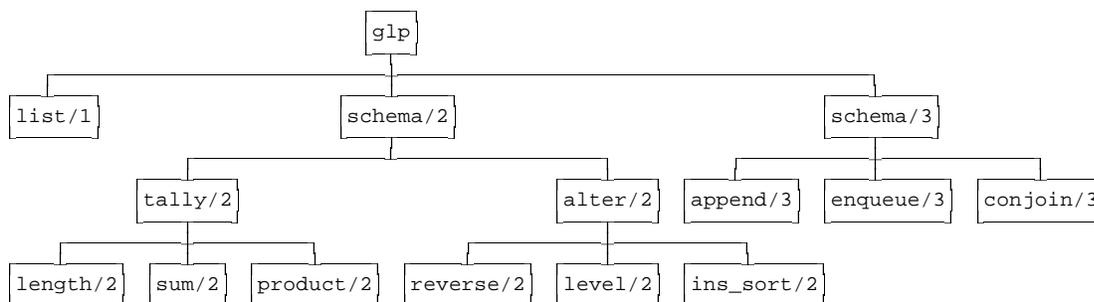
Prolog is based on predicate logic rather than a specific computer architecture, it requires the mastery of only a few constructs to produce fairly complex and interesting programs, and its syntax can be made to appear very much like English (or any other natural language). Therefore, intuitively it seems that Prolog should be fairly easy to learn. However, studies have shown that this is not the case (van Someren, 1985; Taylor & du Boulay, 1987).

Conventional programming languages provide a set of structured programming constructs. However, there are no equivalent constructs in Prolog. The underlying knowledge structure of our schema-based Prolog tutor consists of a set of basic Prolog schemata (Gegg-Harrison, 1989) which are the basic constructs of a structured Prolog for simple recursive list processing. These basic Prolog schemata are an attempt to provide a set of structured programming constructs or primitives for Prolog programmers. For example, programs which process all elements in a given input list can be captured by the global list processing (glp) schema given on page :spotref refid=glpsch. in

## Chapter 1.

Schemata also exist for partial list processing programs which recursively process all elements of a list until some relation holds (e.g., the first occurrence of a specified element or reaching a specified position in the list). More complex programs can be constructed either by having multiple termination criteria (e.g., the nth occurrence of a specified element) or by basing the search on variable positions from the back of the list (e.g., the last occurrence of a specified element).

Using second-order Prolog schemata which permit optional subgoals and arguments, we have found that a large number of simple recursive Prolog programs can be organized hierarchically based on their syntax (Gegg-Harrison, 1989). These hierarchies provide a wealth of Prolog programming examples and techniques which can be used by the tutoring system to tailor its presentation to its students. One possible schema hierarchy is given in Figure 3.



**Figure 3.** Global List Processing Schema Hierarchy

The global list processing (glp) schema (given on page 2 in Chapter 1) is at the root of this hierarchy. It can be instantiated to produce the list/1 program:

```
list([]).
list([H|T]) :- list(T).
```

which validates that its first argument is a legal Prolog list. Alternatively, the global list processing schema can be instantiated to produce a two argument global list processing schema:

```
schema2([], θ1).
schema2([H|T], θ2) :- «θ1(θ3«, θ4»j), »schema2(T, θ5)«, θ2(θ6«, θ7»k)».
```

and a three argument global list processing schema:

```
schema3([], θ1, θ8).
schema3([H|T], θ2, θ9) :- «θ1(θ3«, θ4»j), »schema3(T, θ5, θ10)«, θ2(θ6«, θ7»k)».
```

We can further instantiate the two argument global list processing schema into tally/2:

```
tally([], θ1).
tally([H|T], X) :- «θ1(θ3«, θ4»j), »tally(T, Y), X is f(H, Y)».
```

where  $f(H,Y)$  represents an arbitrary arithmetic expression. The two argument global list processing schema can also be instantiated into `alter/2`:

```
alter([],  $\vartheta_1$  ).
alter([H|T], X ) :-  $\ll \theta_1 ( \vartheta_3 \ll , \vartheta_4 \gg^j ) , \gg$  alter(T,Y) , p(H,Y,X) .
```

where  $p(H,Y,X)$  is an arbitrary predicate which has three list arguments. We can further instantiate `tally/2`, `alter/2`, and the three argument global list processing schema into programs for `length/2`, `sum/2`, `product/2`, `reverse/2`, `level/2`, `sort/2` (`insertion_sort/2`), `append/3`, `enqueue/2`, and `conjoin/2`.

## 5.2 Schema-Based Instruction

The underlying philosophy of our Prolog tutor is to teach programming through program schemata. Rather than teaching recursion as a general technique by presenting the mathematical foundations of recursive function theory whereby the student is given a universal recursive schemata (e.g., (Yokomori, 1986)) or using an example-based method whereby the student is forced to generalize a general understanding of recursion from specific examples, schema-based instruction teaches recursion through a set of general programming techniques. Thus, the student is relieved of the task of inducing the basic concept of recursion while being provided with a "toolbox" of general programming techniques.

Because of the hierarchical nature of our domain, it is possible to measure a student's capability with respect to her understanding of the schemata (or programming techniques). When a program is assigned, a schema is selected to serve as a template for the student to complete. The template serves as a Prolog microworld. These Prolog microworlds limit the number of ways of implementing a program. If the student is unable to complete the template to produce the desired program (or if she incorrectly completes the template) then hints are given. Hints include explaining the components of the templates (i.e., describing the microworld) and providing the student with partial or complete solutions to the assigned programs (i.e., simplifying the microworld).

The student's performance on the assigned program determines her knowledge zone with respect to the assigned schema. If the student is able to solve the problem (i.e., produce the desired program) without any hints then the schema (i.e., programming technique) is within her ability. If she can solve the problem after being given some of the above hints then the programming technique is within her capability, but is not a mature concept within her ability yet. Finally, if the student is still unable to solve the problem after she has been given hints then the programming technique is considered outside her "zone of proximal development." Once the knowledge zone has been identified, the tutor can use it to adapt the lesson plan to the student's cognitive potential. The lesson plan can be modified in a number of ways, including the selection of the next problem and template to be assigned.

Lesson	Problem	Template	Example
1	tr_sum/2	tr_tally/2	tr_length/2
2	tr_product/2	tr_template/2	-
3	tr_odd_sum/2	gtr_tally/2	tr_count/2
4	tr_even_sum/2	gtr_template/2	-
5	tr_pos_product/2	gtr_template/2	-
6	sum/2	template/2	length/2
7	count/2	g_template/2	tr_count/2
8	append/3	template/3	enqueue/3
9	reverse/2	-	-

**Figure 4.** Initial Lesson Plan

The basic lesson plan is the same for each student as shown in Figure 4. Each of the programs of the lessons have corresponding entries in the schema hierarchy of Figure 3. The following notation has been used:

- The prefix `tr_` has been used to represent the tail recursive version of the program. Note that the absence of the `tr_` prefix implies the more general (declarative or non-accumulator) version of the program.
- The prefix `g_` has been used to represent a guarded version of the schema. For programs, the actual guard (e.g., `odd`) is part of the predicate name.<sup>15</sup> A guarded version of a program is one which contains multiple recursive clauses. For example, `tr_odd_sum/2` (which is the tail recursive version of the `sum/2` program using the `odd` guard) has the following normal form implementation:

```
tr_odd_sum(L,S) :- tr_odd_sum(L,0,S).
tr_odd_sum([],S,S).
tr_odd_sum([H|T],X,S) :- odd(H),!,Y is X+H,tr_odd_sum(T,Y,S).
tr_odd_sum([H|T],X,S) :- tr_odd_sum(T,X,S).
```

Each lesson consists of three components: problem, template, and example. The general form of each lesson is as follows. If a new concept is being introduced then the tutor gives the student a brief overview of the new concept in order to provide an appropriate context. If the lesson includes an example then the example is shown and explained. The problem is then described and the student

---

<sup>15</sup>The one exception to this convention is the program `count/2` (`tr_count/2`) which is actually a guarded version of `length/2` (`tr_length/2`).

is asked to complete the template in order to produce a correct program. A tutoring session is initiated by the student entering tutor. The following scenario<sup>16</sup> gives the interaction between a student who has mastered tail recursion and is now ready to be introduced to the more general forms of recursion with lesson 6:

Student:       tutor

Tutor:        Welcome to the Schema-Based Prolog Tutor!  
              What is your name?

Student:       Barb

Tutor:        OK, Barb, let's look at more general forms of recursion today. We have already looked at tail-recursive versions of `tr_length/3`, `tr_sum/3`, and `tr_product/3`. Now let's look at alternative, more declarative approaches to solving these problems. For example, the tail-recursive program for `tr_length/3` looks like:

```
length(List,Length) :- tr_length(List,0,Length).
tr_length([],Acc,Acc).
tr_length([H|T],N,L) :- M is N+1,tr_length(T,M,L).
```

and the corresponding declarative version of `length/2` looks like:

The "length" of the empty list is 0 (Base Case):

```
length([],0).
```

The "length" of a non-empty list is equal to 1 more than the "length" of its tail (Recursive Step):

```
length([H|T],Length) :- length(T,L),Length is L+1.
```

Your task is to write a Prolog program which takes as input a list and returns the sum of the list of elements using this declarative approach. You've done this before with the tail-recursive `tr_sum/3`:

```
sum(List,Sum) :- tr_sum(List,0,Sum).
tr_sum([],Acc,Acc).
tr_sum([H|T],N,S) :- M is N+H,tr_sum(T,M,S).
```

---

<sup>16</sup>This scenario was actually produced by our Prolog tutor. It is important to note, however, that this portion of the tutor is not fully developed. Although it is capable of producing this scenario, it may fail to produce similar scenarios. It is included here to show the potential power of our schema-based Prolog tutor.

Now try writing the declarative version of `sum/2` by completing the following template:

```
sum([],*).
sum([H|T],Sum) :- sum(T,S),*.
```

At this point, the student has two options. She can either attempt to write the program or she can request assistance. If she attempts to write the program and produces a correct program then the tutor assigns her the next problem in the original lesson plan which is `count/2`. Because she was able to solve the problem without any assistance from the tutor, it is assumed that the student has a good grasp of the programming construct being taught (i.e., the construct is well within her "zone of proximal development") so the lesson is tailored to the student by omitting the remedial problems of `product/2` and `square_sum/2` and generalizing the template (i.e., instead of assigning template `g_tally/2` in lesson 7 the tutor assigns the more general template `g_schema/2`):

Student: `sum([],0).`  
`sum([H|T],Sum) :- sum(T,S),Sum is S+H.`

Tutor: Very good, Barb! OK, now let's try a similar problem. Let's assume that our list now contains arbitrary elements (i.e., some of the elements are integers and some of them are non-integers) and we want to "count" the number of integers in the list. You've already given a tail-recursive solution for this problem:

```
count(List,Count) :- tr_count(List,0,Count).
tr_count([],Acc,Acc).
tr_count([H|T],N,C) :- integer(H),!,M is N+1,
                        tr_count(T,N,C).
tr_count([H|T],N,C) :- tr_count(T,N,C).
```

Now try writing this program again using the declarative approach you've just learned by completing the following template:

```
count([],*).
count([H|T],*) :- *,!,*,count(T,*),*.
count([H|T],*) :- *,count(T,*),*.
```

If the student is unable to solve the problem then she can request assistance from the tutor by typing `help`. The first level of assistance that the tutor provides is to explain the components of the template. There are two ways to explain the components of the template. The first approach is to simply break the template down into its recursive function components, base case and recursive step, and explain the role of each component:

Student: `help`

Tutor: There are 2 major components to this template:

- . base case
- . recursive step

For the base case, the basic idea is to stop processing when the list becomes empty and return 0 for the sum.

For the recursive step, the basic idea is to remove the first element from the input list and recursively invoke the main predicate with the tail of the list and then add the value of the head to the sum of the tail of the list.

Which component do you want to attempt first:

1. base case
2. recursive step
3. need further assistance?

After explaining each of these components, the tutor gives the student a chance to complete them one at a time or the student can request further assistance. If a student requests further assistance then the tutor provides her with a more detailed template "explanation" by providing her with a more specific template to complete:

Student: help

Tutor: Try writing the program by completing the following template:

```
sum([],*).
sum([H|T],Sum) :- sum(T,S),Sum is *.
```

By providing a more specific template for the student to complete, the tutor has given the student a partial solution to the problem. This type of "hint" may be sufficient for some students to complete the program. For such students, the problem is within their "zone of proximal development" but is not a mature concept yet. It is now the goal of the tutor to ensure that this concept is more fully developed. This can be done by assigning a remedial problem (e.g., product/2) which belongs to the same class of programs as the problem that she just completed. At this point, the tutor is fairly confident that the student could solve the new problem given the more-specific template, but must ensure that the student can solve this problem given the original template:

Student: 

```
sum([],0).
sum([H|T],Sum) :- sum(T,S),Sum is S+H.
```

Tutor: Very good, Barb!  
OK, now let's try a similar problem. Write a program

which takes as input a list and returns the product of the list of elements by completing the following template:

```
product([],*).
product([H|T],Product) :- product(T,P),*.
```

Upon successful completion of this remedial lesson, the tutor returns to the 7th lesson of the student's lesson plan and assigns count/2 with template g\_tally/2:

Student: `product([],1).`  
`product([H|T],Product) :- product(T,P),Product is P*H.`

Tutor: Very good, Barb!

OK, now let's try a similar problem. Let's assume that our list now contains arbitrary elements (i.e., some of the elements are integers and some of them are non-integers) and we want to "count" the number of integers in the list. You've already given a tail-recursive solution for this problem:

```
count(List,Count) :- tr_count(List,0,Count).
tr_count([],Acc,Acc).
tr_count([H|T],N,C) :- integer(H),!,M is N+1,
                       tr_count(T,N,C).
tr_count([H|T],N,C) :- tr_count(T,N,C).
```

Now try writing this program again using the declarative approach you've just learned by completing the following template:

```
count([],*).
count([H|T],*) :- integer(H),!,count(T,*),*.
count([H|T],*) :- count(T,*),*.
```

If the student is still unable to solve the problem after being given a partial solution (i.e., a more-specific template) then the complete solution is given and explained. In this case, the problem is considered outside the student's "zone of proximal development" so the tutor must alter its original instructional goal and focus on a smaller class of problems. Note that in this case, it is not clear whether or not the student can handle this smaller class of problems. This can be tested by assigning a remedial problem (e.g., product/2 and asking the student to complete the more-specific template. Even if she is able to handle this problem then another remedial problem (e.g., square\_sum/2 which is the same as sum/2 except the addition operator is replaced with a "new" operator #) should be assigned with the original template to ensure that the original class of problems is within the student's capabilities.

Student: help

Tutor: Since you seem to be having so much trouble, let me give you my solution to the problem:

The "sum" of the empty list is 0 (Base Case):

```
sum([],0).
```

The "sum" of a non-empty list is equal to the "sum" of its tail plus the head of the list (Recursive Step):

```
sum([H|T],Sum) :- sum(T,S),Sum is S+H.
```

OK, Barb, let's continue with another problem. Write a program which takes as input a list and returns the product of the list of elements by completing the following template:

```
product([],*).
product([H|T],Product) :- product(T,P),Product is *.
```

Student: 

```
product([],1).
product([H|T],Product) :- product(T,P),Product is P*H.
```

Tutor: Very good, Barb!

OK, now let's try a similar problem. Write a program which takes as input a list and returns the "square\_sum" (i.e., the sum of the squares of the elements) of the list of elements (where "X is Y # Z" binds X to the expression  $Y + Z^2$ ) by completing the following template:

```
square_sum([],*).
square_sum([H|T],SSum) :- square_sum(T,S),*.
```

If the student is able to complete this second remedial problem given the original template then the tutor returns to the 7th lesson of the student's lesson plan and assigns count/2 with template `g_tally/2`:

Student: 

```
square_sum([],0).
square_sum([H|T],SSum) :- square_sum(T,S),SSum is S # H.
```

Tutor: Very good, Barb!

OK, now let's try a similar problem. Let's assume that our list now contains arbitrary elements (i.e., some of the elements are integers and some of them are non-integers) and we want to "count" the number of integers in the list. You've already given a tail-recursive solution for this problem:

```

count(List,Count) :- tr_count(List,0,Count).
tr_count([],Acc,Acc).
tr_count([H|T],N,C) :- integer(H),!,M is N+1,
                        tr_count(T,N,C).
tr_count([H|T],N,C) :- tr_count(T,N,C).

```

Now try writing this program again using the declarative approach you've just learned by completing the following template:

```

count([],*).
count([H|T],*) :- integer(H),!,count(T,*),*.
count([H|T],*) :- count(T,*),*.

```

We have focussed our discussion on the teaching of computer programming, specifically teaching novice Prolog programmers recursive list processing. We have presented schema-based instruction as an alternative to the approaches advocated in most introductory and intermediate Prolog texts. By explicitly presenting Prolog schemata and presenting programs in the same class as a coherent unit, schema-based instruction stresses the importance of classes of programs and promotes the acquisition of basic Prolog programming constructs. Use of schema-based instruction need not be limited to programming. Schemata are fundamental to most human cognitive processes.

One way that human beings overcome the limitations of their memory is by chunking (i.e., organizing individual items that have been learned as a single group). Miller (1956) has proposed that humans have a memory span of  $7 \pm 2$  chunks. This finding has been validated by studying experts and novices in a number of domains, including chess, go, circuit design, physics, and computer programming. These studies have shown that the key difference between experts and novices is not the size of their memory span, but rather their ability to chunk information together into meaningful units.

The seminal study that showed that experts are better than novices at chunking meaningful information was conducted by Chase and Simon (1973) using chess players of differing skill levels. They had 3 levels of chess players: masters, class A players, and beginners. Two sets of chess board configurations were used:

- Set 1 - actual middle game configurations
- Set 2 - random board configurations

Each board configuration was displayed to each of the chess players for 5 seconds and then removed. Then each player was asked to reconstruct the board configuration from memory. The results of the experiment showed that if the board configuration represented an actual middle game configuration then the chess masters correctly recalled more pieces than the class A players who in turn correctly recalled more pieces than the beginners. For random board configurations, however, the superiority of the chess masters completely disappeared. In fact, the chess masters actually recalled fewer pieces than the class A players and the beginners. In addition to measuring the accuracy of the board configuration that the chess players produced, Chase and Simon also measured the order in which they produced the board configuration. This measurement enabled them to identify the size and number of chunks that each group of players used. They found that

chess masters recalled larger chunks than the other players. Similar findings also exist for the oriental board game of go (Reitman, 1976) and circuit design (Egan & Schwartz, 1979).

Schemata provide a method of organizing meaningful information about complex domains. In physics problem solving, for example, Chi, Fe<ovich, & Glaser (1981) found that experts have more and better problem schemata than novices. They also found that novices tend to categorize problems based on surface features of the problem statement, while experts categorized problems with respect to applicable physical laws. Similar findings exist for computer programmers (Adelson, 1981; van Someren, 1990). Thus, a schema-based approach to teaching computer programming seems to promote the transition from novice to expert programmer.

We have also argued that our schema-based instructional approach provides an ideal framework for implementing the guided learning environment necessary to measure the student's knowledge zone. The applicability of this approach, however, is not limited to teaching recursive Prolog programming. It can also be used for Lisp programming by extending Soloway's work on Lisp templates (Soloway, 1985). More generally, this approach is amenable to any hierarchical domain. Taking as our generalization operator that a given microworld is more complex than another one, any domain for which increasingly complex microworlds (Burton, Brown, & Fischer, 1984) is applicable is a suitable domain for our approach.

## Chapter 6

### Summary

#### 6.1 Major Contributions

The ADAPT debugger has advanced the "state of the art" in automated program debugging in 2 major ways:

1. ADAPT accepts a larger class of programs than APROPOS2 and ADAPT uses fewer normal form programs than APROPOS2.
2. The class of *reverse/2* programs that ADAPT recognizes has been characterized. This is something that has never been done for any automated program debugger. In fact, no one has ever attempted to characterize the class of recognizable responses for any tutoring system, regardless of the domain.

In addition to advancements in automated program debugging, the schema-based Prolog tutor advances the "state of the art" in programming language tutoring by exploiting the use of program schemata to enable instruction that can be tailored to the individual needs of its students. Furthermore, minor contributions have been made to the field of logic programming:

1. The Generalization Algorithm given in Appendix A is a more efficient algorithm for computing MSGs than existing algorithms (Plotkin, 1970; Reynolds, 1970).
2. ADAPT is the first program debugger to use unfolding/folding to permute and restructure arguments within subgoals.
3. The Characterization Theorem shows that all standard recursive form *reverse/2* programs can be captured by a single normal form program.

#### 6.2 Future Research

ADAPT is actually capable of recognizing some non-standard recursive form *reverse/2* programs. It can recognize correct *reverse/2* programs which contain more than one terminating clause. This can be achieved by unfolding the recursive call subgoal in the recursive clause of the normal form program until it has the same number of terminating clauses as the target program. This is possible since the *reverse/2* normal form program originally has a single terminating clause. Because the original normal form program has a single terminating clause, each time the recursive call subgoal in the recursive clause is unfolded it produces another terminating clause assuming the terminating

clause's head unifies with the recursive call subgoal which is the case for "naive" reverse/2. Unfolding the recursive call subgoal in the recursive clause of the normal form program produces a program which has two terminating clauses:

```
reverse([], []).
reverse([X], [X]).
reverse([H1, H2 | T], R) :- reverse(T, M),
                             append(M, [H2], M1), append(M1, [H1], R).

append([], L, L).
append([H | T], X, [H | L]) :- append(T, X, L).
```

Unfolding the recursive call subgoal of the recursive clause of this newly generated program with respect to the original normal form program produces:

```
reverse([], []).
reverse([X], [X]).
reverse([X1 , X2], [X2 , X1]).
reverse([H1 , H2 , H3 | T], R) :- reverse(T, M), append(M, [H3], M2),
                                     append(M2, [H2], M1), append(M1, [H1], R).
append([], L, L).
append([H | T], X, [H | L]) :- append(T, X, L).
```

which has 3 terminating clauses. Repeating this procedure n-3 more times produces:

```
reverse([], []).
reverse([X], [X]).
.
.
.
reverse([X1 , ... , Xn-1], [Xn-1 , ... , X1]).
reverse([H1 , ... , Hn | T], R) :- reverse(T, M), append(M, [Hn], Mn-1),
                                     append(Mn-1, [Hn-1], Mn-2), ... , append(M1, [H1], R).
append([], L, L).
append([H | T], X, [H | L]) :- append(T, X, L).
```

which is a reverse/2 implementation with n terminating clauses. As mentioned in Chapters 2 and 4, ADAPT is also capable of recognizing correct reverse/2 programs that remove multiple elements from the front and/or the back of the input list.

Our methodology works well for identifying whether or not a given reverse/2 program is correct since the Characterization Theorem guarantees that ADAPT can capture all correct standard recursive form programs. The bug detector is fairly effective at actually detecting bugs for programs that have easily identifiable programming techniques. However, there are incorrect programs in which the programming technique cannot be easily identified by outside observation. For example, in variation #44 in Appendix B the program contains a reassignment error in the second and third clauses. It also contains an apparent invalid append in the first subgoal of each of these clauses:

```
reverse([], L).
reverse([H], Ans) :- Ans = [Ans | H].
```

$$\text{reverse}([H|T], \text{Ans}) : -\text{Ans} = [\text{Ans}|H], \text{reverse}(T, \text{Ans}).$$

Clearly, the student did not attempt to implement a back processing version of reverse/2. However, it is not clear whether her program is an incorrect accumulator reverse/2 or an incorrect naive reverse/2. At first glance, it appears to be an incorrect accumulator reverse/2 in which the student merely attempted to accumulate the result in a standard programming language variable. Upon closer inspection, however, one sees that the result is being accumulated in the wrong order leading to another interpretation of the student's intentions in which she has an incorrect virtual machine for Prolog<sup>17</sup> in which she believes that either subgoal execution occurs from right-to-left rather than left-to-right or that the recursive call is always selected first. Such a virtual machine coupled with a standard misconception about Prolog variables would explain the creation of this incorrect naive reverse/2 program. In such cases, an accurate student model is essential in order to determine which technique was actually employed by the student.

User models have been one of the prime focuses of attention in many areas of artificial intelligence and cognitive science research, including general conversational systems (Kobsa & Wahlster, 1989), librarian systems (Rich, 1979), and intelligent tutoring systems (Self, 1974, 1988; VanLehn, 1988). In all of these studies, however, the emphasis in student modeling<sup>18</sup> has been in modeling the student from the system's point of view. For example, tutoring systems model what the system thinks the student has learned rather than what the student thinks the tutor has taught. Although this distinction appears subtle, it is critical in understanding a student's misconceptions when her underlying model of the domain is drastically different from that which the teacher is attempting to present.

In constructing a student model, there are two possible approaches. One approach is to predict a model of the student and then attempt to verify the model or make modifications to the model by evaluating the student's performance on a set of tests that the tutor requires the student to solve. This approach relies on the assumption that one is able to make an accurate prediction on the selection of models. Otherwise, the tests will simply reject the model leaving no room for modification. Another approach to student modeling is to have the student provide a model of herself.

In any learning situation, the student is (implicitly) building a model of what she thinks the instructor wants her to learn. This model may be, and in fact often is, very different from the model that the instructor is attempting to teach. Because the differences may be so great, it is inappropriate to attempt to force the student's model into the instructor's model as is done with overlays (Carr & Goldstein, 1977). One way around this is to force the student to adopt the system's model by tightly monitoring her interaction as is done with Anderson's tutors for Lisp (Reiser & Anderson, 1985), geometry (Anderson, Boyle, & Yost, 1985), and algebra (Lewis, Milan, & Anderson, 1987) or to limit her interaction as is done with Woolf's MENO-TUTOR (Woolf & McDonald, 1984) and

---

<sup>17</sup>The virtual machine for Prolog is the programmer's conceptual model of Prolog (i.e., how she thinks the Prolog interpreter/compiler executes her program).

<sup>18</sup>We use the term student model as opposed to the more general term user model since the major focus of this dissertation is in the area of intelligent tutoring systems. Note, however, that the discussion in this section applies to user models in general.

Bonar's BRIDGE (Bonar & Cunningham, 1988).

The most common way of evaluating a student is via testing. This is the method of choice for public schools from kindergarten through graduate school. Note, however, that graduate schools generally promote the writing of research papers and giving presentations which closely parallels self-evaluation. In existing computer tutors, evaluation of the student is done by requiring the student to answer a set of questions or solve a problem. For example, for tutors that aid novice programmers, the system requires the student to write a number of computer programs and then debugs them in order to determine what the student knows and what misconceptions she might be harboring.

Incorporation of the student's view into the student model can be obtained in tutorial systems by forcing the student to engage in self-evaluation. There are at least two forms of self-evaluation. One form of self-evaluation requires the student to tell the tutor about herself (i.e., what the student thinks is important). This approach has been implemented in the GRUNDY librarian system (Rich, 1979). GRUNDY works with a set of models called stereotypes which are collections of attributes that are obtained by having the student provide a description of herself from a pre-determined set of descriptive keywords. This corresponds to asking the student what knowledge she knows and having her enumerate everything she knows. Although this approach has some applications as evidenced by the success of GRUNDY, such a method is not practical for intelligent tutoring systems. Thus, we must look for an alternate method of self-evaluation.

One possibility is to incorporate self-evaluation into the standard testing methodology. This can be achieved by asking the student to write a program and then explain how she thinks it works. Taylor (1988) has identified several misconceptions novice Prolog programmers have about the Prolog virtual machine. The ideal interface for identifying the student's virtual machine would be a sophisticated Prolog tracer which would enable the student to describe exactly what she thinks the program does even though this may be drastically different from the way the Prolog interpreter would execute the program. Such a tracer would be an invaluable aid in the analysis of programs produced by students with incorrect virtual machines for Prolog (e.g., variations #44-#55 in Appendix B).

A skill that is often overlooked in the computer science curriculum is the art of debugging software. No courses are taught that specifically address this skill, yet most computer scientist students become fairly respectable debuggers by the time they graduate. For the most part, they learn this skill through practice. They are asked to write programs in many of their classes. Because their first attempts at writing a given program often result in incorrect programs, they are forced to debug these programs. As a result of repeating this process again and again, they develop a methodology for debugging. Automated programming tutors make it possible to explicitly teach the art of debugging.

One possibility is to incorporate a SOPHIE-like "gaming environment" (Brown, Burton, & de Kleer, 1982) into our schema-based Prolog tutor. The idea is to have the "inserter" (which could be the tutor itself) introduce an arbitrarily chosen bug into an otherwise correct program. Then the "debugger" (which would be the student in a traditional setting or the other student in a competitive learning environment) must find the bug in the program. ADAPT can enhance the "gaming environment" by inserting the bug into the normal form program and then applying transformations to this buggy normal form program producing a large array of unique programs which all have the same bug in them. Assuming the existence of a debugging methodology, this enhancement would

enable our schema-based Prolog tutor to explicitly teach its students how to approach the task of debugging.

## 6.3 Conclusions

Automated tutors for novice programmers have been shown to be useful in the classroom. Research efforts in this area have primarily focussed on the automated program debugging portion of the tutoring system. This dissertation described an automated tutor for novice Prolog programmers which embodies a methodology for teaching recursive programming through schemata while at the same time providing a robust automated program debugger.

Debugging recursive programs is a difficult task even for experienced programmers. Thus, the automation of this process is a formidable objective. We have presented an approach to automated program debugging that exploits the use of program schemata in both its algorithm recognition and bug detection phases. We have argued that the evaluation of program debuggers consists of two important phases. In the first phase, the class of acceptable programs must be identified. In the second phase, the debugger must be shown to be sound and complete with respect to this class of programs. Although necessary for the first phase, empirical studies are not sufficient for the second phase.

In addition to presenting a program debugger which is capable of recognizing a larger class of programs with fewer normal form programs than existing program debuggers, we also provided a characterization of the class of programs that the debugger can recognize. Being able to provide a characterization of the domain of an automated tutoring system is essential to the acceptance of computer-based tutors, yet the ADAPT debugger is the only system to provide such a characterization for a complex domain. Like the PROUST and TALUS research projects (in their times), the ADAPT research project described in this dissertation is what we believe to be the current "state of the art" in automated program debugging.

# Bibliography

- Adam, A. & Laurent, J. (1980). LAURA, A System to Debug Student Programs. *Artificial Intelligence*, 15, 75-122.
- Adelson, B. (1981). Problem Solving and the Development of Abstract Categories in Programming Languages. *Memory & Cognition*, 9, 422-433.
- Anderson, J.R., Boyle, C.F., & Yost, G. (1985). The Geometry Tutor. *Proceedings of the 9<sup>th</sup> International Joint Conference on Artificial Intelligence*, Los Angeles, California, 1-7.
- Anderson, J.R., Pirolli, P., & Farrell, R. (1988). Learning to Program Recursive Functions. In M. Chi, R. Glaser, & M. Farr (Eds.), *The Nature of Expertise*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Anderson, J.R. & Reiser, B.J. (1985). The Lisp Tutor. *Byte*, 10, 159-175.
- Apt, K.R. & van Emden, M.H. (1982). Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29, 841-862.
- Barr, A., Beard, M., & Atkinson, R.C. (1976). The Computer as a Tutorial Laboratory: The Stanford BIP Project. *International Journal of Man-Machine Studies*, 8, 467-496.
- Bol, R.N. (1991). *Loop Checking in Logic Programming*. Ph.D. Dissertation, Department of Software Technology, University of Amsterdam, Amsterdam, The Netherlands.
- Bonar, J. & Cunningham, R. (1988). Bridge: An Intelligent Tutor for Thinking about Programming. In J. Self (Ed.), *Artificial Intelligence and Human Learning: Intelligent Computer-Assisted Instruction*, New York: Chapman and Hall.
- Boyer, R.S. & Moore, J.S. (1979). *A Computational Logic*. New York: Academic Press.
- Brna, P., Bundy, A., Dodd, T., Eisenstadt, M., Looi, C.K., Pain, H., Smith, B., & van Someren, M. (1991). Prolog Programming Techniques. *Instructional Science*, 20, 111-133.
- Brown, A.L. & Ferrara, R.A. (1985). Diagnosing Zones of Proximal Development. In J.V. Wertsch (Ed.), *Culture, Communication, and Cognition: Vygotskian Perspectives*. New York: Cambridge University Press.
- Brown, J.S., Burton, R.R., & de Kleer, J. (1982). Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPE I, II, and III. In D. Sleeman & J.S. Brown (Eds.), *Intelligent Tutoring Systems*. New York: Academic Press.

- Burstable, R.M. & Darlington, J. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24, 44-67.
- Burton, R.R., Brown, J.S., & Fischer, G. (1984). Skiing as a Model of Instruction. In B. Rogoff & J. Lave (Eds.), *Everyday Cognition: Its Development in Social Context*. Cambridge, MA: Harvard University Press.
- Campione, J.C., Brown, A.L., Ferrara, R.A., & Bryant, N.R. (1984). The Zone of Proximal Development: Implications for Individual Differences and Learning. In B. Rogoff and J.V. Wertsch (Eds.), *Children's Learning in the "Zone of Proximal Development"*. San Francisco: Jossey-Bass.
- Carr, B. & Goldstein, I.P. (1977). *Overlays: A Theory of Modeling for Computer-Aided Instruction*. AI Lab Memo 406, Massachusetts Institute of Technology, Cambridge, Massachusetts.
- Chan, T. & Baskin, A.B. (1988). "Studying with the Prince" - The Computer as a Learning Companion. In *Proceedings of the ITS-88 International Conference on Intelligent Tutoring Systems*, Montreal, Quebec, 194-200.
- Clark, K.L. (1978). Negation as Failure. In H. Gallaire & J. Minker (Eds.), *Logic and Data Bases*. New York: Plenum Press.
- Chase, W.G. & Simon, H.A. (1973). Perception in Chess. *Cognitive Psychology*, 4, 55-81.
- Chi, M.T.H., Feltovich, P.J., & Glaser, R. (1981). Categorization and Representation of Physics Problems by Experts and Novices. *Cognitive Science*, 5, 121-152.
- Clark, K.L. & Tärnlund, S (1977). A First-Order Theory of Data and Programs. In *Proceedings of the 1977 IFIP Congress*, Toronto, Ontario, 939-944.
- Clocksink, W.F. & Mellish, C. (1987). *Programming in Prolog (3<sup>rd</sup> edition)*. New York: Springer-Verlag.
- Corbett, A.T. & Anderson, J.R. (in press). The Lisp Intelligent Tutoring System: Research in Skill Acquisition. In J. Larkin, R. Chabay, & C. Scheftic (Eds.), *Computer-Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Damon, W. (1984). Peer Education: The Untapped Potential. *Journal of Applied Developmental Psychology*, 5, 331-343.
- Dillenbourg, P. & Self, J. (in press). Designing Human-Computer Collaborative Learning. In

- C. O'Malley (Ed.), *Computer-Supported Collaborative Learning*. New York: Springer-Verlag.
- Doise, W., Mugny, G., & Perret-Clermont, A. (1975). Social Interaction and the Development of Cognitive Operations. *European Journal of Social Psychology*, 5, 367-383.
- Doise, W., Mugny, G., & Perret-Clermont, A. (1976). Social Interaction and Cognitive Development: Further Evidence. *European Journal of Social Psychology*, 6, 245-247.
- Egan, D.E. & Schwartz, B.J. (1979). Chunking in Recall of Symbolic Drawings. *Memory & Cognition*, 7, 149-158.
- Elsom-Cook, M. (1988). Guided Discovery Tutoring and Bounded User Modelling. In J. Self (Ed.), *Artificial Intelligence and Human Learning: Intelligent Computer-Assisted Instruction*. New York: Chapman and Hall.
- van Emden, M.H. & Kowalski, R. (1976). The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23, 733-742.
- Fischer, G. (1987). A Critic for Lisp. *Proceedings of the 10<sup>th</sup> International Joint Conference on Artificial Intelligence*, Milan, Italy, 177-184.
- Forman, E.A. & Cazden, C.B. (1985). Exploring Vygotskian Perspectives in Education: The Cognitive Value of Peer Interaction. In J.V. Wertsch (Ed.), *Culture, Communication, and Cognition: Vygotskian Perspectives*. New York: Cambridge University Press.
- Gegg-Harrison, T.S. (1989). Basic Prolog Schemata. *Proceedings of the NACLP'89 Workshop on Logic Programming Environments: The Next Generation*, Cleveland, Ohio.
- Gegg-Harrison, T.S. (1990). Zoning in on Student Models: Modeling Cognitive Potential in a Schema-Based Prolog Tutor. *Proceedings of the 2<sup>nd</sup> International Workshop on User Models*, Honolulu, Hawaii.
- Gegg-Harrison, T.S. (1991). Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20, 173-192.
- Gegg-Harrison, T.S. (1992). Adapting Instruction to the Student's Capabilities. *Journal of Artificial Intelligence in Education*, 3, 169-181.
- van Gelder, A. (1986). Negation as Failure Using Tight Derivations for General Logic Programs. *Proceedings of the 3<sup>rd</sup> International Symposium on Logic Programming*, Salt Lake City, Utah, 127-138.
- Gilmore, D. & Self, J. (1988). The Application of Machine Learning to Intelligent Tutoring

- Systems. In J. Self (Ed.), *Artificial Intelligence and Human Learning: Intelligent Computer-Assisted Instruction*. New York: Chapman and Hall.
- Goldstein, I.P. (1975). Summary of MYCROFT: A System for Understanding Simple Picture Programs. *Artificial Intelligence*, 6, 249-288.
- Hansson, A. & Tärnlund, S (1977). Program Transformation By Data Structure Mapping. In K.L. Clark (Ed.), *Logic Programming*. New York: Academic Press.
- Johnson, W.L. (1986). *Intention-Based Diagnosis of Novice Programming Errors*. Los Altos, CA: Morgan Kaufmann.
- Kanamori, T. & Horiuchi, K. (1987). Construction of Logic Programs Based on Generalized Unfold/Fold Rules. *Proceedings of the 4<sup>th</sup> International Conference on Logic Programming*, Melbourne, Australia, 744-768.
- Kanamori, T. & Kawamura, T. (1988). *Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation*. ICOT Technical Report TR-403, Mitsubishi Electric Corporation, Amagasaki, Japan.
- Kobsa, A. & Wahlster, W. (Eds.) (1989). *User Models in Dialog Systems*. New York: Springer-Verlag.
- Koffman, E.B. & Blount, S.E. (1975). Artificial Intelligence and Automatic Programming in CAI. *Artificial Intelligence*, 6, 215-234.
- Lewis, M.W., Milan, R., & Anderson, J.R. (1987). The TEACHER'S APPRENTICE: Designing an Intelligent Authoring System for High School Mathematics. In G. Kearsley (Ed.), *Artificial Intelligence and Instruction: Applications and Methods*. Reading, MA: Addison-Wesley.
- Lloyd, J.W. (1987). *Foundations of Logic Programming (2<sup>nd</sup> edition)*. New York: Springer-Verlag.
- Looi, C. (1988). *Automatic Program Analysis in a Prolog Intelligent Teaching System*. Ph.D. Dissertation, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.
- McCalla, G.I., Bunt, R.B., & Harms, J.J. (1986). The Design of the SCENT Automated Advisor. *Computational Intelligence*, 2, 76-92.
- McCalla, G.I., Greer, J.E., & the SCENT Research Team (1988). Intelligent Advising in Problem Solving Domains: The SCENT-3 Architecture. *Proceedings of the ITS-88 International Conference on Intelligent Tutoring Systems*, Montreal, Quebec, 124-131.

- Miller, G.A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits of our Capacity for Processing Information. *Psychological Review*, 63, 81-97.
- Murray, W.R. (1988). *Automatic Program Debugging for Intelligent Tutoring Systems*. Los Altos, CA: Morgan Kaufmann.
- Naish, L. (1986). *Negation and Control in Prolog*. New York: Springer-Verlag.
- O'Keefe, R.A. (1990). *The Craft of Prolog*. Cambridge, MA: MIT Press.
- Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.
- Perlmutter, M., Berhrend, S.D., Kuo, F., & Muller, A. (1989). Social Influences on Children's Problem Solving. *Developmental Psychology*, 25, 744-754.
- Plotkin, G.D. (1970). A Note on Inductive Generalization. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*. Edinburgh: Edinburgh University Press.
- Plümer, L. (1990). *Termination Proofs for Logic Programs*. New York: Springer-Verlag.
- Reitman, J.S. (1976). Skilled Perception in Go: Deducing Memory Structures from Inter-response Times. *Cognitive Psychology*, 8, 336-356.
- Reiser, B.J., Kimberg, D.Y., Levitt, M.C., & Ranney, M. (in press). Knowledge Representation and Explanation in GIL, An Intelligent Tutor for Programming. In J. Larkin, R. Chabay, & C. Scheftic (Eds.), *Computer-Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Reynolds, J.R. (1970). Transformational Systems and the Algebraic Structure of Atomic Formulas. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*. Edinburgh: Edinburgh University Press.
- Rich, C. & Waters, R.C. (1990). *The Programmer's Apprentice*. New York: ACM Press.
- Rich, E.A. (1979). User Modeling via Stereotypes. *Cognitive Science*, 3, 329-354.
- Robinson, J.A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12, 23-41.
- Self, J. (1974). Student Models in Computer-Aided Instruction. *International Journal of Man-Machine Studies*, 6, 261-276.

- Self, J. (1988). Knowledge, Belief, and User Modelling. In T. O'Shea & V. Sgurev (Eds.), *Artificial Intelligence III: Methodology, Systems, Applications*. New York: North-Holland.
- Shapiro, E.Y. (1983). *Algorithmic Program Debugging*. Cambridge, MA: MIT Press.
- Sleeman, D. & Brown, J.S. (Eds.) (1982). *Intelligent Tutoring Systems*. New York: Academic Press.
- Soloway, E. (1985). From Problems to Programs via Plans: The Content and Structure of Knowledge for Introductory Lisp Programming. *Journal of Educational Computing Research*, 1, 157-172.
- van Someren, M.W. (1985). *Beginners' Problems in Learning Prolog*. Memorandum 54, Department of Experimental Psychology, University of Amsterdam, Amsterdam, The Netherlands.
- van Someren, M.W. (1990). What's Wrong? Understanding Beginners' Problems with Prolog. *Instructional Science*, 19, 257-282.
- Štěpánková, O. & Štěpánek, P. (1987). Developing Logic Programs: Computing Through Normalization. *Proceedings of the 1<sup>st</sup> Workshop on Computer Science Logic*, Karlsruhe, West Germany, 304-321.
- Sterling, L. & Shapiro, E. (1986). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Massachusetts: MIT Press.
- Tamaki, H. & Sato, T. (1984). Unfold/Fold Transformation of Logic Programs. *Proceedings of the 2<sup>nd</sup> International Logic Programming Conference*, Uppsala, Sweden, 127-138.
- Taylor, J.A. (1988). *Programming in Prolog: An In-Depth Study of Problems for Beginners Learning to Program in Prolog*. Ph.D. Dissertation, School of Cognitive and Computing Science, University of Sussex, Brighton, UK.
- Taylor, J. & du Boulay, B. (1987). Studying Novice Programmers: Why They May Find Learning Prolog Hard. In J.C. Rutkowska & C. Crook (Eds.), *Computers, Cognition, and Development: Issues for Psychology and Education*. New York: John Wiley & Sons.
- Tinkham, N.L. (1990). *Induction of Schemata for Program Synthesis*. Ph.D. Dissertation, Department of Computer Science, Duke University, Durham, North Carolina.
- Ullman, J.D. & van Gelder, A. (1988). Efficient Tests for Top-Down Termination of Logical Rules. *Journal of the ACM*, 35, 345-373.

- VanLehn, K. (1988). Student Modeling. In M.C. Polson & J.J. Richardson (Eds.), *Foundations of Intelligent Tutoring Systems*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Vasak, T. & Potter, J. (1986). Characterisation of Terminating Logic Programs. *Proceedings of the 3<sup>rd</sup> International Symposium on Logic Programming*, Salt Lake City, Utah, 140-147.
- Vygotsky, L.S. (1978). *Mind in Society: The Development of Higher Psychological Processes*. Cambridge, MA: Harvard University Press.
- Wills, L.M. (1990). Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence*, 45, 113-171.
- Woolf, B.P. & McDonald, D.D. (1984). Context-Dependent Transitions in Tutoring Discourse. *Proceedings of the 4<sup>th</sup> National Conference on Artificial Intelligence*, Austin, Texas, 355-361.
- Yokomori, T. (1986). Logic Program Forms. *New Generation Computing*, 4, 305-319.
- Zhang, J. & Grant, P.W. (1988). An Automatic Difference-List Transformation Algorithm. *Proceedings of the 8<sup>th</sup> European Conference on Artificial Intelligence*, Munich, West Germany, 320-325.

## Appendix A

# Most Specific Generalizations

In this appendix we provide an algorithm for finding the most specific generalization of a set of compatible simple expressions. A notation is needed to describe the size of an expression. One such measure is the length of an expression. This is simply the number of non-delimiting symbols in the expression. For example, the length of  $\theta(u, \phi(v, w, x, y), z)$  is 8. This notion is formalized in the following definition:

**Definition.** The length of an expression  $t$ , denoted  $\delta(t)$ , is defined inductively as follows:

1.  $\delta(t)=1$  if  $t \in C \cup \mathcal{V}$ .
2.  $\delta(t)=1+\sum_{i=1}^m \delta(t_i)$  if  $t$  is of the form  $\theta(t_1, \dots, t_m)$  where  $\theta \in \mathcal{F} \cup \mathcal{P}$ .
3.  $\delta(t)=m-1+\sum_{i=1}^m \delta(t_i)$  if  $t$  is of the form  $t_1 \wedge \dots \wedge t_m$  or  $t_1 \vee \dots \vee t_m$ .

Algorithms for finding the most specific generalization of a set of compatible simple expressions have been given by Plotkin [10] and Reynolds [11]. Each of these algorithms have a worst-case execution time complexity of  $O(n^2)$ . The following Most Specific Generalization Algorithm finds the most specific generalization  $M$  for compatible simple expressions  $L_1$  and  $L_2$  in  $O(n \log_2 n)$  time.

### Most Specific Generalization Algorithm.

1. Set  $M \leftarrow L_1$ ,  $\sigma_1 \leftarrow \{\}$ ,  $\sigma_2 \leftarrow \{\}$ , and  $R \leftarrow \{\}$ .
2. Assume  $L_1 = p(u_1, \dots, u_n)$  and  $L_2 = p(v_1, \dots, v_n)$  where  $p \in \mathcal{P}$ . Set  $S = \{\langle u_i, v_i, \langle i \rangle \rangle \mid 1 \leq i \leq n \text{ and } u_i \neq v_i\}$ .
3. While  $S \neq \{\}$ :
  - a. Remove the first triple  $T = \langle u, v, \langle N_1, \dots, N_n \rangle \rangle$  from  $S$ .
  - b. Process  $T$ :

If  $u = f(u_1, \dots, u_m)$  and  $v = f(v_1, \dots, v_m)$  where  $f \in \mathcal{F}$  then add the triples

$\langle u_1, v_1, \langle N_1, \dots, N_n, 1 \rangle \rangle, \dots, \langle u_m, v_m, \langle N_1, \dots, N_n, m \rangle \rangle$  to  $S$  for all  $i \in \{1, \dots, m\}$  such that  $u_i \neq v_i$ .

Otherwise, add  $T$  to  $R$ .

4. Sort the triples in  $R$  using all three elements of the triple as the sorting key.
5. Partition  $R$  based on the first two elements of the triple into  $p$  partitions  $R_1, \dots, R_p$ .
6. For each partition  $R_i$ :
  - a. Remove the first triple  $T = \langle u, v, \langle N_1, \dots, N_n \rangle \rangle$  from  $R_i$ .
  - b. Add the binding  $\xi/u$  to  $\sigma_1$  and  $\xi/v$  to  $\sigma_2$  for a new unique variable  $\xi \in \mathcal{V}$ .
  - c. For all triples  $T' = \langle u, v, \langle M_1, \dots, M_m \rangle \rangle$  in  $R_i$ , replace  $u$  with  $\xi$  in the  $\langle M_1, \dots, M_m \rangle^{\text{th}}$  place in  $M$  producing a new  $M$ .

### Example of the Most Specific Generalization Algorithm.

Given  $L_1 = p(f(a, g(y)), x, g(y))$  and  $L_2 = p(f(b, g(x)), x, g(x))$ :

1. Initialize  $M$  and  $\sigma_i$ :
  - a. Set  $M \leftarrow p(f(a, g(y)), x, g(y))$
  - b. Set  $\sigma_1 \leftarrow \{\}$  and  $\sigma_2 \leftarrow \{\}$
2. Initialize  $S$  and  $R$ :
  - a. Set  $S \leftarrow \{\langle f(a, g(y)), f(b, g(x)), \langle 1 \rangle \rangle, \langle g(y), g(x), \langle 3 \rangle \rangle\}$
  - b. Set  $R \leftarrow \{\}$
3. Process  $S$ :
  - a.  $S = \{\langle f(a, g(y)), f(b, g(x)), \langle 1 \rangle \rangle, \langle g(y), g(x), \langle 3 \rangle \rangle\}$   
 $R = \{\}$
  - b.  $S = \{\langle g(y), g(x), \langle 3 \rangle \rangle, \langle a, b, \langle 1, 1 \rangle \rangle, \langle g(y), g(x), \langle 1, 2 \rangle \rangle\}$   
 $R = \{\}$

- c.  $S = \{\langle a, b, \langle 1, 1 \rangle \rangle, \langle g(y), g(x), \langle 1, 2 \rangle \rangle, \langle y, x, \langle 3, 1 \rangle \rangle\}$   
 $R = \{\}$
- d.  $S = \{\langle g(y), g(x), \langle 1, 2 \rangle \rangle, \langle y, x, \langle 3, 1 \rangle \rangle, \langle y, x, \langle 1, 2, 1 \rangle \rangle\}$   
 $R = \{\langle a, b, \langle 1, 1 \rangle \rangle\}$
- e.  $S = \{\langle y, x, \langle 3, 1 \rangle \rangle, \langle y, x, \langle 1, 2, 1 \rangle \rangle\}$   
 $R = \{\langle a, b, \langle 1, 1 \rangle \rangle\}$
- f.  $S = \{\langle y, x, \langle 1, 2, 1 \rangle \rangle\}$   
 $R = \{\langle a, b, \langle 1, 1 \rangle \rangle, \langle y, x, \langle 3, 1 \rangle \rangle\}$
- g.  $S = \{\}$   
 $R = \{\langle a, b, \langle 1, 1 \rangle \rangle, \langle y, x, \langle 3, 1 \rangle \rangle, \langle y, x, \langle 1, 2, 1 \rangle \rangle\}$
4.  $R = \{\langle a, b, \langle 1, 1 \rangle \rangle, \langle y, x, \langle 3, 1 \rangle \rangle, \langle y, x, \langle 1, 2, 1 \rangle \rangle\}$
5.  $R_1 = \{\langle a, b, \langle 1, 1 \rangle \rangle\}$  and  $R_2 = \{\langle y, x, \langle 3, 1 \rangle \rangle, \langle y, x, \langle 1, 2, 1 \rangle \rangle\}$
6. Process each  $R_i$ :
- a. Process  $R_1$ :
- 1)  $R_1 = \{\langle a, b, \langle 1, 1 \rangle \rangle\}$   
 $M = p(f(a, g(y)), x, g(y))$   
 $\sigma_1 = \sigma_2 = \{\}$
- 2)  $R_1 = \{\}$   
 $M = p(f(\xi_1, g(y)), x, g(y))$   
 $\sigma_1 = \{\xi_1/a\}$  and  $\sigma_2 = \{\xi_1/b\}$
- b. Process  $R_2$ :
- 1)  $R_2 = \{\langle y, x, 1, \langle 3, 1 \rangle \rangle, \langle y, x, 1, \langle 1, 2, 1 \rangle \rangle\}$

$$M = p(f(\xi_1, g(y)), x, g(y))$$

$$\sigma_1 = \{\xi_1/a\} \text{ and } \sigma_2 = \{\xi_1/b\}$$

$$2) \quad R_2 = \{\}$$

$$M = p(f(\xi_1, g(\xi_2)), x, g(\xi_2))$$

$$\sigma_1 = \{\xi_1/a, \xi_2/y\} \text{ and } \sigma_2 = \{\xi_1/b, \xi_2/x\}$$

The most specific generalization of the expressions  $p(f(a, g(y)), x, g(y))$  and  $p(f(b, g(x)), x, g(x))$  is  $p(f(\xi_1, g(\xi_2)), x, g(\xi_2))$ .

**Most Specific Generalization Theorem.** Every non-empty set of compatible simple expressions has a unique most specific generalization up to variable renaming.

**Proof:** We must show that the Most Specific Generalization Algorithm works correctly. In order to show that it works correctly, we must show that it terminates for all inputs and that when the algorithm terminates it produces the most specific generalization of  $L_1$  and  $L_2$ . We do this with the following lemmas. In Lemma 1, we show that the algorithm eventually halts. It is shown that the algorithm produces a generalization of  $L_1$  and  $L_2$  in Lemma 2 and then Lemma 3 shows that this generalization is a most specific generalization of  $L_1$  and  $L_2$ .  $\square$

**Lemma 1:** The Most Specific Generalization Algorithm terminates on all valid inputs.

**Proof:** Let  $S_1$  be the contents of  $S$  prior to the execution of step 3a and let  $S_2$  be the contents of  $S$  after the execution of step 3b on an arbitrary pass through the algorithm. Let's define the length of  $S$  to be  $\delta(S) = \sum_{\langle u, v, p \rangle \in S} \delta(u) + \delta(v)$ . Since  $\delta(S) = 0$  iff  $S = \{\}$ , if we can show that  $\delta(S_2) < \delta(S_1)$  then it

follows that the algorithm executes step 3 of the algorithm a finite number of times. Since  $R$  is created by finite additions made on each pass through step 3 of the algorithm, it follows that  $R$  is finite. Since  $R$  is finite, it follows that steps 4-6 will each terminate and thus the algorithm will eventually halt. Thus, we need only show that  $\delta(S_2) < \delta(S_1)$ . Let  $S_1 = \{\langle u_1, v_1, \langle p_1 \rangle \rangle, \dots, \langle u_n, v_n, \langle p_n \rangle \rangle\}$  (i.e.,  $S = \{\langle u_1, v_1, \langle p_1 \rangle \rangle, \dots, \langle u_n, v_n, \langle p_n \rangle \rangle\}$ ) prior to the execution of step 3a). The algorithm selects the triple  $\langle u_1, v_1, \langle p_1 \rangle \rangle$  in step 3a. We have two cases to consider:

1.  $u_1 = f(u_{1,1}, \dots, u_{1,m})$  and  $v_1 = f(v_{1,1}, \dots, v_{1,m})$  where  $f \in \mathcal{F}$

In this case, the algorithm adds triples  $\langle u_{1,1}, v_{1,1}, \langle p_1, 1 \rangle \rangle, \dots, \langle u_{1,m}, v_{1,m}, \langle p_1, m \rangle \rangle$  to  $S$  prior to looping back to the top of the while loop. Thus,  $S_2 = \{\langle u_2, v_2, \langle p_2 \rangle \rangle, \dots, \langle u_n, v_n, \langle p_n \rangle \rangle, \langle u_{1,1}, v_{1,1}, \langle p_1, 1 \rangle \rangle, \dots, \langle u_{1,m}, v_{1,m}, \langle p_1, m \rangle \rangle\}$  (i.e.,  $S = \{\langle u_2, v_2, \langle p_2 \rangle \rangle, \dots, \langle u_n, v_n, \langle p_n \rangle \rangle, \langle u_{1,1}, v_{1,1}, \langle p_1, 1 \rangle \rangle, \dots, \langle u_{1,m}, v_{1,m}, \langle p_1, m \rangle \rangle\}$ ) after the execution of step 3b).

So,  $\delta(S_2) = \delta(S_1) - (\delta(u_1) + \delta(v_1)) + \sum_{i=1}^m \delta(u_{1,i}) + \delta(v_{1,i})$ . But this reduces to  $\delta(S_2) = \delta(S_1) - 2$  so it follows that

$$\delta(S_2) < \delta(S_1).$$

2. Otherwise

In this case, the triple  $\langle u_1, v_1, \langle p_1 \rangle \rangle$  is removed from  $S$  and nothing is added to  $S$  so clearly  $\delta(S_2) < \delta(S_1)$ .

Thus, after each iteration of the algorithm, the length of  $S$  decreases. Since the length of  $S$  is 0 (i.e.,  $\delta(S) = 0$ ) iff  $S = \{\}$ , it follows that the while loop of step 3 eventually halts. Furthermore, since  $R$  is created by adding finite sets of triples within the while loop, it is clear that  $R$  is finite and thus the number of partitions,  $p$ , that  $R$  can be subdivided into is finite. Then it follows that step 6 is a finite loop. Thus, the Generalization Algorithm halts given any pair of compatible simple expressions.  $\square$

**Lemma 2:** When the algorithm terminates,  $L_1 = M\sigma_1$  and  $L_2 = M\sigma_2$ .

**Proof:** Obviously,  $L_1 = M\sigma_1$  since  $M$  is constructed from  $L_1$  by applying the substitutions in  $\sigma_1$  in a reverse manner. And since each of the variables which are added in step 6b are unique, the replacement operation of step 6c is the inverse operation of substitution.

We provide a proof by contradiction that  $L_2 = M\sigma_2$ . Assume  $L_2 \neq M\sigma_2$ . Then there exists a place  $J = \langle i_1, \dots, i_j \rangle$  such that the term  $v_j$  in the  $J^{\text{th}}$  place in  $L_2$  is the smallest term that is different from the term  $w_j$  in the  $J^{\text{th}}$  place in  $M\sigma_2$ . By smallest term, we mean that  $\delta(v_j) \leq \delta(v_k)$  for any term  $v_k$  in the  $K^{\text{th}}$  place in  $L_2$  that is different from the term  $w_k$  in the  $K^{\text{th}}$  place in  $M\sigma_2$ .

Let  $u_j$  be the term in the  $J^{\text{th}}$  place in  $L_1$ . Note that if the triple  $\langle u_j, v_j, J \rangle$  is added to  $R$  in step 3b then  $v_j = w_j$  after step 6c. Thus, it follows that the triple  $\langle u_j, v_j, J \rangle$  is not added to  $R$ . But there are only two cases in which  $\langle u_j, v_j, J \rangle$  is not added to  $R$ : either  $u_j = v_j$  or  $\langle u_j, v_j, J \rangle$  is never added to  $S$ .

Consider the first case:  $u_j = v_j$ . If  $u_j = v_j$  then the term in the  $J^{\text{th}}$  place in  $L_1$  is identical to the term in the  $J^{\text{th}}$  place in  $L_2$ . But then the term in the  $J^{\text{th}}$  place in  $L_1$  must be identical to the term in the  $J^{\text{th}}$  place in  $M$  since  $\langle u_j, v_j, J \rangle$  is not added to  $R$ . Since  $\xi$  is a unique variable in the bindings  $\xi/u_i$  and  $\xi/v_i$  which is added to  $\sigma_2$  in step 6b and since the term in the  $J^{\text{th}}$  place in  $M$  is only changed if  $\langle u_j, v_j, J \rangle$  is added to  $R$ , it follows that the term in the  $J^{\text{th}}$  place in  $M$  is identical to the term in the  $J^{\text{th}}$  place in  $M\sigma_2$ . Thus, the term in the  $J^{\text{th}}$  place in  $L_2$  is identical to the term in the  $J^{\text{th}}$  place in  $M\sigma_2$  which contradicts the claim that  $J$  exists.

Consider the second case:  $\langle u_j, v_j, J \rangle$  is never added to  $S$ . If  $\langle u_j, v_j, J \rangle$  is never added to  $S$  then the triple  $\langle u_k, v_k, K \rangle$  (where  $K = \langle i_1, \dots, i_k \rangle$  and  $u_k$  is in the  $K^{\text{th}}$  position of  $L_1$  and  $v_k$  is in the  $K^{\text{th}}$  position in  $L_2$  for some  $k < j$ ) is added to  $R$  or  $u_j = v_j$ . We have already considered the case when  $u_j = v_j$ . If  $\langle u_k, v_k, K \rangle$  is added to  $R$  then the binding  $\xi/v_k$  is added to  $\sigma_2$  and the term in the  $K^{\text{th}}$  place in  $M$  is replaced with  $\xi$ . Thus, the term in the  $J^{\text{th}}$  place in  $L_2$  is identical to the term in the  $J^{\text{th}}$  place in  $M\sigma_2$

which contradicts the claim that  $J$  exists. Since there are no other cases to consider, no such  $J$  exists. Thus,  $L_2 = M\sigma_2$ .  $\square$

**Lemma 3:** Any generalization of  $L_1$  and  $L_2$  is also a generalization of the most specific generalization  $M$  produced by the Generalization Algorithm.

**Proof:** Assume there exists a  $G$  such that the expression obtained from the Most Specific Generalization Algorithm,  $M$ , is a generalization of  $G$ . Then there are the following possibilities for  $\sigma$  (where  $G = M\sigma$ ):

1.  $\sigma = \{ \}$

In this case,  $G = M$  so it follows that  $G$  is a generalization of  $M$  (since a simple expression is a generalization of itself).

2.  $\sigma = \{v_1/t_1, \dots, v_n/t_n\}$  where  $\forall i \in \{1, \dots, n\} t_i \in \mathcal{V}$  and where  $\forall j, k \in \{1, \dots, n\} j \neq k$  implies  $t_j \neq t_k$

In this case,  $G$  is an alphabetic variant of  $M$ . Letting  $\mu = \{t_1/v_1, \dots, t_n/v_n\}$ , it is clear to see that  $G\mu = M$  and thus  $G$  is a generalization of  $M$ .

3.  $\sigma = \{v_1/t_1, \dots, v_n/t_n\}$  where  $\exists i \in \{1, \dots, n\}$  such that  $t_i \notin \mathcal{V}$

In this case,  $G$  contains a non-variable term  $t_i$  in some place in which  $M$  contains a variable term  $v_i$ . Since  $G$  is a generalization of  $L_1$  and  $L_2$ , the only way  $G$  can contain a non-variable term  $t_i$  in some place is if both  $L_1$  and  $L_2$  contain the same non-variable term  $t_i$  in that same place. Thus, the algorithm must be adding an unnecessary substitution since the only replacements in the algorithm involve replacing terms of  $X_i$  with variables. But this is not possible since the algorithm only adds triples to  $R$  if the terms in the corresponding places in  $L_1$  and  $L_2$  differ and substitutions are only created for triples added to  $R$ . Therefore, no such  $G$  exists.

4.  $\sigma = \{v_1/t_1, \dots, v_n/t_n\}$  where  $\exists j, k \in \{1, \dots, n\}$  such that  $j \neq k$  and  $t_j = t_k$

Assume  $t_j, t_k \in \mathcal{V}$ , otherwise the previous case applies. Since  $G$  is a generalization of  $L_1$  and  $L_2$ , the term pairs at the places where  $t_j$  and  $t_k$  occur must have been the same. Thus, in this case the algorithm must have made two distinct replacements for two different places that have the same term pair. But this is not possible since all common occurrences of term pairs are partitioned into the same group in step 5 and assigned the same variable in step 6. Therefore, no such  $G$  exists.

Since there are no other possibilities for  $\sigma$ , it follows that every generalization of  $L_1$  and  $L_2$  must also be a generalization of  $M$ . Thus, by the definition of most specific generalizations it follows that  $M$  must be the most specific generalization of  $L_1$  and  $L_2$ .  $\square$

Note that the Generalization Algorithm is easily extended to well-formed formulas and even second-order generalizations.

## Appendix B

# Sample Analyses from the ADAPT Debugger

ADAPT has been tested on over 100 distinct variations of the `reverse/2` program, including the 55 programs given in Looi's dissertation (Looi, 1988) plus an additional set of 70 programs. ADAPT's analysis of each of these programs follows. ADAPT's transformed normal form program is provided for each incorrect program.<sup>19</sup> The only normal form program given to ADAPT for `reverse/2` was:

```
reverse([], []).
reverse([H|T], L) :- reverse(T, M), append(M, [H], L).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

The first 55 variations are taken from Looi's dissertation. This set of programs included only 4 correct implementations: variations #1, #4, #8, and #17. The remaining set is provided to show the robustness of the ADAPT debugger. Note that this set contains over 40 unique correct variations of `reverse/2`: variations #83 - #125.

### Variation #1 (Standard Accumulator)

```
reverse(X, Y) :- reverse(X, [], Y).
reverse([], L, L).
reverse([H|T], Y, Z) :- reverse(T, [H|Y], Z).
```

### Variation #2

```
reverse(X, Y) :- reverse(X, _, Y).
reverse([], L, L).
reverse([H|T], Y, Z) :- reverse(T, [H|Y], Z).
```

### Transformed Normal Form:

```
reverse(X, Y) :- reverse(X, [], Y).
reverse([], L, L).
reverse([H|T], Y, Z) :- reverse(T, [H|Y], Z).
```

---

<sup>19</sup>Note that the transformed normal form program is identical to the student's program whenever the student's program is correct.

**Bugs Found:**

*Variable for constant in 2nd argument of 1st subgoal of 1st clause in reverse/2*

**Variation #3**

```
reverse([],L,L).
reverse([],[]).
reverse(X,Y) :- reverse(X,[],Y).
reverse([H|T],Y,Z) :- reverse(T,[H|Y],Z).
```

**Transformed Normal Form:**

```
reverse([],L,L).
reverse(X,Y) :- reverse(X,[],Y).
reverse([H|T],Y,Z) :- reverse(T,[H|Y],Z).
```

**Bugs Found:**

*Redundant base case in 2nd clause in reverse/2*

**Variation #4 (Standard Accumulator)**

```
reverse(X,Y) :- rev(X,[],Y).
rev([],L,L).
rev([H|T],Y,Z) :- rev(T,[H|Y],Z).
```

**Variation #5**

```
reverse(X,Y) :- reverse(X,[],Y).
reverse([],L,L).
reverse([H1|T1],[H2|T2],Z) :- reverse(T,[H2,H1|T1],Z).
```

**Transformed Normal Form:**

```
reverse(X,Y) :- reverse(X,[],Y).
reverse([],L,L).
reverse([H|T],Y,Z) :- reverse(T,[H|Y],Z).
```

**Bugs Found:**

*Constant for variable in 2nd argument of head of 2nd clause in reverse/3*  
*Variable mismatch in 1st argument in 1st subgoal of 2nd clause in reverse/3*  
*Variable mismatch in 1st part of 2nd argument in 1st subgoal of 2nd clause in*

```
reverse/3
Invalid list in 2nd argument in 1st subgoal of 2nd clause in reverse/3
```

## Variation #6

```
reverse(List Tsil) :- reverse(List,[],Tsil).
reverse([X|List],Temp,[X|Temp]) :- atom(List).
reverse([X|List],Temp,Temp) :- \+atom(List),reverse(List,[X|Temp],[X|Temp]).
```

### Transformed Normal Form:

```
reverse(X,Y) :- reverse(X,[],Y).
reverse([],L,L).
reverse([H|T],Y,Z) :- reverse(T,[H|Y],Z).
```

### Bugs Found:

```
Constant mismatch in 1st argument of head of 1st clause in reverse/3
Constant for variable in 3rd argument of head of 1st clause in reverse/3
Variable mismatch in 3rd argument of head of 2nd clause in reverse/3
Constant for variable in 3rd argument of 2nd subgoal of 2nd clause in reverse/3
```

### Comments:

*This program is clearly incorrect. Although it is not clear what the student intended in the recursive clause, the only problem with the base case is that it cannot handle the empty list case.*

## Variation #7

```
reverse(X,Y):-reverse(X,[],Y).
reverse([],L,L).
reverse([H|L],[X|L1],L2):-reverse(L,L1,L2).
```

### Transformed Normal Form:

```
reverse(A,B) :- reverse(A,[],B).
reverse([],A,A).
reverse([A|B],C,D) :- reverse(B,[A|C],D).
```

### Bugs Found:

```
Constant for variable in 2nd argument of head of 2nd clause in reverse/3
Variable for constant in 2nd argument of 1st subgoal of 2nd clause in reverse/3
```

## Variation #8 (Railway-Shunt)

```
reverse(A,B):-rev(A,[B]).
rev([], [B|_]).
rev([Head|Tail],[B|Temp]):-rev(Tail,[B,Head|Temp]).
```

### Variation #9 (Inverse Naive)

```
reverse([], []).
reverse(L, [H|RT]) :- append(T, [H], L), reverse(T, RT).
```

#### Bugs Found:

*Missing definition for append/3*

#### Comments:

*The student is asked to give a definition for append/3. In this case, if the student gives a correct program for the standard append/3:*

```
append([ ], A, A).
append([A|B], C, [A|D]) :- append(B, C, D).
```

*then the reverse/2 program will be correct.*

### Variation #10

```
reverse([], Y).
reverse(X, Y) :- reverse([H|T], Y).
```

#### Bugs Found:

*Predicate reverse/2 is not structural recursive*

#### Comments:

*The student is asked to re-enter her program for reverse/2. Hopefully, her next attempt would be more understandable. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

### Variation #11

```
reverse([], []).
reverse([HX|X], Y) :- append(Y, HX).
```

*on*

**Bugs Found:**

*Missing definition for append/2*

**Comments:**

*The student is asked to give a definition for append/2. In this case, there is no way that the student can produce a definition for append/2 that would make the program correct. ADAPT does not know this, however, it will determine that the student's program is incorrect when she enters a definition for append/2.*

**Variation #12**

```
reverse([], []).
reverse([H1|T1],[H2|T2]):-reverse(T1,T2),H2 is H1.
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Constant for variable in 2nd argument of head of 2nd clause in reverse/2  
Variable mismatch in 1st argument of 2nd subgoal of 2nd clause in reverse/2  
Variable for constant in 2nd argument of 2nd subgoal of 2nd clause in reverse/2  
Missing argument in 2nd subgoal of 2nd clause in reverse/2*

**Comments:**

*In this case, ADAPT is unable to make sense of the student's H2 is H1 subgoal, so it attempts to match it to its normal form subgoal claiming that the first two arguments are incorrect. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

**Variation #13**

```
reverse([], []).
reverse([H|T],[H|Y]):-reverse(T,Y).
```

on

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Constant for variable in 2nd argument of head of 2nd clause in reverse/2*  
*Missing subgoal in 2nd subgoal of 2nd clause in reverse/2*

**Variation #14**

```
reverse([],[]).
reverse([H1|T],[H2|Y]):-reverse(T,Y).
```

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Constant for variable in 2nd argument in head of 2nd clause in reverse/2*  
*Missing subgoal in 2nd subgoal of 2nd clause in reverse/2*

**Variation #15**

```
reverse([],Y).
reverse([XH|XT],Y):-reverse(XT,Y),append([Y|XH]).
append(X).
```

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Variable for constant in 2nd argument in head of 1st clause in reverse/2*  
*Variable mismatch in 2nd argument in 1st subgoal of 2nd clause in reverse/2*

on

*Missing subgoal in 2nd subgoal of 2nd clause in reverse/2*

### Comments:

*Notice that the `append([Y|XH])` subgoal unfolds away in the student's 2<sup>nd</sup> clause leaving ADAPT to assume that the student forgot the `append` subgoal rather than providing an incorrect `append` subgoal.*

### Variation #16

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, [H], Res).
```

### Bugs Found:

*Missing definition for `append/3`*

### Comments:

*The student is asked to give a definition for `append/3`. In this case, if the student gives a correct program for the standard `append/3`:*

```
append([], A, A).
append([A|B], C, [A|D]) :- append(B, C, D).
```

*then the `reverse/2` program will be correct.*

### Variation #17 (Naive)

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, [H], Res).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

### Variation #18

```
app([], L, L).
app([H|L1], L2, [H|L3]) :- app(L1, L2, L3).
reverse([], []).
reverse([H|T], Res) :- app(Sofar, [H], Res), reverse(T, Sofar).
```

### Bugs Found:

*on*

*Predicate app/3 is not structural recursive*

### Comments:

*The student is asked to re-enter her program for reverse/2. In this case, if the student re-enters the same program with the subgoals in the 2<sup>nd</sup> clause of reverse/2 switched then the reverse/2 program would be correct. Note that this program is existentially terminating since app/3 is coded with it's base case first, but it is not universally terminating.*

### Variation #19

```
app1([H|L1],L2,[H|L3]):-app1(L1,L2,L3).
app1([],L,L).
reverse([],[]).
reverse([H|T],Res):-app1(Sofar,[H],Res),reverse(T,Sofar).
```

### Bugs Found:

*Predicate app/3 is not structural recursive*

### Comments:

*The student is asked to re-enter her program for reverse/2. In this case, if the student re-enters the same program with the subgoals in the 2<sup>nd</sup> clause of reverse/2 switched then the reverse/2 program would be correct. Note that this program is neither existentially terminating nor universally terminating.*

### Variation #20

```
reverse([],[]).
reverse([H|T],Res):-reverse(T,Sofar),append(Sofar,[H],Res).
append([],L,L).
append([H|L1],L2,[H,L3]):-append(L1,L2,L3).
```

### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

### Bugs Found:

*on*

*Invalid list in 3rd argument of head of 2nd clause in append/3*

## Variation #21

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, H, Res).
```

### Bugs Found:

*Missing definition for append/3*

### Comments:

*The student is asked to give a definition for append/3. In this case, if the student gives a correct program for an enqueue-like append/3 (where a single element is appended to the end of the list):*

```
append([], A, [A]).
append([A|B], C, [A|D]) :- append(B, C, D).
```

*then the reverse/2 program will be correct. Note that unlike APROPOS2, ADAPT permits this correct version of reverse/2.*

## Variation #22

```
reverse([], []).
reverse([XH|XT], Y) :- reverse(XT, Y), append(XH, XT, Y).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

### Transformed Normal Form:

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], A, A).
append([A|B], C, [A|D]) :- append(B, C, D).
```

### Bugs Found:

*Variable mismatch in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Variable mismatch in 1st argument in 2nd subgoal of 2nd clause in reverse/2  
Variable for constant in 2nd argument in 2nd subgoal of 2nd clause in reverse/2*

### Comments:

*In this case, ADAPT is unable to make sense of the student's append(XH, XT, Y) subgoal, so it*

*on*

*attempts to match it to its normal form subgoal claiming that the first two arguments are incorrect. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

### Variation #23

```
reverse([], []).
reverse([XH|XT], Y) :- reverse(XT, Y), append([XH], XT, Y).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

#### Transformed Normal Form:

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], A, A).
append([A|B], C, [A|D]) :- append(B, C, D).
```

#### Bugs Found:

*Constant for variable in 2nd argument in head of 2nd clause in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal of 2nd clause in reverse/2*

#### Comments:

*In this case, ADAPT is unable to make sense of the student's `append([XH], XT, Y)` subgoal, so it attempts to match it to its normal form subgoal claiming that the first two arguments are incorrect. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

### Variation #24

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, H, Res).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append([L1, L2, L3]).
```

#### Bugs Found:

*Missing definition for `append/1`, but there is a definition for `append/3`*

*on*

- perhaps the `append([L1,L2,L3])` subgoal has missing arguments

### Comments:

*ADAPT notices that the student's program has a missing definition for `append/1`, but does have a definition for `append/3`. It hypothesizes that the `append([L1,L2,L3])` subgoal has missing arguments which is close to being correct (in this case, the arguments are all there but they are enclosed in list brackets). When this subgoal is corrected, ADAPT reanalyzes the program.*

### Variation #25

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, H, Res).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

### Transformed Normal Form:

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, A, C).
append([], A, [A]).
append([A|B], C, [A|D]) :- append(B, C, D).
```

### Bugs Found:

*Variable for constant in 3rd argument in head of 1st clause in `append/3`*

### Comments:

*ADAPT assumes the `append/3` program is incorrect rather than assuming that the subgoal invocation `append(Sofar, H, Res)` is incorrect. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

### Variation #26

```
reverse([], []).
reverse([H|T], Res) :- reverse(T, Sofar), append(Sofar, H, Res).
append([], L, L).
append([H|L1], L2, [H, L3]) :- append(L1, L2, L3).
```

### Transformed Normal Form:

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, A, C).
```

on

```
append([],A,[A]).
append([A|B],C,[A|D]) :- append(B,C,D).
```

### Bugs Found:

*Variable for constant in 3rd argument in head of 1st clause in append/3  
Invalid list in 3rd argument in head of 2nd clause in append/3*

### Comments:

*First of all, note that ADAPT has identified an "invalid list" in the 3<sup>rd</sup> argument of append/3 (i.e., the student has incorrectly attempted to append an element and a list). This is an example of a compound bug (see Section 3.3.2.2). Secondly, ADAPT assumes the append/3 program is incorrect rather than assuming that the subgoal invocation append(Sofar,H,Res) is incorrect. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

### Variation #27

```
reverse([],[]).
reverse([H|T],Res):-append(Sofar,[H],Res),reverse(T,Sofar).
```

### Bugs Found:

*Missing definition for append/3*

### Comments:

*The student is asked to give a definition for append/3. In this case, if the student gives a correct program for the standard append/3:*

```
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

*then the reverse/2. program will not be structural recursive. ADAPT does not know this, however, it will determine that the student's program is not structural recursive when she enters a definition for append/3.*

### Variation #28

```
reverse([],[]).
reverse(X,Y):-X=[H|T],append(Y,H,Newlist),reverse(T,Newlist).
```

on

**Bugs Found:**

*Missing definition for append/3*

**Variation #29**

```
reverse([], []).
reverse([H|T], Res) :- append(Sofar, [H], Res), reverse(T, Sofar).
append([], List, List).
append([H|T], List2, Result) :- Result=[H|X], append(T, List2, X).
```

**Bugs Found:**

*Predicate append/3 is not structural recursive*

**Comments:**

*The student is asked to re-enter her program for reverse/2. In this case, if the student re-enters the same program with the subgoals in the 2nd clause of reverse/2 switched then the reverse/2 program would be correct.*

**Variation #30**

```
reverse([], []).
reverse([Head|Tail], X) :- append(Y, [Head], X), reverse(Y, Tail).
append([], List, List).
append([H|T], List2, Result) :- Result=[H|X], append(T, List2, X).
```

**Bugs Found:**

*Predicates reverse/2 & append/3 are not structural recursive*

**Comments:**

*The student is asked to re-enter her program for reverse/2. Note that even if the student re-enters the same program with the subgoals in the 2nd clause of reverse/2 switched then the reverse/2 program would be still not be structural recursive. The problem that remains is that the arguments in the reverse(Y, Tail) need to be switched as well.*

**Variation #31**

```
reverse([], []).
reverse([Head|Tail], X) :- conc(Y, [Head], X), reverse(Tail, Y).
```

**Bugs Found:**

*Missing definition for conc/3*

**Comments:**

*The student is asked to give a definition for append/3. In this case, if the student gives a correct program for the standard append conc/3:*

```
conc([], A, A).
conc([A|B], C, [A|D]) :- conc(B, C, D).
```

*then the reverse/2 program will not be structural recursive. ADAPT does not know this, however, it will determine that the student's program is not structural recursive when she enters a definition for conc/3.*

**Variation #32**

```
reverse([], []).
reverse([H|T], X) :- append(X, H, T), reverse(T, Y).
```

**Bugs Found:**

*Missing definition for append/3*

**Variation #33**

```
reverse([], []).
reverse([H|T], X) :- add(H, X, Y), reverse(T, Y).
```

**Bugs Found:**

*Missing definition for add/3*

**Comments:**

*on*

The student is asked to give a definition for `add/3`. In this case, if the student gives a correct program for the standard `append/3`:

```
add([],A,A).
add([A|B],C,[A|D]) :- add(B,C,D).
```

then the `reverse/2` program will not be structural recursive. ADAPT does not know this, however, it will determine that the student's program is not structural recursive when she enters a definition for `add/3`.

### Variation #34

```
reverse([],[]).
reverse([H|T],Y) :- append(H,Y,Y),reverse(T,Y).
```

#### Bugs Found:

*Missing definition for `append/3`*

#### Comments:

The student is asked to give a definition for `append/3`. In this case, there is no way that the student can produce a definition for `append/3` that would make the program correct. ADAPT does not know this, however, it will determine that the student's program is incorrect when she enters a definition for `append/3`.

### Variation #35

```
reverse([],[]).
reverse([H|T],Res) :- append(Sofar,H,Res),reverse(T,Sofar).
```

#### Bugs Found:

*Missing definition for `append/3`*

#### Comments:

on

The student is asked to give a definition for `append/3`. In this case, if the student gives a correct program for the standard `append/3`:

```
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

then the `reverse/2` program will not be structural recursive. ADAPT does not know this, however, it will determine that the student's program is not structural recursive when she enters a definition for `append/3`.

### Variation #36

```
reverse([],[]).
reverse([H|T1],[T2|H]) :- reverse(T1,T2).
```

#### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

#### Bugs Found:

*Invalid append in 2nd argument in head of 2nd clause in reverse/2*

#### Comments:

*Note that the invalid append bug subsumes the missing subgoal and constant for variable bugs that Result from a term by term comparison between the student and transformed normal form programs.*

### Variation #37

```
reverse([],Y).
reverse([H|T1],[T2|H]) :- reverse(T1,T2).
```

#### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

#### Bugs Found:

on

*Variable for constant in 2nd argument in head of 1st clause in reverse/2  
Invalid append in 2nd argument in head of 2nd clause in reverse/2*

### Comments:

*Note that the invalid append bug subsumes the missing subgoal and constant for variable bugs that Result from a term by term comparison between the student and transformed normal form programs.*

### Variation #38

```
reverse([], []).
reverse([N],[N]).
reverse([A|B],[C|A]):-reverse(B,C).
```

### Transformed Normal Form:

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

### Bugs Found:

*Redundant base case in 2nd clause in reverse/2  
Invalid append in 2nd argument in head of 3rd clause in reverse/2*

### Comments:

*Note that the invalid append bug subsumes the missing subgoal and constant for variable bugs that Result from a term by term comparison between the student and transformed normal form programs.*

### Variation #39

```
reverse([], []).
reverse([X],[X]).
reverse([Head|Tail],[Rev_Tail|Head]):-reverse(Tail,[Rev_Tail]).
```

### Transformed Normal Form:

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
```

```
append([A|B],C,[A|D]) :- append(B,C,D).
```

### Bugs Found:

*Redundant base case in 2nd clause in reverse/2  
 Constant for variable in 2nd argument in head of 3rd clause in reverse/2  
 Constant for variable in 2nd argument in 1st subgoal of 3rd clause in reverse/2  
 Missing subgoal in 2nd subgoal in 3rd clause in reverse/2*

### Comments:

*Note that this is actually another case of an invalid append (see Variation #37-#38), but ADAPT is unable to detect it because it is not clear that Rev\_Tail\_Contents is actually the contents of the reverse of the tail since the student enclosed it in list brackets.*

### Variation #40

```
reverse([],[]).
reverse([Head|Tail],[X|Head]):-reverse(X,Tail).
```

### Bugs Found:

*Predicate reverse/2 is not structural recursive*

### Comments:

*Note that this is actually another case of an invalid append (see Variation #37-#38), but ADAPT is unable to detect it because the student has switched the order of the arguments in the recursive call.*

### Variation #41

```
append([],L,L).
append([H],L,[H|L]).
append([H|T],L,Ans):-append(T,L,X),append([H],X,Ans).
reverse([],[]).
reverse([H],[H]).
reverse([H|T],L):-reverse(T,M),append(M,[M|L],L).
```

### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
```

on

```
append([A|B],C,[A|D]) :- append(B,C,D).
```

### Bugs Found:

*Redundant base case in 2nd clause in reverse/2  
Subgoal mismatch in 2nd subgoal in 3rd clause in reverse/2*

### Comments:

*Note that append/3 actually violates standard recursive form since its third clause contains multiple recursive invocations, but ADAPT does detect an error in the student's program since it is unable to match the student's `append(M,[M|L],L)` subgoal with the normal form `append(D,[A],C)` subgoal. Further note that student's definition of `append/3` actually works except that it produces an infinite number of identical correct solutions since the 2<sup>nd</sup> subgoal of the 3<sup>rd</sup> clause unifies with both the 2<sup>nd</sup> and the 3<sup>rd</sup> clauses.*

### Variation #42

```
append([],L,L).
append([H],L,[H|L]).
append([H|T],L,Ans):-append(T,L,X),append([H],X,Ans).
reverse([],[]).
reverse([H],[H]).
reverse([H|T],L):-reverse(T,M),append(M,H,L).
```

### Transformed Normal Form:

```
append([],A,[A]).
append([A|B],C,[A|D]) :- append(B,C,D).
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,A,C).
```

### Bugs Found:

*Redundant base case in 2nd clause in reverse/2  
Variable for constant in 3rd argument in head of 1st clause in append/3  
Incorrect extra base case in 2nd clause in append/3  
Variable for constant in 3rd argument in head of 3rd clause in append/3  
Extra subgoal in 2nd subgoal in 3rd clause in append/3*

### Comments:

*Note that in this case, ADAPT analyzes `append/3` even though it is not in standard recursive form. In its analysis, however, ADAPT mistakenly assumes that the 2<sup>nd</sup> recursive invocation is actually*

*on*

*an extra subgoal. Note that ADAPT claims that the 2<sup>nd</sup> clause of append/3 is an incorrect extra base case since the student has coded the append subgoal with its second argument as an element rather than a list so the clause should be `append([H],L,[H,L])`.*

### Variation #43

```
append([],L,L).
append([H],L,[H|L]).
append([H|T],L,Ans):-append(T,L,X),append([H],X,Ans).
reverse([],[]).
reverse([H],[H]).
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
```

### Transformed Normal Form:

```
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
```

### Bugs Found:

*Redundant base case in 2nd clause in reverse/2  
 Redundant base case in 2nd clause in append/3  
 Variable for constant in 3rd argument in head of 3rd clause in append/3  
 Extra subgoal in 2nd subgoal in 3rd clause in append/3*

### Comments:

*Note that in this case, ADAPT analyzes append/3 even though it is not in standard recursive form. In its analysis, however, ADAPT mistakenly assumes that the 2<sup>nd</sup> recursive invocation is actually an extra subgoal.*

### Variation #44

```
reverse([],L).
reverse([H],Ans):-Ans=[Ans|H].
reverse([H|T],Ans):-Ans=[Ans|H],reverse(T,Ans).
```

### Bugs Found:

*"Reassignment" error in subgoal `Ans=[Ans|H]` in clause:  
`reverse([H|T],Ans):-Ans=[Ans|H],reverse(T,Ans).`*

on

**Variation #45**

```
reverse([],Y).
reverse([H|T],Y):-reverse(T,[H|Y]).
```

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Variable for constant in 2nd argument in head of 1st clause in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal in 2nd clause in reverse/2*

**Comments:**

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

**Variation #46**

```
reverse([],Y).
reverse([H|T],Y):-reverse(T,[H|_]).
```

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Variable for constant in 2nd argument in head of 1st clause in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal in 2nd clause in reverse/2*

**Comments:**

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

**Variation #47**

```
reverse([], []).
reverse([H|T], Y) :- reverse(T, [H|Y]).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

**Bugs Found:**

*Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal in 2nd clause in reverse/2*

**Comments:**

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

**Variation #48**

```
reverse([], []).
reverse(X, Y) :- X=[H|T], reverse(T, [H|Y]).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

**Bugs Found:**

*Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal in 2nd clause in reverse/2*

**Comments:**

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is*

*on*

*needed here.*

### Variation #49

```
reverse([], []).
reverse(X, Y) :- X=[H|T], New=[H|Y], reverse(T, New).
```

#### Transformed Normal Form:

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

#### Bugs Found:

*Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal in 2nd clause in reverse/2*

#### Comments:

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

### Variation #50

```
reverse([], []).
reverse([], Y).
reverse([H|T], [X]) :- reverse(T, [X|H]).
```

#### Transformed Normal Form:

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

#### Bugs Found:

*Incorrect extra base case in 2nd clause in reverse/2*

*on*

Constant for variable in 2nd argument in head of 2nd clause in reverse/2  
 Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
 Missing subgoal in 2nd subgoal in 2nd clause in reverse/2

### Comments:

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

### Variation #51

```
reverse([],[]).
reverse([],Y).
reverse([H|T],Y):-reverse(T,[H|Y]).
reverse(Y,[]).
```

### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

### Bugs Found:

Incorrect extra base case in 2nd clause in reverse/2  
 Constant for variable in 2nd argument in 1st subgoal of 3rd clause in reverse/2  
 Missing subgoal in 2nd subgoal of 3rd clause in reverse/2  
 Incorrect extra base case in 4th clause in reverse/2

### Comments:

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

### Variation #52

```
reverse([],[]).
reverse([],Y).
reverse([H|T],[X]):-reverse(T,[H|X]).
```

### Transformed Normal Form:

on

162

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

### Bugs Found:

*Incorrect extra base case in 2nd clause in reverse/2  
Constant for variable in 2nd argument in head of 3rd clause in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 3rd clause in reverse/2  
Missing subgoal in 2nd subgoal of 3rd clause in reverse/2*

### Comments:

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

### Variation #53

```
reverse([],Y).
reverse([],[]).
reverse([H|T],Y):-reverse(T,[H|Y]).
```

### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

### Bugs Found:

*Incorrect extra base case in 2nd clause in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 3rd clause in reverse/2  
Missing subgoal in 2nd subgoal of 3rd clause in reverse/2*

### Comments:

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

### Variation #54

on

```
reverse([],Y).
reverse([],[]).
reverse(A,B):-recur(A,B).
recur([H|T],Y):-recur(T,[H|Y]).
```

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Incorrect extra base case in 1st clause in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 3rd clause in reverse/2  
Missing subgoal in 2nd subgoal of 3rd clause in reverse/2*

**Comments:**

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

**Variation #55**

```
reverse([H|T],Y):-reverse(T,[H|Y]).
```

**Transformed Normal Form:**

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Missing base case in reverse/2  
Constant for variable in 2nd argument in 1st subgoal of 2nd clause in reverse/2  
Missing subgoal in 2nd subgoal of 2nd clause in reverse/2*

**Comments:**

*on*

*Clearly the program is incorrect, but it is not clear what the student intended. Intention Tracing is needed here.*

### Variation #56

```
reverse([H|T],L):-append(M,[H],L),reverse(T,M).
reverse([],[]).
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
```

#### Bugs Found:

*Predicate reverse/2 is not structural recursive*

### Variation #57

```
reverse(L,R):-enqueue(H,M,R),reverse(T,M),append([H],T,L).
reverse([],[]).
enqueue(X,[],[X]).
enqueue(Y,[A|B],[A|Z]):-enqueue(Y,B,Z).
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
```

#### Bugs Found:

*Predicate enqueue/3 is not structural recursive.*

### Variation #58

```
reverse([H|T],L):-enqueue(M,H,L),reverse(T,M).
reverse([],[]).
enqueue([],X,[X]).
enqueue([A|B],Y,[A|Z]):-enqueue(B,Y,Z).
```

#### Bugs Found:

*Predicate enqueue/3 is not structural recursive.*

### Variation #59

*on*

```
reverse([], []).
reverse([H|T], L) :- enqueue(H, M, L), reverse(T, M).
enqueue(X, [], [X]).
enqueue(Y, [A|B], [A|Z]) :- enqueue(Y, B, Z).
```

**Bugs Found:**

*Predicate enqueue/3 is not structural recursive.*

**Variation #60**

```
reverse([H|T], L) :- enqueue([M, H], L), reverse(T, M).
reverse([], []).
enqueue([], X, [X]).
enqueue([A|B], Y, [A|Z]) :- enqueue([B, Y], Z).
```

**Bugs Found:**

*Predicate enqueue/2 is not structural recursive.*

**Variation #61**

```
reverse([], []).
reverse([H|T], L) :- add(L, H, M), reverse(T, M).
enqueue([], X, [X]).
enqueue([A|B], Y, [A|Z]) :- enqueue(B, Y, Z).
add(A, B, C) :- enqueue(C, B, A).
```

**Bugs Found:**

*Predicate add/3 is not structural recursive.*

**Variation #62**

```
reverse(L, R) :- L = [H|T], reverse(T, M), append(M, [H], R).
reverse([], []).
append([], X, X).
append([A|B], Y, [A|Z]) :- append(B, Y, Z).
```

**Transformed Normal Form:**

*on*

166

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

### Bugs Found:

*Constant for variable in 3rd argument in 1st subgoal of 2nd clause in append/3*

### Comments:

*Because the student coded the recursive subgoal of the 2<sup>nd</sup> clause of append/3 as append(B,Y,Y), the 2<sup>nd</sup> and 3<sup>rd</sup> arguments are unified together. Since the 2<sup>nd</sup> argument is a list consisting of a single element, it is considered a constant which explains why ADAPT thinks that the student has coded a constant instead of a variable in the 3<sup>rd</sup> argument of that subgoal.*

### Variation #63

```
reverse(L,R):-reverse(T,M),L=[H|T],append(M,H,R).
reverse([], []).
append([],X,X).
append([A|B],Y,[A|Z]) :-append(B,Y,Z).
```

### Transformed Normal Form:

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,A,C).
append([],A,[A]).
append([A|B],C,[A|D]) :- append(B,C,D).
```

### Bugs Found:

*Variable for constant in 3rd argument in head of 1st clause in append/3*

### Comments:

*The student's program is definitely incorrect. The question is whether the subgoal append(D,A,C) should have been append(D,[A],C) or the first clause of the append/3 definition append([],L,L) should have been append([],L,[L]) which is what ADAPT assumes.*

### Variation #64

on

```
reverse([],[]).
reverse([H|T],L):-reverse(T,L).
```

### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

### Bugs Found:

*Variable mismatch in 2nd argument of 1st subgoal in 2nd clause in reverse/2*  
*Missing subgoal in 2nd subgoal in 2nd clause in reverse/2*

### Variation #65

```
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
reverse([],[]).
append([],X,X).
append(W,Y,Z):-W=[A|B],append(B,Y,Z),Z=[A|Z].
```

### Bugs Found:

*"Reassignment" error in subgoal Z=[A|Z] in clause:*  
*append(W,Y,Z):-W=[A|B],append(B,Y,Z),Z=[A|Z].*

### Variation #66

```
reverse([],[]).
reverse([H,T],L):-reverse(T,M),append(M,[H],L).
append([],X,X).
append([H|T],X,[H|L]):-append(T,X,L).
```

### Transformed Normal Form:

```
reverse([],[]).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

### Bugs Found:

*Invalid list in 1st argument of head of 2nd clause in reverse/2*

### Variation #67

*on*

```
reverse([], []).
reverse([X],[X]).
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
append([],X,X).
append([H|T],X,[H|L]):-append(T,X,L).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],[A],[A]).
append([A|B],[C],[A|D]) :- append(B,[C],D).
```

**Bugs Found:**

*Redundant base case in 2nd clause in reverse/3*

**Variation #68**

```
reverse([], []).
reverse([X],[X]).
reverse([H|T],[M|H]):-reverse(T,M).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B],C) :- reverse(B,D),append(D,[A],C).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

**Bugs Found:**

*Redundant base case in 2nd subgoal in reverse/2*  
*Invalid append in 2nd argument of head of 3rd clause in reverse/2*

**Variation #69**

```
reverse([], []).
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
append([],[X],X).
append([H|T],[X],[H|L]):-append(T,[X],L).
```

**Transformed Normal Form:**

*on*

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

**Bugs Found:**

*Variable for constant in 3rd argument in head of 1st clause in append/3*

**Variation #70**

```
reverse([], []).
reverse([X], [X]).
reverse([H|T], L) :- reverse(T, M), enqueue(H, M, L).
enqueue(X, [], [X]).
enqueue(X, [H|T], L) :- enqueue(X, T, M), L=[H|M].
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(A, D, C).
append(A, [], [A]).
append(A, [B|C], [B|D]) :- append(A, C, D).
```

**Bugs Found:**

*Redundant base case in 2nd clause in reverse/2*

**Variation #71**

```
reverse([H|T], L) :- reverse(T, M), append(M, [H], L).
append([], X, X).
append([H|T], X, [H|L]) :- append(T, X, L).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

**Bugs Found:**

*Missing base case in reverse/2*

**Variation #72**

*on*

```
reverse([], []).
reverse([H|T], L) :- reverse(T, M), append(M, [H], L).
append([], X, X).
append([X], L, [X|L]).
append([H|T], X, [H|L]) :- append(T, X, L).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

**Bugs Found:**

*Redundant base case in 2nd clause in append/3*

**Variation #73**

```
reverse([X], [X]).
reverse([H|T], L) :- reverse(T, M), append(M, [H], L).
append([], X, X).
append([H|T], X, [H|L]) :- append(T, X, L).
```

**Transformed Normal Form:**

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
append([], [A], [A]).
append([A|B], [C], [A|D]) :- append(B, [C], D).
```

**Bugs Found:**

*Constant mismatch in 1st argument of head of 1st clause of reverse/2  
Constant mismatch in 2nd argument of head of 1st clause of reverse/2*

**Variation #74**

```
reverse([], []).
reverse(L, R) :- append([H], T, L), enqueue(H, M, R), reverse(T, M).
append([], X, X).
append([A|B], Y, [A|Z]) :- append(B, Y, Z).
enqueue(X, [], [X]).
enqueue(Y, [A|B], [A|Z]) :- enqueue(Y, B, Z).
```

**Bugs Found:**

*on*

*Predicate enqueue/3 is not structural recursive.*

## Variation #75

```
reverse([], []).
reverse(L, [H|T]) :- append(M, [H], L), reverse(T, M).
append([], X, X).
append(W, Y, [A|Z]) :- W=[A|B], append(B, Y, Z).
```

### Bugs Found:

*Predicate reverse/2 is not structural recursive*

## Variation #76

```
reverse([], []).
reverse(L, [H|RT]) :- append(T, H, L), reverse(T, RT).
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

### Transformed Normal Form:

```
reverse([], []).
reverse(A, [B|C]) :- append(D, B, A), reverse(D, B).
append([], A, [A]).
append([A|B], C, [A|D]) :- append(B, C, D).
```

### Bugs Found:

*Variable for constant in 3rd argument in head of 1st clause in append/3*

### Comments:

*ADAPT assumes the append/3 program is incorrect rather than assuming that the subgoal invocation append(T, H, L) is incorrect. Clearly the program is incorrect, but it is unclear what the student really intended. Intention Tracing (see Section 4.3) is needed here.*

## Variation #77

*on*

```
reverse(L,R):-reversex(L,R).
reverse([],X-X).
reverse([H|T],Y-X):-Z=[H|X],reverse(T,Y-Z).
reversex(A,B):-reverse(A,[],B).
reverse(List,Accum,Result):-reverse(List,Result-Accum).
```

**Bugs Found:**

*Predicate reverse/2 is mutually recursive*

**Comments:**

*Note that this program only contains 3 predicates: reverse/2, reversex/2, and reverse/3. The first three clauses define the same predicate, not two different ones as intended.*

**Variation #78**

```
reverse(L,R):-reverse(L,R-[]).
reverse([],X-X).
reverse([H|T],Y-X):-Z=[H|X],reverse(T,Y-Z).
```

**Bugs Found:**

*Infinite loop in your program since the accumulator predicate has the same name and arity as the main predicate.*

**Comments:**

*Note that this is a special case of a mutually recursive program where the accumulator predicate has the same name and arity as the initializing predicate.*

**Variation #79**

```
reverse(L,R):-reverse(f(R,[],L).
reverse(f(X,X),[]).
reverse(f(Y,X),[H|T]):-Z=[H|X],reverse(f(Y,Z),T).
```

**Bugs Found:**

*Infinite loop in your program since the accumulator predicate has the same name and arity as the main predicate.*

**Comments:**

*Note that this is a special case of a mutually recursive program where the accumulator predicate has the same name and arity as the initializing predicate.*

**Variation #80**

```
reverse(L,R):-reverse(L,R,[]).
reverse([],X,X).
reverse([H|T],Y,X):-X=[H|X],reverse(T,Y,X).
```

**Bugs Found:**

*"Reassignment" error in the subgoal X=[H|X] in clause:  
reverse([H|T],Y,X) :- B=[B|A],reverse(T,Y,X).*

**Variation #81**

```
reverse(A,B):-reversex(A,[B]).
reverse([], [B|B]).
reverse([Head|Tail],[B|Temp]):-reverse(Tail,[B,Head|Temp]).
```

**Bugs Found:**

*Missing definition for reversex/2, but there is a definition for reverse/2  
- perhaps the program contains a typographical error*

**Variation #82**

```
reverse(A,B):-reverse(A,[B]).
reverse([], [B|B]).
reverse([Head|Tail],[B|Temp]):-reverse(Tail,[B,Head|Temp]).
```

**Bugs Found:**

*Predicate reverse/2 is not structural recursive*

**Comments:**

*The student is asked to re-enter her program for reverse/2. The real problem here is that the*

*on*

*first clause is an "initialization" clause and the last two clauses are a railway-shunt version of an accumulator implementation. These two separate programs need to be given different names.*

### Variation #83 (Naive)

```
reverse(L,R):-L=[H|T],reverse(T,M),append(M,[H],R).
reverse([],[]).
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
```

### Variation #84

```
reverse(L,R):-reverse(T,M),L=[H|T],append(M,[H],R).
reverse([],[]).
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
```

### Variation #85

```
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
reverse([],[]).
append([],X,X).
append(W,Y,[A|Z]):-W=[A|B],append(B,Y,Z).
```

### Variation #86

```
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
reverse([],[]).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
append([],X,X).
```

### Variation #87

```
reverse([H|T],L):-reverse(T,M),enqueue(H,M,L).
reverse([],[]).
enqueue(C,D,E):-append(D,[C],E).
append([],X,X).
append([A|B],Y,[A|Z]):-append(B,Y,Z).
```

### Variation #88

```
append([],X,X).
```

on

```

append([A|B],Y,[A|Z]):-append(B,Y,Z).
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
reverse([],[]).

```

### Variation #89

```

reverse([],[]).
reverse(L,R):-L=[H|T],reverse(T,M),append(M,[H],R).
append([],X,X).
append(V,Y,W):-V=[A|B],append(B,Y,Z),W=[A|Z].

```

### Variation #90

```

reverse([],[]).
reverse(L,R):-L=[H|T],reverse(T,M),append(M,[H],R).
append([],X,X).
append(V,Y,W):-W=[A|Z],append(B,Y,Z),V=[A|B].

```

### Variation #91

```

reverse([],[]).
reverse([H|T],R):-reverse(T,M),append(M,[H],R).
append([],X,X).
append(V,Y,W):-W=[A|Z],V=[A|B],append(B,Y,Z).

```

### Variation #92

```

reverse([],X):-X=[].
reverse([H|T],L):-reverse(T,M),append(M,[H],L).
append([],L,L).
append([H|T],L,[H|R]):-append(T,L,R).

```

### Variation #93

```

reverse([H|T],L):-reverse(T,M),append(M,[H],L).
reverse([],[]).
append([],X,X).
append(W,Y,[A|Z]):-W=[A|B],append(B,Y,Z).

```

### Variation #94

```

reverse([],L,L).
reverse([H|T],X,[H|L]) :- reverse(T,X,L).
append([],[]).
append([H|T],L) :- append(T,M),reverse(M,[H],L).

```

**Comments:**

*Note that ADAPT is not fooled by the name reversal between append/3 and reverse/2 since it looks for the main predicate in the program (and ignores the names the student has assigned to the predicates).*

**Variation #95**

```

reverse([],[]).
reverse(L,R):-enq1(L,[H|T]),reverse(T,M),enq2([M,H],R).
enq1(X,X).
enq2([A|B],C):-append(A,B,C).
append([],X,X).
append([H|T],X,[H|L]):-append(T,X,L).

```

**Variation #96**

```

reverse([],[]).
reverse(L,R):-append([H],T,L),reverse(T,M),append(M,[H],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).

```

**Variation #97**

```

reverse([],[]).
reverse([A|B],R):-append([H],T,[A|B]),reverse(T,M),append(M,[H],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).

```

**Variation #98**

```

reverse([],[]).
reverse([A|B],[C|D]):-append([H],T,[A|B]),reverse(T,M),append(M,[H],[C|D]).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).

```

**Variation #99 (Inverse Naive)**

on

```
reverse([], []).
reverse(L, [H|RT]) :- append(T, [H], L), reverse(T, RT).
append([], A, A).
append([A|B], C, [A|D]) :- append(B, C, D).
```

### Variation #100

```
reverse([], []).
reverse(L, R) :- append(T, [H], L), reverse(T, M), R = [H|M].
append([], X, X).
append([A|B], Y, [A|Z]) :- append(B, Y, Z).
```

### Variation #101

```
reverse([], []).
reverse(L, R) :- enqueue(T, H, L), reverse(T, M), R = [H|M].
enqueue([], X, [X]).
enqueue([A|B], Y, [A|Z]) :- enqueue(B, Y, Z).
```

### Variation #102

```
reverse([], []).
reverse(L, [H|T]) :- last(L, H, M), reverse(M, T).
last([X], X, []).
last([H|T], X, [H|L]) :- last(T, X, L).
```

### Variation #103

```
reverse([], []).
reverse(L, R) :- append(T, [H], L), reverse(T, M), append([H], M, R).
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

### Variation #104

```
reverse([], []).
reverse(L, [H|RT]) :- enqueue(T, H, L), reverse(T, RT).
enqueue([], X, [X]).
enqueue([H|T], X, [H|L]) :- enqueue(T, X, L).
```

### Variation #105

on

```
reverse([], []).
reverse(L, [H|M]) :- append(T, [H], L), reverse(T, M).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

### Variation #106

```
reverse([], []).
reverse(L, [C|D]) :- append(T, [H], L), reverse(T, M), append([H], M, [C|D]).
append([], L, L).
append([H|T], X, [H|L]) :- append(T, X, L).
```

### Variation #107 (Accumulator)

```
reverse(L, R) :- reverse(L, [], R).
reverse([], X, X).
reverse([H|T], X, Y) :- Z=[H|X], reverse(T, Z, Y).
```

### Variation #108

```
reverse(L, R) :- reverse(L, [], R).
reverse([], X, X).
reverse([H|T], X, Y) :- Z=[H|X], reverse(T, Z, Y), Z=[H|X].
```

### Variation #109

```
reverse(L, R) :- reverse(L, [], R).
reverse([], X, X).
reverse([H|T], X, Y) :- Z=[H|Y], reverse(T, X, Z).
reverse(A, B, C) :- reverse(A, C, B).
```

### Variation #110

```
reverse(L, R) :- reverse(L, [R]).
reverse([], [X|X]).
reverse([H|T], [Y|X]) :- Z=[H|X], reverse(T, [Y|Z]).
```

### Variation #111

```
reverse(L, R) :- reverse(L, [R]).
reverse([], [X|X]).
reverse([H|T], [Y|X]) :- Z=[Y, H|X], reverse(T, Z).
```

**Variation #112**

```
reverse(L,R):-reversex(L,[R]).
reversex([], [X|Y]):-X=Y.
reversex([H|T],[Y|X]):-reversex(T,[Y,H|X]).
```

**Variation #113 (Difference List)**

```
reverse(L,R):-reversex(L,R-[]).
reversex([],X-X).
reversex([H|T],X-Y):-reversex(T,X-[H|Y]).
```

**Variation #114**

```
reverse(L,R):-reversex(L,f(R,[])).
reversex([],f(X,X)).
reversex([H|T],f(X,Y)):-reversex(T,f(X,[H|Y])).
```

**Variation #115**

```
reverse(L,R) :- rs_reverse(L,[R]).
rs_reverse([], [X|X]).
rs_reverse([H|T],S) :- rs_reverse(T,Y),insert(S,H,Y).
insert([A|B],H,Y) :- enqueue([A],H,X),append(X,B,Y).
enqueue([],E,[E]).
enqueue([H|T],E,[H|R]) :- enqueue(T,E,R).
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

**Variation #116**

```
reverse(L,R):-reverse(L,R,[]).
reverse([],X,Y):-X=Y.
reverse([H|T],R,X):-reverse(T,R,[H|X]).
```

**Variation #117**

```
reverse(A,B):-append(A,[B]).
append([], [B|B]).
append([Head|Tail],[B|Temp]):-append(Tail,[B,Head|Temp]).
```

**Comments:**

*Note that ADAPT is not fooled by the inappropriate name append/2 assigned to the railway-shunt accumulator portion of the program.*

### Variation #118

```
reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([H|T],X,R):-append([H],X,Y),reverse(T,Y,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

### Variation #119

```
reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse(L,X,R):-append([H],T,L),reverse(T,[H|X],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

### Variation #120

```
reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([A|B],X,R):-append([H],T,[A|B]),reverse(T,[H|X],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

### Variation #121

```
reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse(L,X,R):-append([H],T,L),append([H],X,Y),reverse(T,Y,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

### Variation #122

```
reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([A|B],X,R):-append([H],T,[A|B]),append([H],X,Y),reverse(T,Y,R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).
```

**Variation #123**

```

reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse(L,X,R):-append([H],T,L),append([H],X,[C|D]),reverse(T,[C|D],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).

```

**Variation #124**

```

reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([H|T],X,R):-append([H],X,[C|D]),reverse(T,[C|D],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).

```

**Variation #125**

```

reverse(L,R):-reverse(L,[],R).
reverse([],X,X).
reverse([A|B],X,R):-append([H],T,[A|B]),append([H],M,[C|D]),reverse(T,[C|D],R).
append([],L,L).
append([H|T],X,[H|L]):-append(T,X,L).

```