Automatic Program Debugging for Intelligent Tutoring Systems

William R. Murray

AI TR86-27       June 1986

Support for this research was provided by the Army Research Office, under grant number #ARO DAAG29-84-K-0060 and by NSF CER grant #MCS-8122039.
ACKNOWLEDGEMENTS

I would like to thank Bruce Porter, Elaine Rich, and Mark L. Miller for the invaluable help they have provided in this research. Their time and contributions are deeply appreciated. I would also like to thank Woody Bledsoe and Chris Lengauer for their assistance. Finally, I am grateful to J Moore and Bob Boyer for help understanding their logic and theorem prover. Key ideas in this dissertation can be traced to their work in automatic theorem proving.

The task of formatting this dissertation was greatly eased by use of automated document processing. The Scribe document formatter was conceived of and created by Brian Reid. The current version has been maintained and enhanced by Unilogic, Ltd. The Scribe format definitions for proper dissertation format for The University of Texas at Austin were developed by Richard Cohen.

This work was sponsored in part by NSF CER grant MCS-8122039 and Army Research Office grant DAAG29-84-K-0060.

William R. Murray
The University of Texas at Austin
May, 1986
ABSTRACT

Program debugging is an important part of the domain expertise required for intelligent tutoring systems that teach programming languages. This thesis explores the process by which student programs can be automatically debugged in order to increase the instructional capabilities of these systems. This research presents a methodology and implementation for the diagnosis and correction of nontrivial recursive programs. In this approach, recursive programs are debugged by repairing induction proofs in the Boyer-Moore Logic.

The potential of a program debugger to automatically debug widely varying novice programs in a nontrivial domain is proportional to its capabilities to reason about computational semantics. By increasing these reasoning capabilities a more powerful and robust system can result. This thesis supports these claims by examining related work in automated program debugging and by discussing the design, implementation, and evaluation of Talus, an automatic debugger for LISP programs.

Talus relies on its abilities to reason about computational semantics to perform algorithm recognition, infer code teleology and to automatically detect and correct nonsyntactic errors in student programs written in a restricted, but nontrivial, subset of LISP. Solutions can vary significantly in algorithm, functional decomposition, role of variables, data flow, control flow, values returned by functions, LISP primitives used, and identifiers used. Solutions can consist of multiple functions, each containing multiple bugs. Empirical evaluation demonstrates that Talus achieves high performance in debugging widely varying student solutions to challenging tasks.
# Table of Contents

1. Introduction  
   1.1 Motivation: Domain Experts for Intelligent Tutors  
      1.1.1 The Components of a Complete ITS  
      1.1.2 Inferring Bugs versus Inferring Misconceptions  
   1.2 Why Debugging is Hard  
   1.3 Limitations of Automated Program Debuggers  
   1.4 Debugging Recursive Programs with Induction Theorem Proving  
      1.4.1 The Boyer-Moore Logic and Theorem Prover  
      1.4.2 Debugging Recursive Programs by Repairing Induction Proofs  
      1.4.3 A Simple Case Study: FLATTEN  
   1.5 Talus Debugging Examples  
      1.5.1 Missing Conditional Tests and Infinite Loops in MEMBER  
      1.5.2 Algorithm Recognition, Multiple -- Functions and Formal Variables in MEMTREE  
      1.5.3 Special Forms and Side Effects in PALINDROMES  
   1.6 Summary  
   1.7 Dissertation Overview  

2. Related Work in Automated Program Debugging  
   2.1 A Model of Automated Program Debugging  
   2.2 A Survey of Approaches  
   2.3 Exemplars of Alternate Debugging Approaches  
      2.3.1 Dynamic Analysis: PDS6  
      2.3.2 Heuristic Plan Recognition: PROUST  
      2.3.3 Analysis by Synthesis: GREATERP  
      2.3.4 Analysis of Loop Invariants  
   2.4 Comparison of Approaches  
   2.5 Theoretical Limitations  
   2.6 Summary  

3. An Overview of Talus  
   3.1 The Domain  
      3.1.1 Acceptable Solutions  
      3.1.2 Tasks and Task Representations  
   3.2 The Debugging Capabilities of Talus  
   3.3 An Overview of the Debugging Process  
      3.3.1 A Simple Example  
      3.3.2 Program Simplification  
      3.3.3 Algorithm Recognition  
      3.3.4 Bug Detection  
      3.3.5 Bug Correction  
   3.4 Case Studies  
      3.4.1 MEMTREE  
      3.4.1.1 Algorithm Recognition  
      3.4.1.2 Bug Detection  
      3.4.1.3 Bug Correction  
      3.4.2 SINGLETONS Case Study  
      3.4.3 MEMBER Case Study  
      3.4.4 PALINDROMES Case Study  
      3.4.5 REVERSE Case Study  
   3.5 Computational Semantics in Debugging  
   3.6 Summary  

4. E-frames: A Frame Representation for Functions  
   4.1 Recursively Defined Data Structures and Recursive Functions
II. An Annotated PDS6 Scenario
   237
III. The Task Library
     243
IV. Algorithm Representations for MEMTREE
    247
V. Handout for CS 381K
   251
Glossary
   257
References
   267
Index
   271
List of Figures

Figure 1-1: Relationship of Talus to a Complete ITS 2
Figure 1-2: Talus System Development 14
Figure 2-1: A Model of Automated Program Debugging 21
Figure 2-2: A Taxonomy of Automated Debugging Approaches 24
Figure 2-3: A Taxonomy of Debuggers that use Dynamic Analysis 24
Figure 2-4: A Taxonomy of Debuggers that use Static Analysis 25
Figure 2-5: PROUST Bug Report Suggesting a Misconception 30
Figure 2-6: Best Results of PROUST on RAINFALL Task 31
Figure 2-7: Best Results of PROUST on BANK Task 31
Figure 3-1: The LISP Dialects of Talus 40
Figure 3-2: The Pure Dialect 41
Figure 3-3: An Overview of Debugging in Talus 43
Figure 3-4: Binary Tree Representation of IN 48
Figure 3-5: Algorithm Recognition in Talus 49
Figure 3-6: Bug Detection in Talus 50
Figure 3-7: A Buggy Solution to the MEMTREE Task 53
Figure 3-8: Symbolic Evaluation of MEM 61
Figure 3-9: Symbolic Evaluation of MEM 61
Figure 3-10: COND-TO-IF Simplification of FACT 65
Figure 3-11: Edit Inversion through a COND-TO-IF Transform 66
Figure 5-1: Best First Search during Algorithm Recognition in MEMTREE Case Study 98
Figure 5-2: Search Paths Eliminated by Constraints 102
Figure 6-1: Binary Tree Representation of MEMBER 121
Figure 6-2: Symbolic Evaluation of MEMBER, First Step 121
Figure 6-3: Symbolic Evaluation of MEMBER, Final Result 122
Figure 6-4: Path Traversal with a True Predicate 122
Figure 6-5: Path Traversal with a False Predicate 123
Figure 6-6: A Predicate that Always Causes an Error 123
Figure 6-7: Case Splitting 124
Figure 6-8: Binary Tree Representations for two definitions of IN 126
Figure 6-9: Symbolically Evaluating IN, Case 1 128
Figure 6-10: Symbolically Evaluating IN, Case 2 129
Figure 6-11: Symbolically Evaluating IN, Case 3 130
Figure 6-12: Binary Tree Representations for Two Definitions of MEM 131
Figure 6-13: An Error During Symbolic Evaluation 131
Figure 6-14: The Predicate that Checks Termination Verification Conditions 136
Figure 6-15: The Bug Detection Algorithm of Talus 137
Figure 6-16: Filtering Invalid Conjectures with Counterexamples 139
Figure 7-1: The Bug Isolation Algorithm isolate 156
Figure 7-2: The Improved Bug Isolation Algorithm isolate* 159
Figure 7-3: Graph Representation of REV 163
Figure 7-4: Graph Representation of a Bug Fix to REV 164
Figure 7-5: Display of MEMTREE Debugging 165
Figure 7-6: Graph Representation of an Inserted Conditional Expression 166
Figure 7-7: The Wishing Well Task 169
Figure 7-8: A Buggy Solution to the Wishing Well Task 170
Figure 8-1: Program Simplification, Debugging, and Edit Inversion 173
Figure 8-2: The Core Dialect 175
Figure 8-3: The Extended Dialect 176
Figure 8-4: Graph Representation of a COND-TO-IF Transformation 177
Figure 8-5: Edit Inversion Through a COND-TO-IF Transform 178
Figure 10-1: Alternate Approaches to Automated Program Debugging 228
List of Tables

Table 5-1: Effect of Plausibility Constraints on Search in SINGLETONS Case Study 102
Table 9-1: Summary of Data Analyzed 207
Table 9-2: Summary of Algorithm Recognition 207
Table 9-3: Summary of Data Analyzed 208
Table 9-4: Summary of Bug Correction 208
Table 9-5: Algorithms in Initial SINGLETONS Data 209
Table 9-6: Summary of First Data Set 210
Table 9-7: Summary of Second Data Set 211
Table 9-8: Summary of Third Data Set, LENGTH Task 211
Table 9-9: Summary of Third Data Set, PROG-LENGTH Task 212
Table 9-10: Summary of Third Data Set, FLATTEN Task 212
Table 9-11: Summary of Third Data Set, COPY-BUT-LAST Task 212
Table 9-12: Summary of Third Data Set, SINGLETONS Task 212
Table 9-13: Summary of Third Data Set, SINGLETONS Task, After Reanalysis 213
Table 9-14: Variability in First Data Set 216
Table 9-15: Performance in Algorithm Identification 217
Table 9-16: Performance in Algorithm Recognition, SINGLETONS Task, Third Data Set 217
Table 9-17: Performance in Mapping Functions 217
Table 9-18: Performance in Mapping Formal Variables 218
Table 9-19: Performance in Detecting Buggy Algorithms in Reanalyzed Data 223
Table 9-20: Performance in Detecting Extra Conditional Tests 224
Table 9-21: Performance in Detecting Missing Conditional Tests 224
Table 9-22: Performance in Detecting Bugs in Terminations and Recursions 225
Table 9-23: Performance in Bug Correction 226
Chapter One

Introduction

An intelligent tutoring system (ITS) is a computer based system intended to provide effective, appropriate, and flexible instruction to students [Sleeman 82]. These aims require domain expertise in the subject matter being taught. This dissertation focuses on one domain - computer programming - and on one aspect of the necessary domain expertise - program debugging.

This research explores the relationship between automated program debuggers, capable of providing domain expertise for an ITS to teach a programming language, and computational semantics. Computational semantics is the ability to reason about programs, program execution, programming language semantics, and the data structures that are created and processed by programs.

Talus\(^1\) is a program debugger that relies on its abilities to reason about computational semantics to automatically detect and correct nonsyntactic errors in student programs written in a restricted, but nontrivial, subset of LISP. Solutions can vary significantly in algorithm, functional decomposition, role of variables, data flow, control flow, values returned by functions, LISP primitives used, and identifiers used. Solutions can consist of multiple functions, each containing multiple bugs. Empirical evaluation demonstrates that Talus achieves high performance in debugging widely varying student solutions to challenging tasks.

This dissertation is intended to support the following two-part thesis relating automated program debuggers and the ability to reason about computational semantics:

1. *All* automated program debuggers are limited by restrictions in their capabilities to reason about computational semantics;

2. Talus demonstrates an effective debugging methodology for nontrivial recursive programs that is based on an ability to reason about computational semantics.

An inference mechanism capable of reasoning about the computational semantics of recursive programs must be able to reason about inductively constructed data objects, such as lists, trees, and queues. The work of Boyer and Moore [Boyer 79] in induction theorem proving provides a formalism for this reasoning and its mechanization. The program debugging methodology of Talus builds on their work.

This chapter provides an introduction to this research. Section 1.1.1 further examines the relationship between automated debuggers and complete intelligent tutoring systems. Section 1.1.2 distinguishes between bugs and misconceptions and describes their relationship. Section 1.2 explains why automated debugging is difficult. The limitations of automated program debuggers due to restricted reasoning capabilities are surveyed in Section 1.3. Section 1.4 demonstrates how Talus can overcome many of these difficulties by directly reasoning about computational semantics using the Boyer-Moore Logic and theorem prover. Examples in Section 1.5 illustrate the more advanced debugging capabilities of Talus such as algorithm recognition. Section 1.7 provides an overview of this dissertation and a guide to the reader.

\(^1\)Pronounced tay' ies [Samuel 71]. Talus is the last of the bronze race in classical mythology (see Appendix I).
1.1 Motivation: Domain Experts for Intelligent Tutors

1.1.1 The Components of a Complete ITS

Talus is intended to be an important part of the domain expert of an ITS to teach LISP. A complete intelligent tutoring system would include a student model, a dialog manager, a tutorial expert, courseware, and additional domain expertise such as the program synthesis capabilities present in GREATERP [Reiser 85]. The student model represents the student's current state of knowledge, learning capabilities, background, etc. The dialog manager provides for a natural language (or other friendly) interface to the student. The tutorial expert represents tutorial knowledge (e.g. the T-rules of GUIDON [Clancey 79]) and the ability to use this knowledge, along with the student model, to make pedagogical decisions. The corpus of subject matter to be taught is represented in the courseware module (e.g. the curriculum information network of BIP [Barr 76]).

The relationship of Talus to a complete ITS is shown in Figure 1-1. Talus receives student code as input from the dialog manager or user interface and debugs the program. The resulting debugged code and records of bug fixes are used by the student model to infer possible misconceptions the student may have. This information can then be used by the tutorial expert to provide remedial instruction to resolve the misconceptions. The tutorial expert may directly display the bug corrections, generate a hint, provide a counterexample, or monitor and guide the debugging process in some other manner.

![Diagram of Talus to a Complete ITS](image)

Figure 1-1: Relationship of Talus to a Complete ITS

A domain expert such as Talus is an important part of an ITS to teach a programming language. Reiser, Anderson, and Farrell [Reiser 85] have found that "...most of the learning in acquiring a cognitive skill occurs while the student actually tries to solve problems in the domain." Furthermore, without assistance in debugging, students can confuse the skills of debugging and programming.

This dissertation addresses the diagnosis and correction of student errors in the domain of recursive programming. It emphasizes debugging methodology for intelligent tutoring systems, rather than student
modelling, tutorial strategies, or pedagogical issues (such as intelligent hint generation) in the use of debugged student programs.

1.1.2 Inferring Bugs versus Inferring Misconceptions

Joni, Soloway, Goldman, and Ehrlich [Joni 83] distinguish bugs from misconceptions: "...a bug is an error in a computer program, while a misconception is some conceptualization in the student's mind that can lead to a program bug." They classify bugs into two categories: bugs that prevent the student's program from solving the task, and bugs that do not interfere with the task solution but which suggest student misconceptions. These two categories will be referred to as nonstilistic bugs and stylistic bugs in this dissertation. In LISP, missing conditional tests are nonstylistic bugs while unnecessary conditional tests are stylistic bugs.

The following misconceptions are identified in [Joni 83]:

- **Incorrect Analogies**: The student expects knowledge from another domain, such as English or another programming language, to apply in the language being taught.
- **Overgeneralized Concepts**: A concept is applied in an inappropriate context.
- **Ignorance of "Rules of Programming Discourse"**: Novices do not yet know stylistic rules that simplify code readability and maintainability.
- **Faulty or Incomplete Understanding of a Concept**: A concept is partially or incorrectly learned. Such misconceptions may not surface in simple programs since simple programs do not fully test the student's understanding of the concept, or the student's understanding of how this concept interacts with others.
- **Inability to Correctly Use Multiple Constructs Together**: Individual constructs may be understood in isolation but the proper means of coordinating their use may not be understood.
- **Faulty Program Decompositions**: The task may not be correctly decomposed.

Note that some of these misconceptions, such as the last three, will not manifest themselves in simple tasks. Thus tutors such as GREATERP [Reiser 85] that decompose the programming task into very small units may prevent the expression of these misconceptions, and will be unable to diagnose them. Talus allows larger tasks to be assigned where such misconceptions can surface.

Talus diagnoses both nonstilistic and stylistic bugs, but does not infer misconceptions. However, the debugging performed by Talus is sufficiently accurate and reliable that misconceptions can be inferred from its output. A premise of this research is that complete knowledge of the location of all bugs and the corrections necessary to fix the bugs will greatly simplify the inference of misconceptions. This inference would be performed by the student modeling component of a complete ITS, not by Talus. Important work that relates bugs to misconceptions has been done by Soloway [Soloway 83], Johnson [Johnson 85], Anderson [Reiser 85], and Miller and Goldstein ( [Miller 77] and [Miller 78]).

1.2 Why Debugging is Hard

The problem of detecting and correcting errors in programs is difficult because of significant variability. If the task is to sort an array many alternate algorithms can be used: QUICKSORT, HEAPSORT, BUBBLESORT, etc. Each algorithm can be decomposed into procedures in many ways. For each procedure there can be an infinite number of correct implementations varying in control flow, programming constructs used, identifiers used, formal variable order in parameter lists, and role of
variables. For each implementation there can be an infinite number of bugs, such as misspelled identifiers, incorrect variable updates, missing recursive calls, unintended side effects, divide by zero bugs, etc.

Unanticipated implementations seem to pose an insurmountable problem in diagnosing bugs in programs in nontrivial domains. For trivial domains all or most of the correct and incorrect solutions to tasks can be stored or easily generated and little search is required to identify the student's solution. In nontrivial domains it is impossible to store even all of the correct solutions. It is difficult to generate the solutions since a program synthesizer must be constructed and the student's design choices must be either provided or inferred. The problem for an automatic program debugger then is to accept unanticipated correct implementations while distinguishing them from incorrect implementations.

1.3 Limitations of Automated Program Debuggers

Since the major difficulty in debugging programs is their variability, all program debuggers limit variability to some extent. This dissertation shows that the severity of these limitations is dependent on the degree of ability to reason about computational semantics. Systems with greater abilities to reason about computational semantics, such as Talus, support greater variability in the programs they can analyze.

Although an ability to reason about computational semantics is important, it is not sufficient for the debugging of programs that vary significantly. The research of Johnson (Johnson 85) in the program debugger PROUST has shown that it is also necessary to infer the goals that student code is intended to achieve. Only by knowing the purpose of a program or program fragment can we decide if that purpose is achieved and if this contributes to the overall solution of the task. Johnson refers to this as intention based program diagnosis. The intended role of code in solving a task has also been referred to as its teleology [Goldstein 74].

Thus some limitations in debugging capabilities can be attributed to inability to infer code teleology. However this does not contradict the utility of reasoning about computational semantics. The role of computational semantics is to make possible a comparison between a code fragment and its intended function. It will be shown that Talus uses its ability to reason about computational semantics to increase its ability to reason about code teleology. Talus demonstrates that these two capabilities can sometimes interact in a synergistic fashion.

Program debuggers compensate for an inability to reason about computational semantics and code teleology in one of three ways:

- By providing limited debugging capabilities. Limited debugging capabilities include vague, missed, or incorrect bug diagnoses and false alarms. False alarms occur when correct code is interpreted as buggy. Other common limitations are inabilities to:
  - Recognize algorithms
  - Detect stylistic bugs
  - Correct bugs
  - Debug recursive programs
  - Debug programs with side effects
  - Allow multiple procedures
• Allow unanticipated implementations

• *By imposing severe constraints on programs suitable for analysis.* These constraints include highly limited programming languages or very small task size.

• *Depending on user or teacher guidance.* For example, an oracle can be assumed to provide information that cannot be inferred.

### 1.4 Debugging Recursive Programs with Induction Theorem Proving

This section demonstrates how an ability to reason about computational semantics provides a basis for a program debugging methodology. The methodology illustrated here allows for unanticipated implementations, bug detection, and bug correction. It is the core of Talus, but only a small part of the complete system.

The debugging methodology that Talus uses differs from most related work in program debugging. Talus attempts to construct an inductive proof that student programs are equivalent to correct reference programs. When the proof would ordinarily fail, Talus alters the student's program using the reference program as a source of corrections. Inductive hypotheses establishing the functional equivalence of the two programs allow references to the reference program to be eliminated from bug fixes.

Before examining the debugging methodology, we take a brief look at the Boyer-Moore Logic and Boyer-Moore Theorem Prover [Boyer 79]. The logic will allow us to make precise assertions about computational semantics and the theorem prover can be our inference mechanism.

#### 1.4.1 The Boyer-Moore Logic and Theorem Prover

The Boyer-Moore Logic is a precise formalism for expressing properties of inductively constructed data objects and recursive programs. For example, the conjecture:

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

expresses the commutativity of addition. An inductive proof is necessary to establish that the conjecture is a theorem.

The logic subsumes propositional calculus and allows the use of total functions in terms and as predicates. Free variables in conjectures are universally quantified. Functions are admitted under the Principle of Definition that ensures that they are total functions, i.e. they have no infinite loops. New data types are constructed under the Shell Principle that axiomatizes their properties and the properties of recognizers, constructors, and accessors for the new data type. The basic data types present are CONSes (ordered pairs), natural numbers (0, 1, 2 ...), strings, and boolean constants (T and F). New inductively defined data structures, such as stacks, can be introduced under the Shell Principle.

The Boyer-Moore Theorem Prover is a mechanical theorem prover capable of proving inductive proofs automatically. It has heuristics that choose substitutions for the inductive proof and the best variable to perform the induction over. These heuristics exploit the close relationship between recursion and induction. Other heuristics allow simplification of terms, the unfolding of function definitions, and the generalization of conjectures.
1.4.2 Debugging Recursive Programs by Repairing Induction Proofs

The Principle of Induction in the Boyer-Moore Logic is a formal representation of correct inductive proofs. Talus follows this principle of induction in generating verification conditions that establish the equivalence of a student program with a reference program. These verification conditions can be thought of as lemmas used in the main inductive proof. Bugs are indicated by failed verification conditions. Bugs are repaired by modifying the student's program such that the new verification conditions that result are theorems. The code to repair bugs in the student's program comes from the reference program. Thus the reference program serves both as a specification and a source of corrections for the student's program.

The debugged program is a byproduct of this enforced proof. Alternatively, the proof that the debugged program does in fact solve the task the reference function solves can be viewed as a byproduct of the debugging process. The first view is presented here since the focus is on the derivation of Talus from induction theorem proving. In subsequent chapters, where the concentration is on debugging, the second view will be adopted.

This discussion of the debugging process has assumed a decision procedure for conjecture evaluation, which is impossible due to the undecidability of predicate calculus. Empirical evaluation establishes that errors introduced by this assumption are not serious. If we assume a conjecture evaluator that is sound but not complete, such as the Boyer-Moore Theorem Prover, then this debugging process will never miss any bugs, but may replace some student code fragments unnecessarily (i.e. be subject to false alarms). Alternatively if we assume a conjecture evaluator that is complete but not sound then no false alarms will ever occur, but some bugs may be missed. A counterexample generator can provide such a conjecture evaluator; its use is discussed in Chapter 6.

The undecidability of program equivalence might seem to invalidate this debugging approach. Again, empirical results argue otherwise. In a similar vein, one could decide that since the Boyer-Moore Logic is undecidable the mechanization of a theorem prover in that logic is doomed to failure. However, the Boyer-Moore Theorem Prover has been used successfully to prove difficult theorems, such as the Unique Prime Factorization Theorem from Peano's axioms [Boyer 79].

1.4.3 A Simple Case Study: FLATTEN

We will examine an inductive proof in the Boyer-Moore Logic that establishes that a student function is equivalent to a reference function. Examining the case analysis and conjectures that arise in the proof will show the origin of the verification conditions that Talus examines. Then we introduce a bug into the student's function, see where the proof fails, and how Talus would intervene to repair the program bug and allow the proof to succeed.

This example starts in the middle of a tutorial session, after the student has had some instruction in LISP programming. Assume the student is assigned the following task:

Write a function that returns all the atoms in an arbitrary s-expression in a proper list. For example,

\[
\text{(FLAT } '(A (1 G) J K)) \text{ is (A 1 G NIL J K NIL)}
\]

---

2 Pedagogically, it is more desirable to miss some bug diagnoses than to provide inappropriate instruction based on false alarms.
and provides the following correct solution:

\[
\text{(DEFUN FLAT1 (TREE)} \\
\text{ (IF (LISTP TREE)} \\
\text{ \hspace{1cm} (IF (ATOM (CAR TREE))} \\
\text{ \hspace{2cm} (CONS (CAR TREE) (FLAT1 (CDR TREE))}) \\
\text{ \hspace{2cm} (APPEND (FLAT1 (CAR TREE))} \\
\text{ \hspace{3cm} (FLAT1 (CDR TREE))))}) \\
\text{ \hspace{1cm} (CONS TREE NIL))})
\]

Let \text{REF} be our reference function:

\[
\text{(DEFUN REF (TREE)} \\
\text{ \hspace{1cm} (IF (ATOM TREE)} \\
\text{ \hspace{2cm} (LIST TREE)} \\
\text{ \hspace{3cm} (APPEND (REF (CAR TREE))} \\
\text{ \hspace{4cm} (REF (CDR TREE))}))
\]

and assume that all these function definitions have been accepted by the theorem prover, along with any other nonprimitive functions that are referred to (e.g. \text{APPEND}) in those definitions.

Note that the student and stored function differ in control flow, recursive calls, and implementation of base cases. \text{FLAT1} tests the conditions (\text{LISTP TREE}) and (\text{ATOM (CAR TREE)}) while \text{REF} tests only for the condition (\text{ATOM TREE}). \text{FLAT1} returns (\text{CONS TREE NIL}) when (\text{LISTP TREE}) is false while \text{REF} returns (\text{LIST TREE}) when (\text{ATOM TREE}) is true. \text{FLAT1} has a recursive call in (\text{CONS (CAR TREE) (FLAT1 (CDR TREE))}) while no similar call appears in \text{REF}.

In the transcript below, explanatory comments are italicized.

\[
\text{(PROVE '(EQUAL (REF TREE) (FLAT1 TREE)))}
\]

\text{This command instructs the theorem prover to attempt to prove that}

\text{REF and FLAT1 are computationally equivalent.}

Give the conjecture the name \text{*1}.

We will try to prove it by induction. There are two plausible inductions. However, they merge into one likely candidate induction. We will induct according to the following scheme:

\[
(\text{AND (IMPLIES (ATOM TREE) (P TREE))} \hspace{1cm} ;P \text{ refers to *1} \\
\hspace{1cm} \text{(IMPLIES \hspace{1cm} (AND (NOT (ATOM TREE))} \hspace{1cm} \\
\hspace{2cm} \text{(P (CAR TREE))} \hspace{1cm} \\
\hspace{3cm} \text{(P (CDR TREE))}) \hspace{1cm} \\
\hspace{1cm} \text{(P TREE))})
\]

Linear arithmetic, the lemmas \text{CDR-LESSEQP}, \text{CDR-LESP}, \text{CAR-LESSEQP}, and \text{CAR-LESP}, and the definition of \text{ATOM} inform us that the measure (\text{COUNT TREE}) decreases according to the well-founded relation \text{LESP} in each induction step of the scheme. The above induction scheme produces two new goals:
The theorem prover splits the proof into two cases, the base case and the induction step. The base case is considered first:

Case 2. (IMPLIES (ATOM TREE)
             (EQUAL (REF TREE) (FLAT1 TREE))),

which simplifies, opening up the functions ATOM, REF, and FLAT1, to:

T.

Here is the induction step:

Case 1. (IMPLIES (AND (NOT (ATOM TREE))
                  (EQUAL (REF (CAR TREE))
                        (FLAT1 (CAR TREE)))
                  (EQUAL (REF (CDR TREE))
                        (FLAT1 (CDR TREE))))
                  (EQUAL (REF TREE) (FLAT1 TREE))),

which simplifies, unfolding ATOM, REF, and FLAT1, to the formula:

(IMPLIES (AND (LISTP TREE)
               (EQUAL (REF (CAR TREE))
                      (FLAT1 (CAR TREE)))
               (EQUAL (REF (CDR TREE))
                      (FLAT1 (CDR TREE)))
               (NOT (LISTP (CAR TREE))))
               (EQUAL (APPEND (REF (CAR TREE))
                             (REF (CDR TREE)))
                             (CONS (CAR TREE)
                             (REF (CDR TREE)))))

which we further simplify, rewriting with the lemmas CDR-CONS and CAR-CONS, and opening up the functions ATOM, REF, FLAT1, APPEND, and LISTP, to:

T.

That finishes the proof of *1. Q.E.D.

The theorem was proved. The key point is that by reasoning about computational semantics it was established that FLAT1 is equivalent to REF even though the two have different control flow, recursive calls, and base cases.

Suppose that instead of FLAT1, the student had provided this code:
(DEFUN FLAT2 (TREE)
  (IF (LISTP TREE)
      (IF (ATOM (CAR TREE))
          (CONS (CAR TREE) (FLAT2 (CDR TREE)))
          (CONS (FLAT2 (CAR TREE));Bug
                 (FLAT2 (CDR TREE))))
      (CONS TREE NIL)))

FLAT2 differs from FLAT1 since CONS has replaced APPEND and introduced a bug into the definition.

Now watch what happens when we compare the buggy function FLAT2 to the reference function REF:

(PROVE '(EQUAL (REF TREE) (FLAT2 TREE)))

Give the conjecture the name *1.

We will try to prove it by induction. There are two plausible inductions. However, they merge into one likely candidate induction. We will induct according to the following scheme:

(AND (IMPLIES (ATOM TREE) (P TREE))
     (IMPLIES (AND (NOT (ATOM TREE))
                  (P (CAR TREE))
                  (P (CDR TREE)))
     (P TREE)))

Linear arithmetic, the lemmas CDR-LESSEQP, CDR-LESP, CAR-LESSEQP, and CAR-LESP, and the definition of ATOM inform us that the measure (COUNT TREE) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme produces two new goals:

The base case:

Case 2. (IMPLIES (ATOM TREE)
              (EQUAL (REF TREE) (FLAT2 TREE))),

which simplifies, opening up the functions ATOM, REF, and FLAT2, to:

T.

The induction step:
Case 1. (IMPLIES (AND (NOT (ATOM TREE)))
  (EQUAL (REF (CAR TREE))
    (FLAT2 (CAR TREE)))
  (EQUAL (REF (CDR TREE))
    (FLAT2 (CDR TREE))))
  (EQUAL (REF TREE) (FLAT2 TREE))),

which simplifies, unfolding ATOM, REF, and FLAT2, to the following two new goals:

The induction step can be broken down into two cases, according to whether the CAR of TREE is a CONS or not. Case 1.2 is the first case when the CAR of TREE is not a CONS.

Case 1.2.
  (IMPLIES (AND (LISTP TREE)
    (EQUAL (REF (CAR TREE))
      (FLAT2 (CAR TREE)))
    (EQUAL (REF (CDR TREE))
      (FLAT2 (CDR TREE)))
    (NOT (LISTP (CAR TREE))))
  (EQUAL (APPEND (REF (CAR TREE))
    (REF (CDR TREE)))
  (CONS (CAR TREE)
    (REF (CDR TREE))))).

However this simplifies further, rewriting with the lemmas CDR-CONS and CAR-CONS, and opening up the functions ATOM, REF, FLAT2, APPEND, and LISTP, to:

T.

Here is the second case, when the CAR of TREE is a CONS.

Case 1.1.
  (IMPLIES (AND (LISTP TREE)
    (EQUAL (REF (CAR TREE))
      (FLAT2 (CAR TREE)))
    (EQUAL (REF (CDR TREE))
      (FLAT2 (CDR TREE)))
    (LISTP (CAR TREE)))
  (EQUAL (APPEND (REF (CAR TREE))
    (REF (CDR TREE)))
  (CONS (REF (CAR TREE))
    (REF (CDR TREE))))).

Applying the lemma CAR-CDR-ELIM, we now replace TREE by (CONS X Z) to eliminate (CAR TREE) and (CDR TREE). We would thus like to prove the new goal:
(IMPLIES (AND (EQUAL (REF X) (FLAT2 X))
  (EQUAL (REF Z) (FLAT2 Z))
  (LISTP X))
  (EQUAL (APPEND (REF X) (REF Z))
  (CONS (REF X) (REF Z))))).

We use the first equality hypothesis by cross-fertilizing
(FLAT2 X) for (REF X) and throwing away the equality. We
would thus like to prove the new formula:

(IMPLIES (AND (EQUAL (REF Z) (FLAT2 Z))
  (LISTP X))
  (EQUAL (APPEND (FLAT2 X) (REF Z))
  (CONS (REF X) (REF Z))))

We use the above equality hypothesis by cross-fertilizing
(FLAT2 Z) for (REF Z) and throwing away the equality. The
result is the new conjecture:

(IMPLIES (LISTP X)
  (EQUAL (APPEND (FLAT2 X) (FLAT2 Z))
  (CONS (REF X) (REF Z))))

which we will name *1.1.

We will try to prove it by induction. There are four
plausible inductions. They merge into two likely candidate
inductions, both of which are unflawed. Since both of these
are equally likely, we will choose arbitrarily. We will
induct according to the following scheme:

(AND (IMPLIES (AND (LISTP X)
  (ATOM (CAR X))
  (P Z (CDR X))
  (P Z X))
  (IMPLIES (AND (LISTP X)
    (NOT (ATOM (CAR X)))
    (P Z (CAR X))
    (P Z (CDR X))
    (P Z X))
  (IMPLIES (NOT (LISTP X)) (P Z X))).

Linear arithmetic and the lemmas CDR-LESSP and CAR-LESSP
can be used to establish that the measure (COUNT X) decreases
according to the well-founded relation LESSP in each induction
step of the scheme. The above induction scheme produces six
new conjectures:

Case 6. (IMPLIES (AND (ATOM (CAR X))
  (NOT (LISTP (CDR X)))
  (LISTP X))
  (EQUAL (APPEND (FLAT2 X) (FLAT2 Z))
  (CONS (REF X) (REF Z)))).
This simplifies, applying CDR-CONS and CAR-CONS, and expanding the definitions of ATOM, FLAT2, APPEND, and REF, to:

\[
(\text{IMPLIES } (\text{AND } (\text{NOT } (\text{LISTP } (\text{CAR } X))) \\
(\text{NOT } (\text{LISTP } (\text{CDR } X)))) \\
(\text{NOT } (\text{LISTP } X))).
\]

Applying the lemma CAR-CDR-ELIM, we now replace X by \( (\text{CONS } V \ W) \) to eliminate (CAR X) and (CDR X). The result is:

\[
(\text{IMPLIES } (\text{NOT } (\text{LISTP } V)) (\text{LISTP } W)),
\]

which has two irrelevant terms in it. By eliminating these terms we get the conjecture:

\[ F. \]

Why say more?

************** F A I L E D **************

The proof fails, and it fails for the particular case when the student's code simplifies to the buggy code. The failed verification condition was Case 1.1:

\[
(\text{IMPLIES } (\text{AND } (\text{LISTP TREE}) (\text{LISTP } (\text{CAR TREE}))) ;\text{Case} \\
(\text{EQUAL } (\text{REF } (\text{CAR TREE})) \\
(\text{FLAT2 } (\text{CAR TREE}))) ;\text{Inductive Hypothesis} \\
(\text{EQUAL } (\text{REF } (\text{CDR TREE})) \\
(\text{FLAT2 } (\text{CDR TREE}))) ;\text{Inductive Hypothesis} \\
(\text{EQUAL } (\text{APPEND } (\text{REF } (\text{CAR TREE})) \\
(\text{REF } (\text{CDR TREE}))) ;\text{from Reference Code} \\
(\text{CONS } (\text{REF } (\text{CAR TREE})) \\
(\text{REF } (\text{CDR TREE}))) ;\text{from Student Code})
\]

Note that the invalid verification condition above will become a theorem if we replace the code fragment from the student's code:

\[
(\text{CONS } (\text{REF } (\text{CAR TREE})) (\text{REF } (\text{CDR TREE})))
\]

in the verification condition by the code fragment from the reference code:

\[
(\text{APPEND } (\text{REF } (\text{CAR TREE})) (\text{REF } (\text{CDR TREE})))
\]

If we alter the student's program so that this verification condition would be produced, the resulting debugged student program is:

\[
(\text{DEFUN FLAT2 } (\text{TREE}) \\
(\text{IF } (\text{LISTP TREE}) \\
(\text{IF } (\text{ATOM } (\text{CAR TREE})) \\
(\text{CONS } (\text{CAR TREE}) (\text{FLAT2 } (\text{CDR TREE}))) \\
(\text{APPEND } (\text{REF } (\text{CAR TREE})) ;\text{Bug fix from reference code} \\
(\text{REF } (\text{CDR TREE}))) \\
(\text{CONS } \text{TREE} \text{NIL})))
\]
Thus we have debugged the student’s program by enforcing the verification condition (Case 1.1), using the stored code as a source of corrections. However the student’s code refers to the reference function \texttt{REF}, and the student should have no knowledge of this function. These references can be eliminated by using the inductive hypotheses:

\begin{verbatim}
(EQUAL (REF (CAR TREE))  ;Inductive Hypothesis
  (FLAT2 (CAR TREE)))

(EQUAL (REF (CDR TREE))  ;Inductive Hypothesis
  (FLAT2 (CDR TREE)))
\end{verbatim}

to replace occurrences of \texttt{REF} by occurrences of \texttt{FLAT2}. The replacement occurs in the bug fix in both the verification condition and in the debugged program. This process of replacing stored function identifiers by student function identifiers using inductive hypotheses is called \textit{normalization} in Talus. The final debugged program is:

\begin{verbatim}
(DEFUN FLAT2 (TREE)
  (IF (LISTP TREE)
    (IF (ATOM (CAR TREE))
      (CONS (CAR TREE) (FLAT2 (CDR TREE)))
      (APPEND (FLAT2 (CAR TREE))  ;Normalization
        (FLAT2 (CDR TREE)))))
    (CONS TREE NIL))
\end{verbatim}

The key points of this small example are:

- \textit{Computational Semantics}: By reasoning about computational semantics a program debugger can distinguish between unanticipated implementations that are correct (e.g. \texttt{FLAT1}) and buggy (e.g. \texttt{FLAT2}). Note that no plan library was required, and no pattern matching occurred.

- \textit{Bug Detection and Isolation}: Bugs can be detected by determining where a proof of equivalence breaks down. Case splitting isolates bugs to code fragments. The reference program acts as a specification for the student’s program.

- \textit{Bug Correction}: Bugs can be corrected by altering the student’s program to restore the proof of equivalence. The reference program acts as a source of corrections.

- \textit{Normalization}: The inductive hypotheses allow calls to the reference program to be replaced by calls to the (debugged) student program in bug corrections.

### 1.5 Talus Debugging Examples

The last section introduced the basis for the debugging methodology that Talus uses. As it stands the approach is quite limited, and cannot be used to debug the following:

- Functions that loop forever due to missing conditional expressions. These functions cannot be accepted by the theorem prover since they violate the definitional principle.

- Functions that have formal variables that have different names or roles than those in the reference function.

- Solutions that use multiple functions to solve the task.

- Functions that correctly solve the task but which use a different algorithm than the reference function. Additional lemmas will be required to show that functions based on alternate algorithms are in fact equivalent.
• Functions with COND, LAMBDA, PROG, or mapping functions. These special forms are not present in the logic.

• Functions with side effects. Side effects are not easily expressible in the Boyer-Moore Logic.

What has been presented so far is only a small part of the complete Talus system; the complete Talus system does not suffer from the limitations above. Figure 1-2 illustrates the development of Talus. The bug detection and correction process illustrated can be viewed as the core of Talus. Each layer around the core removes one of the limitations above.

Figure 1-2: Talus System Development

These additional capabilities are best illustrated by examples. The additional reasoning capabilities required to support these extensions will be emphasized. Only Talus input and output will be shown here. Later the debugging process will be explained in full detail.
1.5.1 Missing Conditional Tests and Infinite Loops in MEMBER

The MEMBER task will be used to illustrate what happens when the student provides a function definition that has one or more missing conditional tests. If defined into LISP, the resulting function might loop forever, cause an error break, or merely return incorrect results. The task is:

Write a function that determines whether an item is in a proper list. You need only examine the list at the top level.

and the reference function for MEMBER is:

(DEFUN MEMBER (ITEM BAG)
  (IF (NLISTP BAG)
      F
      (IF (EQUAL ITEM (CAR BAG))
          T
          (MEMBER ITEM (CDR BAG))))
)

Assume the student provides the following definition:

(DEFUN MEM (X L)
  (IF (EQUAL X (CAR L))
      T
      (MEM X (CDR L))))

This can loop forever or break, depending on the LISP dialect, for a proper list L not containing X. Talus detects the missing condition through a process involving symbolic evaluation, and returns the following debugged code:

(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (MEM X (CDR L))))
)

If the student had omitted the second conditional test and provided this solution:

(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (MEM X (CDR L))))

Talus would detect the missing conditional test and return the same debugged code as before.

And finally if the student had entered only:

(DEFUN MEM (X L) (MEM X L))

Talus would detect both missing conditions, insert them in correct order, and then debug the remainder of the code, replacing (MEM X L) by (MEM X (CDR L)). Again the debugged code would be the same.

The detection of missing conditions requires an ability to symbolically evaluate the student's code. The Boyer-Moore Logic and theorem prover can be used here. Missing conditions are corrected by using conditionals from the reference function. Again the reference function acts both as specification and as a source of corrections.
Talus reasons about logical implications in performing symbolic evaluation of programs. Symbolic evaluation allows Talus to simplify function definitions by assuming that the input data satisfies certain predicates. This allows a case analysis to be performed and bugs to be isolated to particular program fragments. In order to perform this symbolic evaluation Talus determines if the assumptions about the input data imply that a conditional test evaluates to be true, false, or an error. If none of these implications can be made, then case splitting is necessary, indicating that a missing condition has been detected.

Infinite loops, such as in the last definition of MEM, are detected as violations of well-founded relations. A well-founded relation has no infinite sequence of objects, each of which is smaller than its predecessor in the sequence ([Boyer 79], p. 13). Intuitively, we require that the arguments in recursive calls diminish in some manner that guarantees that a base case in the function will eventually be reached.

For example if X is a CONS and we always take the CDR of X in all recursive calls eventually X will not be a CONS. If the function does not call itself recursively in that case then the function always terminates for all inputs. The verification condition that Talus would check for each recursive call would be:

\[
\text{(IMPLIES (LISTP X) (LESSP (COUNT (CDR X)) (COUNT X)))}
\]

where COUNT is a function that measures the complexity of an data structure. LESSP is a well-founded relation. Both are defined in the Boyer-Moore Logic. Talus would discover the infinite loop in the last definition of MEM, even after inserting the missing conditions, since \(\text{L} \) in the recursive call \((\text{MEM X L})\) does not diminish according to the measure COUNT and well-founded relation LESSP.

1.5.2 Algorithm Recognition, Multiple Functions and Formal Variables in MEMTREE

The previous debugging examples assumed that the student's solution uses only one function and the same algorithm as the reference function. What about tasks requiring multiple functions and where more than one algorithm can solve the same task? We also need to consider how multiple formal variables are dealt with. To illustrate how Talus copes with these complications, consider the MEMTREE task:

Write a function that determines whether an atom is one of the leaves of a tree.

Assume the student enters the following buggy solution:

\[
\text{(DEFUN MEMTR (X TR) (IF (ATOM TR) NIL (OR (MEMTR X (CAR TR)) (MEMTR X (CDR TR))))})
\]

Talus will return the following debugged code:

\[
\text{(DEFUN MEMTR (X TR) (IF (ATOM TR) (EQUAL X TR) (OR (MEMTR X (CAR TR)) (MEMTR X (CDR TR))))})
\]
The basic debugging approach already presented is adequate for this debugging, but Talus also allows multiple algorithms and multiple functions. The algorithm in the solution above explores the CAR and CDR of the tree recursively for the item being searched for. The solution below uses a different algorithm. The tree is first flattened into a bag. Then the item is tested for membership in the bag. Here is a buggy implementation of this second algorithm:

\begin{verbatim}
(DEFUN MEMTR (AT CONS)
  (IN AT (FLAT NIL CONS)))
(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
    ANS
    (FLAT (FLAT ANS (CDR TR))
      (CAR TR))))

(DEFUN IN (X L)
  (IF (LISTP L)
    (IF (EQUAL L (LIST X))
      L
      (IF (NOT (EQUAL (CAR L) X))
        (IN X (CAR L))
        L))
    NIL))
\end{verbatim}

Talus would return the following debugged code for this solution:

\begin{verbatim}
(DEFUN MEMTR (AT CONS)
  (IN AT (FLAT NIL CONS)))
(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
    (CONS TR ANS)
    (FLAT (FLAT ANS (CDR TR))
      (CAR TR))))

(DEFUN IN (X L)
  (IF (LISTP L)
    (IF (EQUAL L (LIST X))
      L
      (IF (NOT (EQUAL (CAR L) X))
        (IN X (CDR L))
        L))
    NIL))
\end{verbatim}

To allow for multiple functions and algorithms, Talus needs to perform algorithm recognition. Associated with each task are possible algorithms that can solve that task. Each algorithm is itself associated with reference functions that implement that algorithm. Using partial matching heuristics, Talus determines the algorithm the student used and the mapping between reference functions and student functions. Then formal variables between paired functions must be paired. Finally debugging can proceed as before.

This algorithm recognition and function mapping process requires the following kinds of heuristic reasoning about computational semantics:

- Determining the algorithm that is most similar to the student’s algorithm.
• Determining the best mapping between the algorithm's reference functions and the student's functions. This mapping should optimize the similarity of paired functions and minimize unpaired functions.

• Determining the best mapping between the formal variables in two paired functions. This mapping should optimize the similarity of the roles of the paired formal variables.

Since it is very difficult to formalize "similarity" the reasoning above is termed heuristic, in contrast to the formal reasoning about precisely defined functions that occurs in the theorem prover.

The above example illustrated Talus debugging a buggy implementation to a correct algorithm. Talus can also debug solutions where the algorithm itself is incorrect since it contains a design flaw or reflects a misunderstanding of the task.

1.5.3 Special Forms and Side Effects in PALINDROMES

Talus can debug programs containing CONDs, LAMBDA expressions, mapping functions, and PROGs. Side effects to shared list structure are also handled. None of these special forms or side effects are present in the Boyer-Moore Logic. To illustrate these capabilities, consider the PALINDROMES task:

Write a function that takes a list of elements and returns, in order, those elements which are lists that are unchanged when reversed. For example,

PALINDROMES of (8 I (A B A) (A D) (E) NIL ((1 2) (1 2)) 9)) is...

((A B A) (E) ((1 2) (1 2)))

NOTE: all atoms that are elements in the input list are discarded. Thus,

PALINDROMES of (X Y Z) is NIL.

The student's buggy solution, shown below, uses MAPCAN. MAPCAN is like MAPCAR in that it applies a function to each element in a list, but is different in that it splices the results together with NCONC. Note that this splicing can produce side effects to shared list structure.

(DEFUN RETURN-PALINDROMES (L)
 (MAPCAN
 (FUNCTION
 (LAMBDA (EL)
 (IF (AND (LISTP EL) (EQUAL EL (REV EL)))
 (LIST L)
 NIL))
 (CDR L))))

(DEFUN REV (LIS)
 (PROG (ANS)
 CONTINUE
 (IF (NULL LIS) (RETURN ANS) NIL)
 (SETQ ANS (CONS LIS ANS))
 (SETQ LIS (CDR LIS))
 (GO CONTINUE))

The above example illustrated Talus debugging a buggy implementation to a correct algorithm. Talus can also debug solutions where the algorithm itself is incorrect since it contains a design flaw or reflects a misunderstanding of the task.

1.5.3 Special Forms and Side Effects in PALINDROMES

Talus can debug programs containing CONDs, LAMBDA expressions, mapping functions, and PROGs. Side effects to shared list structure are also handled. None of these special forms or side effects are present in the Boyer-Moore Logic. To illustrate these capabilities, consider the PALINDROMES task:

Write a function that takes a list of elements and returns, in order, those elements which are lists that are unchanged when reversed. For example,

PALINDROMES of (8 I (A B A) (A D) (E) NIL ((1 2) (1 2)) 9)) is...

((A B A) (E) ((1 2) (1 2)))

NOTE: all atoms that are elements in the input list are discarded. Thus,

PALINDROMES of (X Y Z) is NIL.

The student's buggy solution, shown below, uses MAPCAN. MAPCAN is like MAPCAR in that it applies a function to each element in a list, but is different in that it splices the results together with NCONC. Note that this splicing can produce side effects to shared list structure.

(DEFUN RETURN-PALINDROMES (L)
 (MAPCAN
 (FUNCTION
 (LAMBDA (EL)
 (IF (AND (LISTP EL) (EQUAL EL (REV EL)))
 (LIST L)
 NIL))
 (CDR L))))

(DEFUN REV (LIS)
 (PROG (ANS)
 CONTINUE
 (IF (NULL LIS) (RETURN ANS) NIL)
 (SETQ ANS (CONS LIS ANS))
 (SETQ LIS (CDR LIS))
 (GO CONTINUE)))
The debugged code that Talus returns is:

```lisp
(DEFUN RETURN-PALINDROMES (L)
  (MAPCAN
    (FUNCTION
      (LAMBDA (EL)
        (IF (AND (LISTP EL) (EQUAL EL (REV EL)))
          (LIST EL)
          NIL))
      L))

(DEFUN REV (LIS)
  (PROG (ANS)
      CONTINUE
        (IF (NULL LIS) (RETURN ANS) NIL)
        (SETQ ANS (APPEND (LIST (CAR LIS)) ANS))
        (SETQ LIS (CDR LIS))
        (GO CONTINUE))
)
```

To debug this solution Talus first simplified the student’s program by applying a sequence of program simplification transforms to reduce the student’s solution to an equivalent solution that uses no PROGs, LAMBDA’s, or mapping functions. The list splicing (NCONC) performed by MAPCAN was replaced by list concatenation (APPEND) since Talus established that this transformation would not alter the values returned by the program.

The reduced program was then debugged and the bug fixes traced back to the original program. Reasoning about the applicability and sequencing of program transformations is required to reduce the student’s program. Further reasoning about how to trace the edits back to the original program through these transformations is required after the simplified program is debugged.

In this example, NCONC was replaced by APPEND so the simplified program contained no side effect producing functions. Talus can also debug programs where this kind of simplification is impossible and the reduced program can produce side effects. Section 3.4.5 presents an example where Talus debugs a program with side effect producing functions.

1.6 Summary

The key points of this chapter are:

- **All** automated program debuggers are limited by an inability to reason about computational semantics.
- Limitations in automated program debuggers appear in three forms:
  - Partial debugging, lack of robustness, vague and unreliable diagnostics.
  - Restrictions on the domain, task size, and programming language that reduce solution variability.
  - Reliance on user guidance.
- The Boyer-Moore Logic and the Boyer-Moore Theorem Prover provide a formalism and mechanism for reasoning about the computational semantics of recursive programs and recursively defined data structures.
- Recursive programs can be debugged by repairing a proof of equivalence.
• A correct reference program serves both as a specification and source of corrections.
• Case splitting in the proof allows bugs to be isolated to code fragments.
• Use of inductive hypotheses allows references to the reference program to be removed in corrections to the student’s program.

1.7 Dissertation Overview

Chapter 2 discusses related work in automatic program debugging, providing specific examples of debugging limitations imposed by an inability to reason about computational semantics. The remainder of the dissertation focuses on Talus, its debugging process, evaluation, and possible extensions. Except where noted otherwise, the existing implementation of Talus is fully operational and performs as described.

Chapter 3 provides an overview of Talus, specifying more precisely the variability in solutions allowed, and the debugging capabilities of Talus. An overview of the complete debugging process is provided there and case studies are used to illustrate the process. Chapter 4 discusses the frame-like program representation that Talus uses to represent abstract computational features of recursive programs. Chapter 5 explains how the algorithm recognition process makes use of these features to identify algorithms in buggy student programs. Chapter 6 examines the bug detection process in detail. Chapter 7 discusses how Talus automatically corrects bugs it has detected. Chapter 8 explains how the Talus implementation has been extended to debug programs containing side effects and special forms not present in the core dialect. Chapter 9 presents the results of an empirical evaluation of Talus. Chapter 10 summarizes this research and discusses extensions of Talus to other domains such as LOGO and PROLOG.
Chapter Two

Related Work in Automated Program Debugging

This chapter surveys related work in automated program debugging, illustrating how limitations in performance can be attributed to inabilities to reason about computational semantics. Before looking at specific debuggers, a general model and a classification hierarchy are presented. These facilitate characterizing and comparing the various approaches that have been developed over the past fifteen years. Afterwards, representative systems that illustrate alternate approaches and their limitations are studied.

2.1 A Model of Automated Program Debugging

The program debugging methodologies discussed in this chapter are specializations of the model in Figure 2-1. Task Specifications are mapped to Code Abstractions derived from the code being debugged. This mapping process allows the task of debugging to be decomposed into simpler debugging problems. When decomposition is no longer possible an Implementation critic enumerates discrepancies between code and specifications and interprets these as bugs. These components are explained below.

![Diagram of debugging model]

Figure 2-1: A Model of Automated Program Debugging

The task specifications are precise criteria for successful completion of the task. The following
methods of task specification have been used by the debugging systems cited:

- **Input/Output Pairs** - BIP [Barr 76].
- **Constraints on Program Output** - MYCROFT [Goldstein 74].
- **Expected Program Traces** - PDS6 [Shapiro 83].
- **Predicate Calculus Input/Output Specifications** - Katz & Manna [Katz 76].
- **Model Programs** - LAURA [Adam 80], Talus.
- **Plan Templates** - MENO-II [Soloway 83].
- **Goals to be Achieved, specified in some Problem Description Language** - PROUST [Johnson 84].

Automated debuggers frequently simplify the debugging task by ignoring various details of the code and extracting other features. These features, rather than the original code, are compared to the task specification to detect bugs. Either the code or its execution can be abstractly represented.

Automated debuggers that abstract features directly from the code perform static analysis [Miller 86]. Frequently the code is parsed into an internal graph representation that explicitly represents control and data flow while ignoring details of their implementation. The PLAN formalism of the Programmer's Apprentice Project [Rich 81], used by the debugger SNIFTER [Shapiro 81], is an example. LAURA and MENO-II also parse student programs into graph representations. A less common approach is to synthesize loop invariants from the code.

Automated debuggers that abstract features from code execution on specific examples perform dynamic analysis. The amount of information extracted from a program's execution can vary significantly: BIP merely examines a program's output while SNIFTER records a complete trace of procedure calls and changes to all mutable data structures.

The implementation critic compares task specifications and code abstractions for discrepancies that it can interpret as bugs. Erroneous bug reports (false alarms) result if discrepancies due to correct implementation variants are misinterpreted as bugs. Bugs are missed when the code abstraction process obscures their presence or if the critic cannot detect them. The following indications have been used to detect bugs:

- **Differences between Plan Templates and Parsed Code** - MENO-II, PROUST.
- **Differences between Expected and Actual Execution Traces** - PDS6, BIP.
- **Inability to Formally Verify Program Specifications** - In [Katz 76], Talus.
- **Counterexamples Disproving Computational Equivalence of Student Code and Expected Code** - Talus.
- **Inability to Synthesize Student Code** - GREATERP [Reiser 85], TURTLE [Miller 82].
- **Failed Parses using a Plan Grammar** - In [Ruth 73].

As with many AI problems, a divide and conquer strategy can simplify the overall debugging

---

3This distinction between program debuggers that perform static analysis and those that perform dynamic analysis originated with Minsky and Papert [Hewitt 86].
problem. The box marked MAP in Figure 2-1 performs this problem decomposition function in our model of an automated program debugger. When possible, task specifications and student code are both decomposed into smaller units and then mapped together. Student program fragments should be mapped to the task goals they are intended to achieve. This mapping can be difficult to determine when bugs are present in the student's program. Nondeterminism can arise so subsequent errors can occur and backtracking may be necessary. Sometimes constraints inherent in the programming language allow an easier means of decomposing the debugging problem. Examples of mapping methods employed are:

- **Decomposition of Conjunctive Goals in PROLOG** - PDS6.
- **Case Splitting** - Talus.
- **Plan Recognition** - MENO-II, PROUST, Talus.
- **Teleological Mapping** - MYCROFT. Differs from plan recognition in the lack of a plan library.

What role does computational semantics have in program debugging? Consider that debugging can be broken down into three steps:

- **Determine Code Teleology.** Infer what the student intended the code to do.
- **Determine if Code Teleology Satisfies Task Specifications.** Determine if the intentions are a correct strategy to solve the task.
- **Determine if Code Satisfies Code Teleology.** Determine if the code is correct to satisfy the intentions.

It is in the last step that computational semantics plays an important role. This process can be simplified if code teleology is represented using task specifications. In this representation, code fragments or statements are mapped to the task specifications that the code is intended to achieve. With this representation of code teleology the debugging process simplifies to two steps:

- **Infer a Mapping of Task Specifications to Code.**
- **Determine if Code Satisfies Task Specifications.**

Recent research [Johnson 85] has shown the necessity of the first step. The research presented here demonstrates the value of an ability to reason about computational semantics in both steps.

### 2.2 A Survey of Approaches

Now that we have seen what is required to debug programs automatically, we can examine specific approaches to debugging. Automated debuggers can be classified by their primary means of bug detection as shown in Figure 2-2. The most distinctive split is between those debuggers that rely on program execution on specific examples to detect bugs and those that do not. The former perform **dynamic analysis** while the latter perform **static analysis**.

The four categories under dynamic analysis (see Figure 2-3) differ in the amount and kind of information extracted from a program's execution history. The simplest approach, adopted by BIP, is to compare program output on given examples with expected output. A more comprehensive approach is to examine all program side effects, not just input and output. For example, MYCROFT examines all side effects produced by a program; its domain is LOGO and the side effects are lines drawn and changes to the turtle state. Even more information about a program's execution is recorded by SNIFTER. SNIFTER records a **complete** program execution history including procedure calls and returns, and changes to mutable data structures such as side effects to shared list structure. In between the two extremes of BIP
Approaches to Automated Debugging

Static Analysis
- Plan-based Program Analysis
- Analysis of Loop Invariants
- Analysis of I/O Pairs
- Plan Parsing

Dynamic Analysis
- Program Verification Approaches
- Debugging Recursive Programs by Debugging Inductive Proofs
- Analysis of Program Side Effects
- Trace Analysis
- Analysis by Synthesis
- Heuristic Plan Recognition

Figure 2-2: A Taxonomy of Automated Debugging Approaches

and SNIFFER, PDS6 only examines procedure calls and returns, and monitors rather than records program execution.

Dynamic Analysis

- Analysis of I/O Pairs
- Analysis of Program Side Effects
- Trace Analysis
- Analysis of Program Execution History

Figure 2-3: A Taxonomy of Debuggers that use Dynamic Analysis

Debuggers using static analysis can be further divided into those using program verification approaches, such as Talus, and those using plan-based program analysis (see Figure 2-4). The
debugging approaches in the plan-based program analysis category are form-based; this means that they look for surface structural forms [Rich 86]. Most of the inference in these systems is performed in looking for these forms or accounting for differences between forms and actual code. These systems are syntactic in that they do not represent and reason about the semantics of programming language constructs. In contrast the debugging approaches based on program verification do perform this kind of reasoning. These approaches attempt to prove a student’s program correct according to formal specifications. Proof failures are interpreted as bugs in the student’s program. These bugs are corrected by altering the program to repair the proof.

Figure 2-4: A Taxonomy of Debuggers that use Static Analysis

We will consider approaches based on plan-based program analysis first. These approaches look for instances of plans in either the student’s code or its parsed representation. These plans are structural forms, such as code templates, that provide a mapping between task specifications and student code.

4This statement does not mean that these systems have absolutely no knowledge of programming language semantics, rather that this knowledge is frozen in the form of rewrite rules or plan transformations that account for common implementation variants. This inability to reason about computational semantics during the debugging process limits the ability of these systems to correctly debug unanticipated implementations.
Recall that this mapping represents the teleology, i.e. the intended purpose of the components of the student's code, and allows the overall debugging problem to be decomposed into smaller and simpler debugging problems. Plans include common task decompositions and common implementations of primitive goals. Each plan associates some part of the user's code with some part of the overall task specifications. Associated with plans are features that indicate when the plan is appropriate to use. Different debuggers use different metrics to determine how well plans match code. A common choice is to measure the number of bugs that result assuming that a plan matches. The plan that minimizes the number of bugs found is the best match.

Bug detection in plan-based program analysis relies on detecting mismatches between plans and code. Some mismatches are recognized as common implementation variants of expected code using plan transformation rules; these are not interpreted as bugs. Others can be recognized as instances of specific program bugs by bug rules. Mismatches that cannot be accounted for by either of these two methods can:

- **Result in incomplete or aborted program analysis.**
- **Be interpreted as bugs always.** In this case correct implementations result in false alarms.
- **Be ignored.** In this case buggy implementations are not detected.
- **Generate Warnings.** In this case the debugger avoids either of the above errors by being noncommittal or vague. An example of such a warning ([Johnson 85], p. 223) is:

```
1. This analysis is incomplete. There are parts of this program that I could not understand, so this analysis may not be totally correct.

This program does not implement the maximum.
I had problems with some of the code.
The statements in question are:
IF RAINFALL > HIGHESTRAINFALL THEN ...
RAINFALL := HIGHESTRAIN
```

The three categories of Plan-based Program Analysis are distinguished by their representation of plans. In plan parsing, grammars represent plans and student programs are parsed in terms of the grammar. In heuristic plan recognition plans are stored in a plan library and partially matched to student code. In analysis by synthesis, plans are synthesized to the code level. There is some overlap in categories, as indicated by the dashed lines in Figure 2-4. PROUST can synthesize plans, but not to the code level. Talus detects certain kinds of bugs in its algorithm recognition phase but its primary means of bug detection is based on program verification.

In the plan parsing approach, grammars represent plans and student programs are parsed in terms of the grammar. This method is similar to syntactic analysis in natural language processing. Ruth's debugger [Ruth 73] analyzes simple sorting programs with this approach and a grammar similar to an ATN (augmented transition network).

In heuristic plan recognition plans are typically retrieved from a plan library and partially matched to parsed student code. MENO-II and PROUST use this approach, but LAURA matches a parsed model program to the parsed student code, rather than use a plan library. PROUST matches more than one plan template to a single program, each template corresponding to a task goal. The finer grain size templates and context sensitive matching of PROUST enable it to perform more refined analyses than MENO-II. Plan transformation rules and bug rules enable PROUST to explain match discrepancies as
implementation variants or known bugs.

Although PROUST synthesizes complete plans from a collection of plan templates, it does not synthesize code and thus is not categorized here as using analysis by synthesis. In analysis by synthesis the debugger can synthesize complete solutions to the assigned task. This has the advantage that an ITS using such a system can synthesize code to correct bugs or complete solutions when the student cannot proceed. However, the number of solutions that can be generated in a nontrivial domain is so large that the task size must be severely limited. Alternatively, the tutor can subdivide tasks into smaller steps that must be solved one at a time; this is the approach taken by GREATERP.

2.3 Exemplars of Alternate Debugging Approaches

This section illustrates various approaches by presenting exemplars of alternate approaches. Specific examples support the claim that each debugging system is limited by abilities to reason about computational semantics. We will examine three fully implemented systems:

- **PDS6** - illustrating dynamic analysis, and trace analysis in particular,
- **PROUST** - illustrating the heuristic plan recognition approach,
- **GREATERP** - illustrating the analysis by synthesis approach,

and also consider a design for an automated debugger based on an analysis of loop invariants.

2.3.1 Dynamic Analysis: PDS6

PDS6 [Shapiro 83] is a debugging system that interactively debugs pure PROLOG programs by monitoring program execution. The user provides examples of bugs and answers queries about the correctness of procedure traces. Infinite loops are caught by limiting the depth of the stack of procedure calls. PDS6 calls its debugging procedures recursively on the conjuncts in conjunctive goals to isolate bugs.

PDS6 detects three types of bugs in programs:

- **Correctness** - Does the program succeed on goals for which it should fail? PDS6 detects these bugs by simulating program execution on such a goal and returning the first procedure call that returns an incorrect output.

- **Finite Failure** - Does the program fail on goals for which it should succeed? These bugs are detected by comparing an expected top-level execution trace for such a goal with an actual trace. When a procedure call in the actual trace erroneously fails, the debugging algorithm is called recursively on the procedure call.

- **Nontermination** - Does the program exceed a bounded stack size on some computation? Nontermination bugs are trapped by allowing only a fixed stack size during program execution.

PDS6 corrects bugs by searching bug equivalence classes. A bug equivalence class is a set of clauses that are perturbations of a buggy clause; the perturbations introduced reflect common errors. One bug equivalence class consists of variable misspellings. Another is incorrect use of arithmetic tests. The buggy clause is used as a seed for the search. Instances of the clause that are correct are used to constrain the search: only clauses that also cover those instances are acceptable. Note that the bug equivalence classes to consider are parameters to the bug correction algorithm and corrections for clauses with multiple bugs from differing equivalence classes cannot be discovered.
Although PDS6 can compare actual program execution with expected execution to isolate bugs to the level of incorrect clauses and uncovered goals, it requires an oracle to provide information that it cannot infer. The oracle, typically the user, provides the following information:

- **The Correctness of Procedure Calls** - E.g. "Query: append([1,2],[2,3],[1,2,3])?" to which the user would answer "yes" or "no" according to whether or not that is a correct goal for append to succeed on.

- **Expected Output of Procedure Calls that Fail** - E.g. "Query: isort([2,1],X)?" to which the user answers "yes" or "no" if isort can succeed on that goal for any binding of X. If the user replies "yes", PDS6 asks "Which X?" and the user must supply a value for X that causes the goal to succeed.

- **Violation of Well-founded Orderings** - E.g "Is (insert(1,[3],X),insert(3,[1],X)) a legal call?" A reply of "no" means that this sequence of goals violates a well-founded ordering that should be true for insert.

- **Correct Instances of Buggy Clauses** - E.g. "What is a reason for (append([X][Y],Z,U) :- append([X][Y],Z,[X][U])?" The user must answer with a goal that this clause should cover, e.g. "append([1,2],[3,4],[1,2,3,4])."

The oracle can be partially mechanized if constraints on correct solutions are known. However the oracle is implemented, PDS6 always relies on the user to provide examples of program bugs.

The user, and the oracle if it is distinct from the user, provide the following inference capabilities that are lacking in PDS6:

- **Boundary Cases to Test.** The user is responsible for testing all boundary cases in the code.

- **Case analysis.** Again the user is responsible for ensuring that the examples test all parts of the code.

- **Algorithm recognition, role of formal variables.** PDS6 need not infer these since the oracle knows the algorithm and the purpose of each formal variable, and uses this information in answering queries.

- **Acceptable Bug Corrections.** The user accepts or rejects the proposed bug corrections of PDS6. Constraints provided by user procedure declarations also aid PDS6 in determining acceptable corrections.

- **Clauses to Modify and Add.** When a goal is uncovered the user must inform PDS6 which clause should have covered it. If no clause should have covered it, the user must add the missing clause manually.

- **Goals to Cover in Corrections.** These also aid in the search for corrections to buggy clauses, since acceptable corrections must cover these goals.

Thus both an inability to infer code teleology and limitations in the ability to reason about computational semantics are compensated for by user query. For the reader familiar with PROLOG, Appendix II provides a detailed trace of the debugging of a quicksort program, with particular emphasis paid to the role of user query and the dependence of the debugger on it.

In summary, PDS6

- **Can Perform Limited Reasoning about Execution Traces to Isolate Bugs.**

- **Cannot Reason about Operator Semantics.** E.g. it cannot prove that < subsumes =<.

- **Relies Heavily on User Query.**

- **Cannot Debug Programs with Side Effects.**
• Cannot Correct Bugs Outside of its Equivalence Classes.
• Only Debugs Programs with respect to the Examples Provided.

The last limitation similarly applies to all debuggers that rely on dynamic analysis: they can only debug student programs with respect to a finite set of examples. The student’s program may always contain a bug that the analyzer cannot discover due to lack of an example that will uncover that bug.

2.3.2 Heuristic Plan Recognition: PROUST

The heuristic approach, exemplified by PROUST, uses stored plan templates to match against parsed student programs. PROUST has a plan library that associates task goals with plan templates. Plan templates are associated with expected code in the user’s plan that they can match against; in addition a plan may specify subgoals to be added to PROUST’s agenda of goals. The intentions of the statements in the student’s program are fully determined if each statement has been matched to some part of a plan and there are no unachieved goals on the goal agenda. When more than one plan can match, heuristics attempt to select the one that predicts the fewest bugs in the student’s code. Heuristics must be used since an exhaustive search is not practical.

The success of this approach relies not only on the heuristics but on the plans in the plan library and their coverage of alternate task solutions. For example, there are three plans for achieving the goal of averaging a set of numbers read in from the terminal. One of the plans is shown below:

AVERAGE PLAN


Posterior goals:
  Count (?New, ?Count)
  Sum (?New, ?Sum)
  Guard Exception (component Update: of goal Average.
  ( (?Count from goal Count) = 0 )

Exception condition:
  ( (?Count from goal Count) = 0 )

Template:
  (component Mainloop: of goal Read & Process)
  followed by:

The variables ?Avg, ?Sum, ?Count, and ?New will match against student identifiers. The code template associated with this plan occurs after “Update:” in the template slot. The posterior goals slot adds three goals to the goal agenda:

• The Count Goal. This counts the number of inputs accepted.

• The Sum Goal. This sums up the valid inputs.

• The Guard Exception Goal. This prevents a divide by zero error in the Average goal when the variable ?Count in the Count goal is zero.

The information in the template slot indicates that the main loop of the Read & Process goal, which reads in the inputs, should precede the computation of the average.
Plan-difference rules are another means of increasing the coverage of the plans in the plan library. These account for discrepancies between student code and the code predicted by matched plan templates. There are two kinds:

- **Plan Transformation Rules** - These are essentially rewrite rules that allow for common implementation variants. For example, if a PROUST plan template predicts the boolean expression "NEW > 0" but the student's code at that point is "0 < NEW" a rule that establishes their equivalence fires.

- **Bug Rules** - These rules account for differences due to specific bugs. For example, the Typo rule can explain "9999" as a typographical error if "99999" was expected.

Code that cannot be accounted for by these rules cannot be analyzed.

When the plans and plan-difference rules of PROUST are sufficient to explain the intended purpose of student code, PROUST generates accurate bug reports. Misconceptions can also be inferred. For example, in one program where a student incorrectly uses the statement

```plaintext
NEW := NEW + 1
```

instead of

```plaintext
READ (NEW)
```

PROUST generates the bug report shown in Figure 2-5 [Johnson 85].

It appears that you were trying to use line 12 to read the next input value. Incrementing NEW will not cause the next value to be read in. You need to use a READ statement here.

The statement in question is:

```plaintext
NEW := NEW + 1
```

Figure 2-5: PROUST Bug Report Suggesting a Misconception

An empirical evaluation of PROUST provides a succinct summary of its strengths and weaknesses. Figure 2-6 shows the best results of PROUST in analyzing a task called RAINFALL. It shows that

- PROUST is highly accurate (94 %) in diagnosing bugs when it fully accounts for the intention of all code in terms of its plans, plan transformation rules, and bug rules.

- There are a significant number of false alarms, even when programs are fully analyzed.

- There are a significant number of programs that cannot be fully analyzed. Performance in bug recognition falls off sharply for such programs.

The performance of PROUST on a more difficult task shows that these flaws are magnified as task difficulty increases (see Figure 2-7). For the more difficult BANK task, PROUST can only perform a full analysis of 50 % of the student programs.

---

5In the RAINFALL task students are asked to compute the average and maximum of a sequence of numbers entered as terminal input. The number of nonzero and nonnegative inputs must also be printed.

6In the BANK task students are asked to process simple account deposits and withdrawals, printing out summaries of the transactions and the final account balance.
Partial or failed analyses, false alarms, and unrecognized bugs are due to:

- *Failure of Plan Recognition Heuristics.* The wrong plan was selected when more than one applied.

- *Misuse of Variables.* The student’s code uses variables inconsistently; this complicates plan matching.
- *Constructs that PROUST cannot analyze.* E.g. NOT, GOTO, and boolean variables.

- *Failure of Plan-Difference Rules.* A rule to explain an implementation variant or a bug is missing, or the rule is present but does not fire since its triggering conditions are incorrect.

An increased ability to reason about computational semantics could alleviate the last two problems. By reasoning about the semantics of constructs, PROUST could accept a larger PASCAL subset and recognize alternate implementations with fewer plan-difference rules.

A specific example where an ability to reason about computational semantics reduces the need for plan-difference rules is shown in the following four plans. These plans represent implementation variants of the assignment $\text{Total} := \text{Total} - \text{New} - \text{Const}$.

---

**COMPOUND DEDUCT PLAN 1**

| Variables: | $\text{Total}, \text{New}$ |
| Constants: | $\text{Const}$ |
| Template:  | $\text{Update}$: $\text{Total} := (\text{Total} - \text{New})$
|            | $\text{Total} := (\text{Total} - \text{Const})$ |

**COMPOUND DEDUCT PLAN 2**

| Variables: | $\text{Total}, \text{New}, \text{Temp}$ |
| Constants: | $\text{Const}$ |
| Template:  | $\text{Update}$: $\text{Temp} := (\text{Total} - \text{New})$
|            | $\text{Total} := (\text{Temp} - \text{Const})$ |

**COMPOUND DEDUCT PLAN 3**

| Variables: | $\text{Total}, \text{New}, \text{Temp}$ |
| Constants: | $\text{Const}$ |
| Template:  | $\text{Update}$: $\text{Temp} := (\text{New} + \text{Const})$
|            | $\text{Total} := (\text{Total} - \text{Temp})$ |

**COMPOUND DEDUCT PLAN 4**

| Variables: | $\text{Total}, \text{New}, \text{Temp}$ |
| Constants: | $\text{Const}$ |
| Template:  | $\text{Update}$: $\text{Temp} := (\text{Total} - \text{Const})$
|            | $\text{Total} := (\text{Temp} - \text{New})$ |

---

An ability to reason about the computational semantics of the operators "$+$" and "$-$" would subsume these plans since the assignments could be shown to be equivalent.

2.3.3 Analysis by Synthesis: GREATERP

The analysis by synthesis approach relies on an ability to synthesize code that satisfies task specifications. The student's design decisions are either inferred from the completed program, or traced as the student develops the program. The latter approach is the model tracing approach used by GREATERP. As each token in the student's program is entered, GREATERP checks to see if the student is following a design path known to be correct or buggy. Buggy paths are pruned as soon as they are
detected by explaining the student’s misconception, then allowing the student to try again. If the student cannot determine how to proceed GREATERP can assist him and if necessary provide the code.

The advantage of this approach is that very specific misconceptions can be diagnosed and advice can be given immediately, in the context of the error. The disadvantage of this approach is that the student is highly constrained in the solutions that can be developed. The student must conform to the task decomposition and coding sequence that GREATERP enforces. As seen in Section 1.1.2, some misconceptions will not surface for simple tasks.

GREATERP dynamically represents the student’s partial task solution as the solution is coded. At any point GREATERP assumes that it correctly knows all previous design decisions; these design decisions constitute its student model. This assumption will be violated if GREATERP allows solutions to tasks to use one of several algorithms where any of the algorithms has overlapping design paths. The assumption will also be violated if GREATERP allows alternate functional decompositions, if any of the decompositions have overlapping design paths. When two design paths leading to alternate implementations overlap, GREATERP cannot determine which design is being followed until the implementation is further elaborated. Thus GREATERP must either:

- Disallow these kinds of variability.
- Add user query to disambiguate the intended design.
- Increase its reasoning capabilities so parallel paths can be followed until pruning can occur.
- Assume a design path and allow backtracking if the wrong design is chosen.

If these kinds of variability are allowed some discrepancies between predicted implementations and expected implementations can be explained either in terms of an incorrect design path being assumed, or in terms of a bug in the student’s program. Again GREATERP may have to defer judgement before this situation can be disambiguated. This is similar to the problem of interpreting discrepancies in heuristic plan recognizers: either the wrong plan has been chosen to interpret the program, or the correct plan has been chosen and the student’s program has a bug. The present tutor cannot defer judgement⁷, thus it must disallow the kinds of variability discussed above.

Thus regardless of whether or not it is pedagogically desirable to decompose tasks step by step as GREATERP does, the debugging capabilities of GREATERP are limited since GREATERP cannot:

- Perform algorithm recognition.
- Infer the role of functions in algorithms.
- Infer the role of formal variables in functions.

2.3.4 Analysis of Loop Invariants

A formal approach to automatic program debugging, by Katz and Manna [Katz 76], extends the logical analysis of programs to include program incorrectness and a means of correcting incorrect programs. Program statements are related to synthesized inductive invariants. When these invariants are insufficient to establish a proof of correctness, program statements are altered so the necessary inductive

---

⁷GREATERP provides immediate feedback after each program token has been entered by the student in accord with the tutoring principles advocated in [Reiser 85].
invariants are derived. Synthesizing these inductive invariants and determining what program statements to alter is difficult; no implementation of their design exists to date.

Talus also attempts to debug programs by ensuring that a proof of correctness will succeed, however specifications in Talus are function definitions, not predicate calculus. Loop invariants are unnecessary, thus the problem of synthesizing invariants or altering code to derive necessary invariants does not arise. Furthermore, a logic and theorem prover capable of reasoning about these specifications already exists.

2.4 Comparison of Approaches

Static analysis and dynamic analysis are two competing approaches to bug detection that offer differing benefits and drawbacks. The main advantage of dynamic analysis over static analysis is its ability to rapidly localize errors without requiring a complete analysis of the student's programs. The main disadvantages are:

- Some stylistic bugs cannot be detected. (Discussed below.)
- Some stylistic bugs can only be detected by analyzing a very detailed program trace.
- Programs are only debugged with respect to the examples tried.
- Debugging programs with side effects is difficult since execution traces will depend not only on inputs but also program state.

The main advantages of static analysis over dynamic analysis are:

- Ability to detect stylistic errors that are difficult or impossible to detect with dynamic analysis.
- Ability to prove programs correct or incorrect with respect to task specifications.
- Ability to infer misconceptions by examining context surrounding a bug.
- No special treatment of side effects is necessary. This is true for approaches based on plan-based program analysis but not for those based on program verification. In the latter case it is difficult to formalize and reason about the semantics of programming language primitives that cause side effects.

The main disadvantage of static analysis is the necessity of interpreting the intentions of student code through plan recognition or other means. Program debugging can be impossible or incorrect if the interpretation is faulty.

The ability to detect stylistic errors in dynamic analysis is restricted. The following stylistic errors cannot be reliably detected with dynamic analysis:

- Unreachable code.
- Conditional tests that are always true or false.

Even if all previous program execution traces suggest the presence of these errors, further examples, not yet tried, can invalidate these assumptions. The following stylistic errors can be detected with dynamic analysis, but only if a very detailed execution history is analyzed:

- Inefficient Code. E.g. unnecessary CONSING, copying arrays unnecessarily, recomputing a constant value inside a loop, setting a variable to its current value.
- Violations of Rules of Programming Discourse. E.g. lack of modularity in code, "sphagetti code", using inappropriate programming constructs, nonmnemonic identifiers, inappropriate variable declarations, lack of appropriate data abstraction.
The stylistic errors above can be detected provided that crucial information, such as variable names and choice of programming constructs, is not compiled away. Detection of inefficient code may require metrics of time and space used and a trace of variable references. On the other hand, some stylistic errors, such as unnecessary procedure calls, can be detected from a simple trace of procedure invocations. Even though some stylistic errors can be detected with dynamic analysis, static analysis is preferable for a more thorough analysis of stylistic errors and for a reconstruction of underlying misconceptions through an examination of surrounding program context.

Considering approaches within static analysis, the main advantage program verification approaches have over plan-based approaches is the ability to prove implementations correct (or incorrect) without recourse to any plan or bug library. The main disadvantage is the difficulty of reasoning about the computational semantics of imperative programs in procedural programming languages, especially if loop invariants arise. Analysis by synthesis has the advantage that bug corrections can be synthesized and the tutor can explain its design process. Help can be given to students who cannot progress further by providing hints on program design and the actual code if necessary. However, heuristic plan recognition is more appropriate when it is difficult to synthesize code or the student's design decisions cannot be traced or easily inferred.

Heuristic plan recognition and all other approaches of plan-based program analysis can be viewed as systems that attempt to parse student programs using two separate, but related, grammars. The language of the first grammar is the set of all correct student programs for a task. The language of the second grammar is the set of all buggy programs for a task. Usually the second grammar is a superset of the first grammar obtained by adding bug rules, such as the bug rules in PROUST. In PROUST if a program cannot be parsed into either grammar through the use of plans and plan-difference rules then the program is not completely analyzed. In GREATERP if a program cannot be accounted for by GREATERP's rules to synthesize correct and buggy programs then it is disallowed. The student must continue with a correct implementation, or GREATERP will provide one.

These grammars for correct and buggy programs are explicit in the plan parsing approach but they are implicit in heuristic plan recognition and analysis by synthesis. In heuristic plan recognition correct student programs are parsed using plans and plan transformation rules. Buggy student programs can be parsed with the addition of bug rules. In analysis by synthesis a correct program is a program that can be synthesized using correct design rules only. Programs are considered buggy when additional bug rules must be invoked to account for the student's program [Reiser 85].

Although all plan recognition approaches can be modeled with grammars, the grammars implicit in heuristic plan recognition and analysis by synthesis have the following advantages over the explicit grammars in plan parsing approaches:

- **Nonterminals of the grammar correspond to student goals.** The nonterminal nodes of the grammars implicit in GREATERP [Reiser 85] and PROUST [Johnson 85] represent psychologically plausible plans corresponding to correct and incorrect design decisions that arise in a task's solution. This property of the grammars facilitates the inference of misconceptions (in both PROUST and GREATERP) and the explanation of intermediate design steps (in GREATERP). Debuggers based on plan parsing, such as [Ruth 73], tend to have less meaningful nonterminals that correspond to discrete program components rather

---

8The Ada Tutor designed by Miller [Miller 86] relies on dynamic analysis to detect stylistic errors of this kind.
than student goals.\textsuperscript{9}

- \textit{Parsing is simplified} by using plans in heuristic plan recognition or by tracing design decisions in analysis by synthesis. Plans can simplify the parsing of programs just as scripts can simplify the parsing of stories [Schank 77]. In analysis by synthesis the student's design can be traced since nonterminals correspond to task goals.

Returning to the overall classification hierarchy, note that debuggers are classified by \textit{primary means of bug detection}, thus actual systems may mix approaches. For example, Talus performs algorithm recognition using a plan library but detects bugs within functions using a program verification approach. PROUST uses analysis by synthesis to search for the best combination of nested matchings of plan templates to explain the student's code, but it does not synthesize Pascal code. SNIFFER records a program's execution history but also has a library of bug experts to detect specific bugs.

Debuggers performing symbolic evaluation are classified as performing static analysis rather than dynamic analysis since specific examples and a program interpreter or compiler are not required for symbolic evaluation. For example, [Laubsch 81] symbolically evaluates student programs to produce an effect-description that describes a program's side effects. The effect-description is matched to plans in a plan library. Since plan matching is the primary means of bug detection this approach belongs to the heuristic plan recognition category. Talus also symbolically evaluates student programs, but for a different purpose, to derive verification conditions. Since bugs are detected by failed verification conditions Talus uses a program verification debugging approach.

Within each approach in the taxonomy, those debuggers with greater capabilities to reason about computational semantics have greater debugging capabilities. Some examples:

- In dynamic analysis, PDS6 is more powerful than BIP since it has greater capabilities to reason about program execution traces.

- Considering heuristic plan recognition only, PROUST is more powerful than MENO-II since it can perform plan recognition at a finer level and has more facilities for accounting for code discrepancies with its plan transformation and bug rules.

- GREATERP is more powerful than TURTLE [Miller 82] since goals are represented explicitly and a forward chaining inference system allows GREATERP to reason about them.

\subsection{2.5 Theoretical Limitations}

The approaches compared above are all bounded by theoretical limitations. Assume that some program P performs perfect bug detection. Let our task specifications disregard program output and side effects, then P solves the halting problem since any nonterminating program is buggy, and any terminating program is correct. Thus no perfect bug detection method can exist, no matter what approach we use.

This theoretical limitation obscures more important performance questions. Even \textit{human} analysis fails when the human cannot comprehend the solution. For example, consider the difficulty of finding the bugs in a solution to a difficult task such as "Implement ANSI Standard FORTRAN in C". Thus we

\footnotesize{\textsuperscript{9}In contrast, SPADE [Miller 79] is a program analysis system based on plan parsing where nonterminal nodes of the grammar capture psychologically plausible task decompositions and design decisions. SPADE uses this grammar to provide a pedagogical environment that encourages students to be aware of design decisions that are normally implicit.}
should not ask "Will this debugger find all the bugs in student programs and never generate false alarms?" since the answer will always be no for nontrivial domains. Instead we should ask "What percentage of bugs does this debugger find, and how often does it generate false alarms?" Another pertinent question will be "How much variability can the debugger tolerate in student solutions without serious performance degradation?"

2.6 Summary

The key points of this chapter are:

• Static analysis and dynamic analysis are the two main approaches to automated debugging.

• Dynamic analysis is characterized by tracing program execution history; static analysis is characterized by analyzing student programs in terms of plans or formal specifications.

• PDS6, PROUST, and GREATERP are exemplars of quite different debugging approaches; still each is limited by restrictions in its ability to reason about computational semantics.

• Debuggers with greater abilities to reason about computational semantics have greater debugging capabilities:
  • More specific and accurate bug detection
  • Greater ability to correct errors
  • Greater tolerance for solution variability
  • Greater tolerance for larger and more complex tasks

• Perfect bug detection with no false alarms is impossible in nontrivial domains.
Chapter Three
An Overview of Talus

Chapter 2 pointed out limitations in existing automated debugging systems due to their inability to reason about computational semantics. This chapter addresses the question of how to use an ability to reason about computational semantics to debug recursive programs automatically. One approach is presented, that of Talus, an automated debugger for programs in LISP.

Before examining the debugging process of Talus the domain in which it operates is examined. Limitations on student solutions and the task knowledge that Talus starts off with are described. Then the specific kinds of bugs that Talus can detect and correct are enumerated. With this background, the debugging process of Talus is examined and then illustrated with several case studies.

3.1 The Domain
This section more precisely delimits the domain in which Talus operates and the knowledge which Talus has prior to debugging.

3.1.1 Acceptable Solutions
To simplify discussion of the debugging process three LISP dialects will be defined. Each is a proper subset of its successor, as illustrated in Figure 3-1. The dialects are:

- **The Pure Dialect.** The pure dialect contains the basic pure LISP functions for constructing pairs, lists, and numbers. Accessors and recognizers for these data objects are also provided, along with special forms required to define recursive functions that operate on these data structures.

- **The Core Dialect.** The core dialect adds to the pure dialect additional functions that perform side effects on shared list structure, property lists, and arrays.

- **The Extended Dialect.** The extended dialect extends the core dialect by adding special forms for mapping functions and imperative programming.

The basic debugging process illustrated in Chapter 1 operates on the pure dialect. This process must be extended by heuristics before the core dialect can be allowed. Then programs in the extended dialect can be debugged by simplifying them to the core dialect and debugging the simplified programs. Programs are simplified through the application of equivalence preserving program transformations; these are discussed in Section 3.3.2. The pure dialect is shown in Figure 3-2. Further discussion of the other dialects is deferred until the debugging extensions necessary for them are presented.

Student solutions are checked for syntactic errors before being analyzed. Syntactic errors are detected by Talus, but not corrected. Here are the syntactic errors Talus checks for:

- **Wrong Format in Special Form.** E.g. (COND (NULL L) NIL ...)

- **Wrong Number of Arguments.** Wrong number of arguments to a primitive, or inconsistent number of arguments in function calls or definitions of nonprimitives. E.g. (EQUAL (CAR L X)).

- **Binding Constants or Defining Primitive Functions.** E.g. T, F, and NIL cannot be used in variable lists; CAR cannot be redefined.
Figure 3-1: The LISP Dialects of Talus

- *Free Variables in Function Definitions.* Thus we avoid scoping issues.
- *Side Effects in Conditional Tests.* Disallowed since this would complicate symbolic evaluation (discussed in Section 3.3.2).
- *Side Effects in Actual Arguments of LAMBDA Expressions.* Disallowed since this would prevent LAMBDA's from being removed in program simplification transformations (discussed in Section 3.3.2).
- *References to Undefined Nonprimitive Functions.*

Talus also checks for some rational form violations [Goldstein 74]; these are violations of rules of programming discourse that can be recognized without knowledge of the assigned task. For example, Talus disallows functions that contain formal or local variables that are never referred to. By disallowing solutions with syntactic errors or violations of rational form, Talus simplifies its program analysis. An enormous amount of variability still remains, as will be seen in the case studies.

3.1.2 Tasks and Task Representations

Talus presently contains descriptions of 18 tasks; these are stored in a task library. The simpler tasks require the student to define functions that perform basic set operations or list manipulations, such as checking for set membership or reversing a list. More complicated tasks require several functions for
Special Forms: DEFUN, IF, QUOTE

List Operators: CONS, CAR, CDR, LIST, APPEND

Arithmetic Operators: ADD1, SUB1, PLUS, DIFFERENCE, TIMES, QUOTIENT

Boolean Connectives: AND, OR, NOT, IMPLIES

Predicates: ATOM, LISTP, NILP, ZEROP, NUMBERP, EQUAL, GREATERP, LESSP, =, NULL

Constants: T, NIL, F

Numbers: Natural Numbers Only (0, 1, 2...)

Notes:

- F evaluates to NIL.
- (LISTP NIL) = NIL.

Figure 3-2: The Pure Dialect

their solution or the use of side effects. Appendix III contains descriptions of the 18 task assignments in
the task library.

Each task has a task representation that includes the following information:

- The Task Assignment. Instructions to give the student.

- Algorithms. Identifiers naming acceptable algorithms for the solution of the task.

- Algorithm Representations. Frame representations of the algorithms above. The algorithm
  representation emphasizes abstract computational features to facilitate algorithm recognition
  in buggy student programs.

- Reference Functions. Functions that correctly implement the algorithm they are associated
  with and which are stylistically correct. These functions specify correct implementations and
  can be used to correct buggy implementations.

Note that algorithm representations support recognition of the student's algorithm while reference
functions allow specification of correct implementations of an identified algorithm.

3.2 The Debugging Capabilities of Talus

Within the domain just presented, Talus can detect many different kinds of nonsyntactic bugs, both
stylistic and nonstylistic, at three different levels of abstraction:

- The Algorithm Level. The design of the task solution.

- The Function Level. The functional decomposition of the chosen algorithm.

- The Implementation Level. The code that implements each function.
Bugs at the algorithm level reflect a misunderstanding of the task, or a fault in the student’s strategy for solving the task. Talus can detect when a student is using a buggy algorithm and provide commentary explaining why the design is faulty. Talus can also recognize and comment on algorithms that are correct but inefficient. Talus treats buggy algorithms as nonstylistic bugs while inefficient algorithms are considered merely stylistic bugs.

There are two kinds of bugs at the function level: extra functions and missing functions. Extra functions are superfluous functions not necessary to the implementation of the chosen algorithm. For example, a function CONCAT is superfluous if its definition is identical to APPEND because there is no need to reproduce the functionality of an existing primitive. Extra functions are stylistic bugs while missing functions are nonstylistic bugs. Missing functions are functions essential to the implementation of an algorithm that are not present in the student’s solution.

Common bugs at the implementation level include extra and missing conditional tests. Extra conditional tests are tests not strictly necessary for implementing a function. They can be subsumed by other tests. Extra conditionals are only stylistic bugs since the resulting function definition, even though more verbose, may still operate correctly. In contrast, missing conditional tests are always nonstylistic bugs since either nontermination, errors, or termination with incorrect values result.

Other common bugs occur in function terminations or recursions. Wrong termination bugs occur when a base case of a function returns the wrong value. Wrong recursion bugs occur if an incorrect recursive call is performed, or if the call is correct but the results are not correctly processed. Specific kinds of wrong recursion bugs are:

- **Wrong Variable Updates.** An actual parameter is incorrect in a function call.

- **Wrong Function Call.** The actual parameters are correct in a function call, but the wrong function is called. For instance, CONS is called instead of APPEND.

Talus can correct bugs in any of the nonstylistic bug categories above:

- **Buggy algorithms.** Having explained the design fault in the student’s algorithm Talus continues the debugging process assuming the student used the closest matching algorithm that correctly solves the task.

- **Missing Functions.** These are defined from the reference functions associated with the identified algorithm.

- **Missing Conditional Tests.** The necessary tests are extracted from reference functions and inserted into the student’s functions.

- **Wrong Termination/Recursion Bugs.** Code fragments from reference functions replace buggy code fragments in student code. Heuristics minimize the amount of code replaced. Notice that all bug corrections originate in reference functions. The following stylistic bugs are detected but not corrected:

  - **Inefficient Algorithms.**

  - **Extra Conditionals.** However, the recursions or terminations guarded by the conditionals are checked for bugs.

---

10This is a convenient oxymoron. The algorithm is "buggy" with regard to the assigned task, but correctly solves some other task that is usually a misreading of the one assigned.
3.3 An Overview of the Debugging Process

With this understanding of the domain limitations and capabilities of Talus, its debugging process can be explained. Debugging, as shown in Figure 3-3, is a four stage process:

- **Program Simplification.** The student’s functions are simplified, by transformations that preserve computational equivalence, into a form and dialect that is more suitable for analysis.

- **Algorithm Recognition.** The simplified functions are parsed into frames; these are matched to frames in the task representation. This partial matching process not only identifies the student’s algorithm but also pairs reference functions with student functions.

- **Bug Detection.** For each function pair, Talus performs the case splitting that would occur in an inductive proof of equivalence. Verification conditions that establish the base case and induction steps are generated.

- **Bug Correction.** Violated verification conditions are repaired by altering student functions; reference code supplies the corrections. These corrections are traced back through transformations to the student’s original code.

Each step is examined in more detail later in this section, after a small example has been presented.

![Figure 3-3: An Overview of Debugging in Talus](image-url)
3.3.1 A Simple Example

A simple example will help orient the reader to the overall debugging process. This example expands on the MEMTREE example first seen in Section 1.5.2. Only terse commentary on the debugging is provided here since a more detailed explanation will be provided in Section 3.4.1, after the reader has a more thorough understanding of the debugging process. Some details have been suppressed. The task and the student's solution are repeated below:

**Task - MEMTREE**

Write a function that determines whether an atom is one of the leaves of a tree.

(DEFUN MEMTR (AT CONS)
  (IN AT (FLAT NIL CONS)))

(DEFUN FLAT (ANS TR)
  (IF (ATOM TR) ANS
      (FLAT (FLAT ANS (CDR TR))
            (CAR TR))))

(DEFUN IN (X L)
  (IF (LISTP L)
      (IF (EQUAL L (LIST X))
          L
          (IF (NOT (EQUAL (CAR L) X))
              (IN X (CAR L))
              L))
      NIL))

No program simplification is required since Talus can analyze the functions in their present form. The results of the next step, algorithm recognition, are:

Algorithm used: MEMTREE-FLATTEN.

<table>
<thead>
<tr>
<th>Student Fns</th>
<th>Matched to Reference Fns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAT</td>
<td>to FLATTEN</td>
</tr>
<tr>
<td>IN</td>
<td>to MEMBER</td>
</tr>
<tr>
<td>MEMTR</td>
<td>to MEMTREE</td>
</tr>
</tbody>
</table>

Mapping of Formal Variables:

| FLAT to MCFLATTEN: (TR/TREE, ANS/ANSWER). |
| IN to MEMBER: (L/BAG, X/ITEM). |
| MEMTR to MEMTREE: (CONS/TREE, AT/ITEM). |

Now Talus can compare each student function to its paired reference function to see if they are equivalent. These comparisons require inductive proofs. The following verification conditions check the validity of these proofs:
Verification Conditions for FLAT

; First Case: (ATOM TR)

[Bug found:

(IMPLIES (ATOM TR) ; Case
       (EQUAL (CONS TR ANS) ; From Stored Fn
                ANS)) ; From Student Fn

is invalid.]

HINT: Looks like you used the variable ANS instead of the function call (CONS TR ANS) in FLAT.

; Second Case: (NOT (ATOM TR))

[Check:

(IMPLIES (NOT (ATOM TR))
       (EQUAL (FLAT (FLAT ANS (CDR TR))
                (CAR TR))
                (FLAT (FLAT ANS (CDR TR))
                (CAR TR))))

is a theorem.]

Verification Conditions for IN

; First Case: (NLISTP L)

[Check:

(IMPLIES (NLISTP L) (IFF F NIL))

is a theorem.]

; Second Case:
; (AND (NOT (NLISTP L))
;     (EQUAL X (CAR L)))

; Forced Assumption: (EQUAL L (LIST X))

[Check:

(IMPLIES (AND (NOT (NLISTP L))
              (EQUAL X (CAR L))
              (EQUAL L (LIST X))
              (IFF T L))

is a theorem.]
;Second Case:
; (AND (NOT (NLISTP L))
; (EQUAL X (CAR L)))))

;Forced Assumption: (NOT (EQUAL L (LIST X)))

[Check:

(IMPLIES (AND (NOT (NLISTP L))
 (EQUAL X (CAR L))
 (NOT (EQUAL L (LIST X))))
 (IFF T L))

is a theorem.]

;Third Case:
; (AND (NOT (NLISTP L))
; (NOT (EQUAL X (CAR L))))

[Bug found:

(IMPLIES (AND (NOT (NLISTP L))
 (NOT (EQUAL X (CAR L))))
 (IFF (IN X (CDR L))
 (IN X (CAR L))))

is invalid.]

HINT: Looks like you called the function CAR instead of function CDR in IN.

Verification Conditions for MEMTR

;First and Only Case: T

[Check:

(IMPLIES T (IFF (IN AT (FLAT NIL CONS))
 (IN AT (FLAT NIL CONS))))

is a theorem.]

Reference code is used to correct bugs in the student code. The final debugged code is:

(DEFUN MEMTR (AT CONS)
 (IN AT (FLAT NIL CONS)))

(DEFUN FLAT (ANS TR)
 (IF (ATOM TR)
  (CONS TR ANS)
  (FLAT (FLAT ANS (CDR TR))
   (CAR TR)))))

(DEFUN IN (X L)
(IF (LISTP L)
  (IF (EQUAL L (LIST X))
    L
    (IF (NOT (EQUAL (CAR L) X))
      (IN X (CDR L))
      L)))
NIL)

Keeping this example in mind, the reader can better understand the more detailed examination of program debugging presented below. Details of algorithm recognition and verification condition generation that may be unclear now will be explained. Recall that debugging proceeds in four stages: program simplification, algorithm recognition, bug detection, and bug correction. These will be explained in turn, starting with program simplification.

3.3.2 Program Simplification

Program simplification facilitates program debugging by reducing the number of constructs that must be reasoned about and by representing programs in a form amenable to symbolic evaluation. The first aim is achieved by rewriting programs in the extended dialect into computationally equivalent programs in either the core dialect or the pure dialect. The second aim is achieved by rewriting conditionals into IF-Normal Form. In IF-Normal Form:

- The only conditional expression is IF.
- No IF expression occurs as part of the test of another IF expression.
- No IF expression occurs inside a function call.

The solution to the MEMTREE task just seen was already in IF-Normal Form. However if COND had been used in place of IF, program simplification would have been necessary.

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. Figure 3-4 shows the binary tree representation of the function IN that occurs in the MEMTREE example. Symbolic evaluation consists of determining a path in the tree from the root to a terminal node. This representation also facilitates the detection of missing conditionals. A more detailed discussion appears in Chapter 6.

Program simplification is accomplished by means of program simplification transformations that preserve computational equivalence. Most transformations are simple rewrites, for example, a function with COND can easily be reexpressed with IF. Some are not simple rewrites but algorithmic in nature such as the transformation of a function with PROG into a set of recursive functions without PROG.

Of course these transformations are not much good if the results of debugging the simplified programs cannot be related to the student's original programs. Consequently each transformation is represented by two graphs, one graph for the program before and one graph for the program after the transformation. Of critical importance are links between the graphs that tie together s-expressions copied and transformed in the program transformation. It is only by following these links, and by taking into account the semantics of individual program transformations, that Talus can trace edits back to the original student code. This tracing capability allows Talus to identify and correct bugs in the context of the student's original code. An ITS that uses Talus needs this contextual information to infer misconceptions and to generate appropriate hints.
3.3.3 Algorithm Recognition

All functions, stored or student, are parsed into E-frames. Stored algorithms from the task representation and student solutions are both represented as collections of E-frames. E-Frame slots represent abstract properties of recursive functions that (partially) enumerate the elements of a recursively defined data structure. A function’s E-frame has slots representing its recursion type (tree, list, or number), recursive calls, terminations, variable updates, and task role (main, constructor, or predicate). The E-frame representation facilitates a robust algorithm recognition 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. E-frames are discussed in detail in Chapter 4. Chapter 5 examines the entire algorithm recognition process in depth; only an introduction is provided here.

Figure 3-5 provides an overview of how Talus recognizes algorithms. The simplified student functions and stored algorithms are the input to a partial matching process. Talus maps reference functions associated with algorithms to the student functions. Talus identifies the student’s intended algorithm as the stored algorithm whose reference functions occur in the best function mapping found. Thus the algorithm recognition process not only identifies the student’s algorithm but also pairs student functions to reference functions. Using these function pairs, bug detection and correction can be performed in a similar manner to that presented in Chapter 1.

Talus performs a best first search to choose between competing algorithms and to map student functions to reference functions. Nodes are partial mappings of student and reference functions for one of the competing algorithms. Function mappings allow for missing or superfluous student functions while using constraints to reduce the search space. Student functions map to reference functions or to EXTRA;
reference functions map to student functions or to EXTRA.\textsuperscript{11} Two functions can be paired only if their parents have already been paired and the functions have the same task role.

A measure of dissimilarity is computed for each partial mapping. Each function pair contributes a penalty that is a weighted sum of the differences between the slots of the corresponding E-frames. Additional penalties are added for functions mapped to EXTRA.

Alternative functional decompositions of algorithms are represented either extensionally as additional algorithm representations or intensionally through the use of global solution transforms. A global solution transform rewrites the reference functions of an algorithm to computationally equivalent functions that more closely match the student’s functions. These transforms account for common algorithm variants. For example, if a reference function is \textit{not} tail recursive and it has been paired with a student function that is, then that reference function will be transformed to an equivalent tail recursive implementation.

For each student function paired to a reference function, Talus applies heuristics to determine a mapping between the formal variables of the functions. The heuristics pair formal variables with similar function roles and data types. Information on the role and data type of formal variables is explicitly represented in function E-frames.

\textsuperscript{11}Student functions mapped to EXTRA are considered superfluous to a correct solution while reference functions mapped to EXTRA are considered essential to a correct solution but missing from the student’s solution.
Talus now rewrites the reference functions of the best function mapping to use the student’s identifiers in place of those assumed in the task representation. The function pairings in the best function mapping allow reference function names to be replaced by student function names; the formal variable pairings between individual function pairs allow reference variable names to be replaced by student variable names.

This customization of the reference functions to use the student’s identifiers is the normalization process briefly alluded to in Chapter 1. Intuitively, it facilitates a direct comparison of reference functions and student functions. Formally, there will be no difference between the verification conditions produced in an inductive proof of equivalence between a normalized reference function and its paired student function and the same proof with the original reference function. By applying inductive hypotheses, the verification conditions produced with the original reference function can be rewritten to be exactly the same as the verification conditions produced with the normalized reference function.

3.3.4 Bug Detection

Figure 3-6 illustrates how Talus debugs a student function matched to a reference function when both functions are in 1F-Normal Form. As described earlier, a binary tree can represent each function, with nonterminal nodes representing conditional tests and terminal nodes representing function terminations or recursions (i.e. recursive calls). The terms that must be true or false to reach a terminal node are the terms governing that node. Each set of terms governing a terminal node is a case.

![Diagram of Bug Detection in Talus](image-url)

Figure 3-6: Bug Detection in Talus
For each case, Talus symbolically evaluates both the student and the reference function. Functions in IF-Normal Form are symbolically evaluated by rewriting function bodies to symbolic values, these are s-expressions that contain no IF expressions. Formal methods can determine if a case implies that a conditional test is true or false. If a conditional test is true then the surrounding IF expression is rewritten to its first branch; if the conditional test is false it is rewritten to its second branch. If neither situation holds, then case splitting occurs and more than one symbolic value is returned. Section 3.4.3 gives an example of case splitting during symbolic evaluation.

Symbolic evaluation can be viewed as determining a path from the root node of a function’s binary tree representation to one or more leaves. For example, in Figure 3-4, IN can be symbolically evaluated to \((\text{IN} \ X \ (\text{CAR} \ L))\) for the case
\[(\text{AND} \ (\text{NOT} \ (\text{ATOM} \ L)) \ (\text{NOT} \ (\text{EQUAL} \ X \ (\text{CAR} \ L))))\]
since this term implies that \((\text{LISTP} \ L)\) is true, the term \((\text{EQUAL} \ L \ (\text{LIST} \ X))\) is false, and the term \((\text{NOT} \ (\text{EQUAL} \ (\text{CAR} \ L) \ X))\) is true.

Extra and missing conditional tests are detected by case splitting in the symbolic evaluation process. When cases from the reference function are used to symbolically evaluate the student’s function, case splitting indicates extra conditional tests in the student’s function. When cases from the student function are used to symbolically evaluate the reference function, case splitting indicates missing conditional tests in the student’s function. (The correction of missing conditional tests is described in Section 3.3.5).

After correcting any missing conditional tests, Talus compares the symbolic values of the student and reference functions. This comparison mirrors an inductive proof of equivalence between the two functions. The induction proof case splits first on the reference cases and then on any extra conditional tests in the student’s code. For each case Talus symbolically evaluates both functions and compares the resulting symbolic values for equivalence. The conjectures that Talus generates to check for equivalence are called functional equivalence verification conditions.

Conjectures generated for the base cases of the student function correspond to the base cases of the induction proof. There are always base cases for the student function since any missing conditionals have already been added at this point. Conjectures generated for the remaining cases correspond to the induction steps of the proof.

Talus also generates termination verification conditions. Associated with each reference function are a measure and well-founded relation that establish the function’s termination. The measure and well-founded relation are part of the task representation. The termination verification conditions check that the same measure and well-founded relation apply to the arguments of all recursive calls in the student’s function. These conjectures correspond to showing that induction steps will always cause some measure that is subject to a well-founded relation to decrease in the induction proof; thus the base case of the proof must always be reached.

All conjectures are well formed formulas in the Boyer-Moore Logic. The Boyer-Moore Theorem Prover can check these formulas once the normalized reference functions have been defined. Other means of conjecture evaluation are discussed in Chapter 6.
3.3.5 Bug Correction

The final step in the debugging process is bug correction. Bug detection and bug correction are actually interleaved since missing conditional tests are first inserted into the student’s function prior to the generation of verification conditions. The missing conditional tests to insert into a student’s function are obtained from the reference function. Details on how the test is extracted from the reference function, and how it is inserted into the buggy student function are deferred until Chapter 6.

Corrections to enforce verification conditions also originate in normalized reference functions. Recall that functional equivalence verification conditions compare two code fragments; these are symbolic values from the student and reference functions. When a functional equivalence verification condition fails, it is enforced, i.e. made true, by replacing all or part of the student’s code fragment with parts of the reference code fragment. The replacements occur in both the verification condition and the student’s program.

Termination verification conditions are similarly enforced. If the arguments to a recursive call cannot be shown to diminish by the expected measure and well-founded relation, then reference code will replace student code. The student code replaced is the symbolic value containing the buggy recursive call. The correction is the symbolic value in the reference function reached under the same case as the student’s code.

Corrections to both functional equivalence and termination verification conditions ensure that the inductive proof of equivalence succeeds and any bugs in the student’s program are repaired. Heuristics attempt to isolate bugs to individual s-expressions or symbols in student code so that the extent of repair is minimized.

As long as all conjectures are correctly evaluated, all bugs in the student’s program will be fixed. The proof of correctness is implicit in the debugging process. In actuality the debugging process is imperfect since there is no decision procedure for the conjectures and the heuristics in the algorithm recognition process can fail. Empirical evaluation, discussed in Chapter 9, demonstrates that bug recognition is high and false alarms are low in spite of these imperfections.

3.4 Case Studies

To help clarify the debugging process, several case studies will be presented. Each case study will emphasize different aspects of debugging:

- **Basic Algorithm Recognition, Bug Detection, and Bug Correction** in the MEMTREE case study.
- **Identification of Buggy Algorithms; Handling Missing Functions** in the SINGLETONS case study.
- **Detection and Correction of Missing Conditions; Failed Termination Verification Conditions** in the MEMBER case study.
- **Simplification of Mapping Functions and Imperative Programs; Edit Inversion** in the PALINDROMES case study.
- **Heuristics to Detect Bugs in Functions with Side Effects** in the REVERSE case study.

Several case studies are necessary since no single example can illustrate all these features. In subsequent chapters further details of these case studies will be examined. Each case study focuses on some particular part of the debugging process; consequently unrelated details are suppressed.
3.4.1 MEMTREE

This case study further elaborates on the MEMTREE example shown in Section 3.3.1 to show how Talus debugs a solution containing multiple functions, multiple formal variables, and multiple bugs. Algorithm recognition is required since the assigned task can be solved by different algorithms. The scenario below is edited for brevity. The scenario starts in the middle of a tutorial session, after the student has had some instruction in LISP programming. The task and solution are the same as in Section 3.3.1; they are repeated below.

**Task - MEMTREE**

Write a function that determines whether an atom is one of the leaves of a tree.

```lisp
(DEFUN MEMTR (AT CONS)  
  (IN AT (FLAT NIL CONS)))
(DEFUN FLAT (ANS TR)  
  (IF (ATOM TR) ANS  
    (FLAT (FLAT ANS (CDR TR))  
      (CAR TR))))
(DEFUN IN (X L)  
  (IF (LISTP L)  
    (IF (EQUAL L (LIST X))  
      L  
      (IF (NOT (EQUAL (CAR L) X))  
        (IN X (CAR L))  
        L))  
      NIL))
```

Figure 3-7: A Buggy Solution to the MEMTREE Task

No program simplification is necessary since the student's program is already in IF-Normal Form. Subsequent case studies illustrate programs that require considerable simplification to be placed into this form.

3.4.1.1 Algorithm Recognition

Talus must now recognize the algorithm the student has used. First, the functions MEMTR, FLAT, and IN are parsed into E-frames. The three E-frames together represent the student's solution.

Talus knows of two different algorithms for the MEMTREE task. The TREE-WALK algorithm explores the CAR and the CDR of a tree separately to see if an atom is in the tree. The MEMTREE-FLATTEN algorithm first flattens the tree and then determines if the atom is a member of the resulting bag. The result of the algorithm recognition process is:

Algorithm used: MEMTREE-FLATTEN.
Student Fns Matched to Reference Fns:
  FLAT to FLATTEN
  IN to MEMBER
  MEMTR to MEMTREE

Solution Transform Applied:
  Transforming FLATTEN to MCFLATTEN to
  better match the student function FLAT.

Talus selects the MEMTREE-FLATTEN algorithm as being more similar to the student’s solution
than the TREE-WALK algorithm. The reference functions FLATTEN, MEMBER, and MEMTREE,
whose E-frames comprise the MEMTREE algorithm, are mapped to the student functions FLAT, IN, and
MEMTR.

The reference functions associated with the identified algorithm are referred to as the stored
solution. Reference functions will also be referred to as stored functions when it is desirable to emphasize
that they are prestored prior to debugging as part of the task representation.

As mentioned previously, Talus has stored global solution transforms that allow it to transform its
stored solution to an equivalent solution that more closely matches the student’s solution. Thus, when
appropriate, MCFLATTEN replaces FLATTEN, and calls to MCFLATTEN replace calls to FLATTEN.
A similar transform allows predicates and predicate calls in stored solutions to be simultaneously
logically inverted.

At this point, Talus maps the formal variables of matched functions by using heuristics that take
into account variable data type:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAT</td>
<td>MCFLATTEN: (TR/TREE, ANS/ANSWER)</td>
</tr>
<tr>
<td>IN</td>
<td>MEMBER: (L/BAG, X/ITEM)</td>
</tr>
<tr>
<td>MEMTR</td>
<td>MEMTREE: (CONS/TREE, AT/ITEM)</td>
</tr>
</tbody>
</table>

3.4.1.2 Bug Detection

Talus now debugs the student functions by comparing them to the stored functions they have been
matched with. Talus matched FLAT to FLATTEN and then transformed FLATTEN to MCFLATTEN to
match FLAT better. The stored function MCFLATTEN is defined as:

```
(DEFUN MCFLATTEN (TREE ANSWER)
  (IF (ATOM TREE)
      (CONS TREE ANSWER)
      (MCFLATTEN (CAR TREE)
           (MCFLATTEN (CDR TREE)
              ANSWER))))
```

In order to facilitate bug detection and allow Talus to replace buggy student code with stored code
to correct bugs, the code above is normalized by replacing the stored function and formal variable names
with the student’s, and then permuting the formal variable order to match the student’s. The result is:

```
(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
      (CONS TR ANS)
      (FLAT (FLAT ANS (CDR TR))
           (CAR TR))))
```
which will be compared to the student’s definition of FLAT, which is repeated here:

(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
      ANS
      (FLAT (FLAT ANS (CDR TR))
            (CAR TR)))))

Talus checks for missing and extra conditionals in the student’s definition and finds none. There are two cases in the reference code: either (ATOM TR) is true or (NOT (ATOM TR)) is true. By comparing the student’s function and stored functions for these two cases, we can determine if they compute the same values under the same conditions. If they do not then a bug is present. The functional equivalence verification conditions generated for these cases, and their evaluation, are shown below:

[BUG found:

(IMPLIES (ATOM TR) ; Case
  (EQUAL (CONS TR ANS) ; From Stored Fn
    ANS)) ; From Student Fn

is invalid.]

HINT: Looks like you used the variable ANS instead of the function call (CONS TR ANS) in FLAT.

[Check:

(IMPLIES (NOT (ATOM TR)))
  (EQUAL (FLAT (FLAT ANS (CDR TR))
               (CAR TR))
          (FLAT (FLAT ANS (CDR TR))
                (CAR TR))))

is a theorem.]

A conjecture is first checked by a conjecture disprover that searches stored examples for a counterexample to the conjecture. The stored examples are sets of bindings of formal variables for each function in a stored task algorithm. They are part of the task representation. If the conjecture evaluates true for all stored examples then it is believed, otherwise it is definitely false and one of the stored examples provides a counterexample to the conjecture.

Conjectures that are believed are then passed to the Boyer-Moore Theorem Prover for formal verification. Functions involved in the conjectures are previously defined using the normalized stored function definitions. If a conjecture is formally proved then no bug is present in the student’s code for that case. If the proof of a believed conjecture fails, then either the conjecture is false or necessary lemmas for the proof to succeed are missing. In this particular case study all conjectures that are believed are proven to be theorems by the Boyer-Moore Theorem Prover.

For more complex examples, proofs may fail due to the absence of necessary lemmas. When this happens correct implementations are considered buggy and replaced by stored code fragments. With this approach, buggy implementations are always detected.
A more practical but less elegant approach is to accept as true the conjectures believed by the conjecture disprover. More complex programs can be debugged since the conjecture disprover needs no lemmas, but some bugs may be missed if no counterexample is found to an invalid conjecture. On the other hand, correct implementations are never considered buggy, and true conjectures that are difficult to prove formally are easily checked by the conjecture disprover.

3.4.1.3 Bug Correction

When a conjecture is invalid, Talus debugs the student's code by altering the student's code so that the conjecture becomes a theorem. Essentially, Talus attempts to verify the student's program using the stored function both as its specification and as a source of corrections. Debugging consists of enforcing the verification conditions when necessary. In the MEMTREE case study, since the student and stored functions are not always equal when (ATOM TR) is true, a bug is present. Talus fixes the student's code by replacing only the student's code fragment for this case with the corresponding stored code fragment. The debugged code is shown below:

(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
      (CONS TR ANS)
      (FLAT (FLAT ANS (CDR TR))
            (CAR TR))))

The student's definition IN is debugged similarly by comparing it to the stored function MEMBER, which is normalized to:

(DEFUN IN (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (IN X (CDR L))))))

This definition is compared to the student's definition, which is repeated below:

(DEFUN IN (X L)
  (IF (LISTP L)
      (IF (EQUAL L (LIST X))
          L
          (IF (NOT (EQUAL (CAR L) X))
              (IN X (CAR L))
              L))
      NIL))

Talus generates the following functional equivalence verification conditions to check whether the student and stored functions are logically equivalent predicates:

(IMPLIES (NLISTP L) (IFF F NIL))

(IMPLIES (AND (NOT (NLISTP L))
               (EQUAL X (CAR L))
               (EQUAL L (LIST X)))
          (IFF T L))
\begin{verbatim}
(IMPLIES (AND (NOT (NLISTP L))
   (EQUAL X (CAR L))
   (NOT (EQUAL L (LIST X))))
   (IFF T L))

(IMPLIES (AND (NOT (NLISTP L)))
   (NOT (EQUAL X (CAR L))))
   (IFF (IN X (CDR L))
   (IN X (CAR L))))

The first three conjectures are theorems while the last is not, indicating a bug which Talus corrects:

(DEFUN IN (X L)
   (IF (LISTP L)
      (IF (EQUAL L (LIST X))
         L
         (IF (NOT (EQUAL (CAR L) X))
         (IN X (CDR L))
         L))
      NIL))

The remaining function, MEMTR, has no bugs and its analysis is omitted.

3.4.2 SINGLETONS Case Study

In the last case study the student had a buggy implementation of a correct algorithm. In this scenario, the student has a buggy implementation of a buggy algorithm. In other words, the student misunderstood the task and used an algorithm that would not solve the stated task, even if the implementation had been correct. When this happens Talus explains why the intended algorithm is incorrect and then proceeds to debug the student's solution using the closest matching correct algorithm. Here is the task and the student's solution:

Design a function (SINGLETONS X) which takes as argument an s-expression X and produces as a result a list of all atoms other than NIL which occur exactly once in X. For example,

<table>
<thead>
<tr>
<th>SINGLETONS of NIL</th>
<th>is NIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLETONS of (A)</td>
<td>is (A)</td>
</tr>
<tr>
<td>SINGLETONS of (A (B (E) B (C D)) E C)</td>
<td>is (A D) or (D A)</td>
</tr>
</tbody>
</table>

Use a function (FLATTEN X), where the argument X is an s-expression with subexpressions nested to any depth, such that the result of (FLATTEN X) is just a list of atoms with the property that all atoms other than NIL appearing in X also appear in (FLATTEN X).

(DEFUN SINGLETONS1 (X)
   (COND ((NULL X) NIL)
         ((ATOM X) (LIST X))
         (T (SINGFLAT1 (SMASH X)))))
\end{verbatim}
(DEFUN SMASH (S)
  (COND ((NULL S) NIL)
    ((ATOM S) (CONS S NIL))
    (T (CONS (SMASH (CAR S)) (SMASH (CDR S)))))))

(DEFUN SINGFLAT1 (X)
  (COND ((ABSENT (CDR X) (CAR X))
    (APPEND (SINGFLAT1 (CDR X)) (LIST (CAR X))))
    (T (SINGFLAT1 (CDR X))))))

(DEFUN ABSENT (L A)
  (COND ((NULL L) T)
    ((EQUAL A (CAR L)) NIL)
    (T (ABSENT (CAR L) A))))

When a precisely described task is frequently misinterpreted by students due to careless reading or misunderstandings it is advantageous to store buggy algorithms along with correct algorithms as part of the task representation. Such is the case here. The buggy algorithms are only used to identify misunderstandings at the algorithm level; all function debugging must be done with correct algorithms.

Buggy algorithms are added to the pool of possible algorithms that Talus can recognize for a task. If the algorithm that most closely matches the student's is a buggy algorithm and the match is clearly better than any correct algorithm, Talus proceeds with the assumption that the student intended to implement the buggy algorithm. Program commentary is generated explaining the misunderstanding, and then debugging proceeds with the closest matching correct algorithm.

Part of the Talus output is program commentary. If Talus has recognized the student's algorithm as one of the correct algorithms it knows about then the commentary describes the role of the functions and formal variables in the algorithm's implementation. If Talus has recognized the student's algorithm as one of the buggy algorithms it knows about, as happens in this example, the commentary describes how the algorithm is incorrect, and attempts to clarify the student's misunderstanding of the task. The commentary provided for this solution is:

**Looks like you misunderstood the task.** The function SINGTONS1 flattens its input and passes the resulting bag to the function SINGFLAT1. The function SINGFLAT1 unifies its input. The result is that one copy of every atom in the input to SINGTONS1 will be returned. This is not the same as returning only those atoms that occur exactly once.

For example,

(SINGFLAT1 '(A B A C C E)) should be (B E), not (A B C E).

**NOTE:** I will continue the debugging process using the SINGTONS-W-MEMBER algorithm, which is the correct algorithm that most closely matches your algorithm.

The commentary is easily generated by substituting the student's identifiers in place of stored identifiers in a template associated with each algorithm. Since identifiers must be paired prior to this
process, commentary generation occurs after all function and formal variable mapping. The commentary templates are part of the task representation.

Talus proceeds to debug the code assuming that the student intended the closest matching algorithm (SINGLETONS-W-MEMBER). A predicate negation solution transform is applied to the stored solution so that its definition of MEMBER is logically inverted; this results in a closer match to the ABSENT function. After normalization, generation of verification conditions and debugging occur just as in the MEMTREE case study. Here is the debugged code which Talus provides, which includes a definition of the missing function REMOVE that is necessary for the SINGLETONS-W-MEMBER algorithm:

```lisp
(DEFUN SINGLETONS1 (X)
  (COND ((NULL X) NIL)
         ((ATOM X) (LIST X))
         (T (SINGFLAT1 (SMASH X)))))

(DEFUN SMASH (S)
  (COND ((NULL S) NIL)
         ((ATOM S) (CONS S NIL))
         (T (APPEND (SMASH (CAR S)) (SMASH (CDR S))))))

(DEFUN SINGFLAT1 (X)
  (COND ((NILSP X) NIL)
         ((ABSENT (CDR X) (CAR X))
          (APPEND (SINGFLAT1 (CDR X)) (LIST (CAR X)))
          (T (SINGFLAT1 (REMOVE (CAR X) (CDR X))))))

(DEFUN ABSENT (L A)
  (COND ((NULL L) T)
         ((EQUAL A (CAR L)) NIL)
         (T (ABSENT (CDR L) A))))

Missing Definition for REMOVE.

(DEFUN REMOVE (ITEM BAG)
  (COND ((NILSP BAG) NIL)
         ((EQUAL ITEM (CAR BAG))
          (REMOVE ITEM (CDR BAG)))
         (T (CONS (CAR BAG)
                  (REMOVE ITEM (CDR BAG))))))
```

Note that Talus detected bugs at all three levels of abstraction. The student used a buggy algorithm that Talus identifies. Talus allows debugging to continue by assuming that the student really intended the closest matching correct algorithm. However, with this assumption the student's functional decomposition is incorrect since a definition for REMOVE was not supplied. Talus supplies the missing function definition using the reference function REMOVE. Finally, in debugging the student's functions, Talus discovers numerous bugs at the implementation level, such as the use of CONS instead of APPEND in the function SMASH.

Talus also detected and corrected a missing conditional in SINGFLAT1. The MEMBER case study presents a more detailed examination of the detection and correction of missing conditionals.
3.4.3 MEMBER Case Study

We consider the MEMBER task to study:

- Detection and Correction of Missing Conditionals
- Detection and Correction of Nontermination

The task assignment and the first solution we will consider is:

```
Write a function that determines whether an item is in a proper list. You need only examine the list at the top level.
```

```
(DEFUN MEM (X L)
  (IF (LISTP L)
      (MEM X (CDR L))
      NIL))
```

In this solution the student forgot to include the check to see whether X is the first element of L.

Talus will detect missing conditionals by comparing its reference function MEMBER to MEM. The missing conditional to be inserted will also come from the reference function. The unnormalized reference function MEMBER is defined as:

```
(DEFUN MEMBER (ITEM BAG)
  (IF (NLISTP BAG)
      F
      (IF (EQUAL ITEM (CAR BAG))
          T
          (MEMBER ITEM (CDR BAG)))))
```

After normalization the reference function appears as:

```
(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (MEM X (CDR L)))))
```

Recall that extra and missing conditional tests are detected by case splitting in the symbolic evaluation process. When cases from the student function are used to symbolically evaluate the reference function, case splitting indicates missing conditional tests in the student’s function. In this example, there are two cases to consider in the student’s code: either (LISTP L) is true or false. When Talus assumes that (LISTP L) is false, the reference function can be symbolically evaluated to F as shown in Figure 3-8.

When Talus assumes that (LISTP L) is true, case splitting is necessary to symbolically evaluate the reference function, as shown in Figure 3-9. The necessity of case splitting when the predicate (EQUAL X (CAR L)) is encountered in the reference function indicates that there is no conditional test logically equivalent to (EQUAL X (CAR L)) in the student’s function.

Talus corrects the error by inserting the predicate (EQUAL X (CAR L)) from the normalized reference function into the student’s code. The insertion point is above the code reached when (LISTP L)
Case: (NOT (LISTP L))

\( (\text{NLISTP } L) \)

\( (\text{EQUAL } X \ (\text{CAR } L)) \)

\( F \)

\( (\text{MEM } X \ (\text{CDR } L)) \)

\( T \)

Theorem Used in Symbolic Evaluation:

\( (\text{IMPLIEDS} \ (\text{NOT} \ (\text{LISTP} \ L)) \ (\text{NLISTP} \ L)) \)

Figure 3-8: Symbolic Evaluation of MEM

Case Splitting Required for:

CASE: (LISTP L)

\( (\text{EQUAL } X \ (\text{CAR } L)) \)

\( (\text{EQUAL } X \ (\text{CAR } L)) \)

\( F \)

\( (\text{MEM } X \ (\text{CDR } L)) \)

\( T \)

Theorem Used in Symbolic Evaluation:

\( (\text{IMPLIEDS} \ (\text{LISTP} \ L) \ (\text{NOT} \ (\text{NLISTP} \ L))) \)

Figure 3-9: Symbolic Evaluation of MEM

is true, i.e. immediately above the s-expression (MEM X (CDR L)). That s-expression is copied to both branches under the new conditional test, the result is:
(DEFUN MEM (X L)
  (IF (LISTP L)
        (IF (EQUAL X (CAR L)) ; Inserted Conditional
            (MEM X (CDR L))
            (MEM X (CDR L)))
        NIL))

Note that the reference code could be used to correct the student's code since it was normalized so that variable references to ITEM and BAG were replaced by references to X and L.

Talus generates verification conditions as before; the final debugged code is:

(DEFUN MEM (X L)
  (IF (LISTP L)
        (IF (EQUAL X (CAR L))
            (MEM X (CDR L))
            NIL))

As an extreme example, and to illustrate the detection and correction of possible infinite loops, consider the following solution to MEMBER:

(DEFUN MEMB (ELEMENT SET) (MEMB ELEMENT SET))

Talus first detects the missing conditional tests and then inserts them. The result is

(DEFUN MEMB (ELEMENT SET)
  (IF (NILISTP SET)
      (MEMB ELEMENT SET)
      (IF (EQUAL ELEMENT (CAR SET))
          (MEMB ELEMENT SET)
          (MEMB ELEMENT SET))))

Debugging proceeds as usual with the following verification conditions being generated:

Case 1: (NILISTP SET)

Function Equivalence Verification Condition

(IMPLIES (NILISTP SET)
          (IFF P (MEMB ELEMENT SET)))

Termination Verification Condition

(IMPLIES (NILISTP SET)
          (LESSP (COUNT SET) (COUNT SET)))
Case 2: \[ \text{AND} \ (\text{NOT} \ (\text{NLISTP} \ \text{SET})) \]
\[ \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]

*Function Equivalence Verification Condition*

\[ \text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ \text{SET}))) \]
\[ \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]
\[ \ (\text{IFF} \ T \ (\text{MEMB} \ \text{ELEMENT} \ \text{SET})) \]

*Termination Verification Condition*

\[ \text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ \text{SET}))) \]
\[ \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]
\[ \ (\text{LESSP} \ (\text{COUNT} \ \text{SET}) \ (\text{COUNT} \ \text{SET}))) \]

Case 3: \[ \text{AND} \ (\text{NOT} \ (\text{NLISTP} \ \text{SET}))) \]
\[ \ (\text{NOT} \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]

*Function Equivalence Verification Condition*

\[ \text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ \text{SET}))) \]
\[ \ (\text{NOT} \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]
\[ \ (\text{IFF} \ (\text{MEMB} \ \text{ELEMENT} \ (\text{CDR} \ \text{SET})) \]
\[ \ (\text{MEMB} \ \text{ELEMENT} \ \text{SET})) \]

*Termination Verification Condition*

\[ \text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ \text{SET}))) \]
\[ \ (\text{NOT} \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]
\[ \ (\text{LESSP} \ (\text{COUNT} \ \text{SET}) \ (\text{COUNT} \ \text{SET}))) \]

For all three cases the functional equivalence verification conditions are true since the MEMB in the conjectures refers to the definition of the normalized reference function, not the student’s initial or the currently partially debugged definition. Thus, without the termination verification conditions, all of which are false, Talus would not detect the infinite loops in MEMB. However, since the termination verification conditions are false, each symbolic value in the student’s definition of MEMB is considered buggy and is corrected with code from the corresponding symbolic value from the reference function. The final debugged code is:

\[
\text{(DEFUN MEMB} \ (\text{ELEMENT} \ \text{SET}) \]
\[ \ (\text{IF} \ (\text{NLISTP} \ \text{SET}) \]
\[ \ F \]
\[ \ (\text{IF} \ (\text{EQUAL} \ \text{ELEMENT} \ (\text{CAR} \ \text{SET}))) \]
\[ \ T \]
\[ \ (\text{MEMB} \ \text{ELEMENT} \ (\text{CDR} \ \text{SET}))) \))
\]
3.4.4 PALINDROMES Case Study

The PALINDROMES case study will demonstrate program simplification and edit inversion in the debugging process. First a simpler example is presented to clarify what happens in these steps. Consider the following task and solution:

**FACTORY TASK:** Write a function FACT that takes one argument N and returns the factorial of N. N will always be a natural number.

```
(DEFUN FACT (N)
  (COND ((ZEROP N) 0)
       (T (TIMES N (FACT (ADD1 N))))))
```

Talus first simplifies FACT to IF-Normal Form by rewriting all CONDs to IFs. The resulting program is shown below:

```
(DEFUN FACT (N)
  (IF (ZEROP N)
    0
    (TIMES N (FACT (ADD1 N)))))
```

Figure 3-10 shows part of the graph representation for the COND-TO-IF transformation that occurred in the simplification of the function FACT. Note the links between expressions that have been copied (dashed lines) or transformed (dotted lines). These links allow Talus to invert edits from the simplified code back to the original program.

Now Talus can debug the program since it is in IF-Normal form. The following hints are generated during the process:

**HINT:** Looks like you used the function ADD1 instead of the function SUB1 in FACT.

**HINT:** Looks like you used the number 0 instead of the number 1 in FACT.

The simplified program is debugged as shown:

```
(DEFUN FACT (N)
  (IF (ZEROP N)
    1
    (TIMES N (FACT (SUB1 N)))))
```

Talus traces these bug fixes back to the original code by following links in the graph representation of the program simplification transformation, as shown in Figure 3-11. The bug corrections in the original program are shown here:

```
(DEFUN FACT (N)
  (COND ((ZEROP N) 1)
       (T (TIMES N (FACT (SUB1 N))))))
```

A more complicated series of program transformations is required for the PALINDROMES case study. Assume we have the following task and solution:
Write a function that takes a list of elements and returns, in order, those elements which are lists that are unchanged when reversed. For example,

\[ \text{PALINDROMES of } (8 \ I \ (A \ B \ A) \ (A \ D) \ (E) \ NIL \ ((1 \ 2) \ (1 \ 2)) \ 9) \]

is...

\[ ((A \ B \ A) \ (E) \ ((1 \ 2) \ (1 \ 2))) \]

NOTE: all atoms that are elements in the input list are discarded. Thus,

\[ \text{PALINDROMES of } (X \ Y \ Z) \text{ is } \text{NIL}. \]
(DEFUN RETURN-PALINDROMES (L)
  (MAPCAN
   (FUNCTION
    (LAMBDA (EL)
      (IF (AND (LISTP EL) (EQUAL EL (REV EL)))
       (LIST L)
       NIL))
    (CDR L))))

(DEFUN REV (LIS)
  (PROG (ANS)
    CONTINUE
      (IF (NULL LIS) (RETURN ANS) NIL)
      (SETQ ANS (CONS LIS ANS))
      (SETQ LIS (CDR LIS))
      (GO CONTINUE)))

Figure 3-11: Edit Inversion through a COND-TO-IF Transform
First, Talus simplifies the MAPCAN in RETURN-PALINDROMES to a PROG:

```
(DEFUN RETURN-PALINDROMES (L)
  (PROG (_AC _LIST-LEFT)
     (SETQ _LIST-LEFT (CDR L))
     LOOP
     (IF (NULL _LIST-LEFT) (RETURN _AC) NIL)
     (SETQ _AC
       (NCONC _AC
         ((LAMBDA (EL)
           (IF (AND (LISTP EL)
                     (EQUAL EL (REV EL)))
             (LIST L)
             NIL))
          (CAR _LIST-LEFT))))
     (GO _LOOP))))
```

The variables beginning with underscores are internal identifiers that Talus creates. _AC accumulates partial results; _LIST-LEFT is the remainder of the list not yet seen. To perform this transformation, Talus instantiates a template for MAPCAN that expresses MAPCAN as a PROG. The slots of the template are underlined; as shown here they are filled with s-expressions directly copied from the initial program.

Both PROGs are then simplified to equivalent recursive functions. This algorithmic transformation will be discussed in detail in Chapter 8. The LAMBDA expression in RETURN-PALINDROMES can be expanded since there are no side effects in (CAR _LIST-LEFT). The result is:

```
(DEFUN RETURN-PALINDROMES (L) (_LOOP NIL (CDR L))))
```

;Recursive Function for MAPCAN in RETURN-PALINDROMES

```
(DEFUN _LOOP (_AC _LIST-LEFT)
  (IF (NULL _LIST-LEFT)
       _AC
       (_LOOP (NCONC _AC
                  (IF (AND (LISTP (CAR _LIST-LEFT))
                           (EQUAL (CAR _LIST-LEFT)
                                   (REV (CAR _LIST-LEFT))))
                           (LIST L)
                           NIL))))
  )
```

```
(DEFUN REV (LIS)
  (_CONTINUE NIL LIS))
```

;Recursive Function for Loop in REV

```
(DEFUN _CONTINUE (ANS LIS)
  (IF (NULL LIS)
      ANS
      (_CONTINUE (CONS LIS ANS) (CDR LIS))))
```

Talus has simplified the original program from the extended dialect into the core dialect. The program is not yet in IF-Normal Form, so Talus applies rewrites that move the IF's outside of all function calls. Talus also simplifies the program by removing the function REV; since it is nonrecursive it is
expanded inline in the function _LOOP. Now the program appears as:

(DEFUN RETURN-PALINDROMES (L) (_LOOP NIL (CDR L)))

(DEFUN _LOOP (AC LIST-LEFT)
  (IF (NULL _LIST-LEFT)
    AC
    (IF (AND (LISTP (CAR _LIST-LEFT))
      (EQUAL (CAR _LIST-LEFT)
        (_CONTINUE NIL (CAR _LIST-LEFT))))
      (_LOOP (NCONC AC (LIST L))
        (CDR _LIST-LEFT))
      (_LOOP (NCONC AC NIL)
        (CDR _LIST-LEFT))))))

(DEFUN _CONTINUE (ANS LIS)
  (IF (NULL LIS)
    ANS
    (_CONTINUE (CONS LIS ANS) (CDR LIS))))

This program version is in IF-Normal Form. Now, Talus attempts to simplify the program to the pure dialect by removing functions that cause side effects. There is one such function in the program above, NCONC. NCONC *splices* two lists together, destructively altering the last CONS cell in its first argument. However if the first argument is fresh list structure\textsuperscript{12}, Talus can replace NCONC by APPEND without altering the values returned by the student’s program. Since Talus can determine that AC is an accumulator variable initialized to NIL, and since LIST always creates fresh list structure, it is safe to replace NCONC by APPEND in the program above. The final program version, and the bugs that Talus finds, are shown below:

(DEFUN RETURN-PALINDROMES (L) (_LOOP NIL (CDR L)) ) \textit{Bug Found}

(DEFUN _LOOP (AC LIST-LEFT)
  (IF (NULL _LIST-LEFT)
    AC
    (IF (AND (LISTP (CAR _LIST-LEFT))
      (EQUAL (CAR _LIST-LEFT)
        (_CONTINUE NIL (CAR _LIST-LEFT))))
      (_LOOP (APPEND AC (LIST L)) ) \textit{Bug Found}
        (CDR _LIST-LEFT))
      (_LOOP (APPEND AC NIL)
        (CDR _LIST-LEFT))))))

(DEFUN _CONTINUE (ANS LIS)
  (IF (NULL LIS)
    ANS
    (_CONTINUE (CONS LIS ANS)
      (CDR LIS)))) \textit{Bug Found}

Talus corrects the bugs, using the reference code as a source of corrections:

\textsuperscript{12}A list is fresh list structure if there are no pointers to it, i.e. its "reference count" is 0.
(DEFUN RETURN-PALINDROMES (L)
  (_LOOP NIL L)) ;Bug Fix

(DEFUN _LOOP (_AC _LIST-LEFT)
  (IF (NULL _LIST-LEFT)
      _AC
      (IF (AND (LISTP (CAR _LIST-LEFT))
        (EQUAL (CAR _LIST-LEFT)
          (_CONTINUE NIL (CAR _LIST-LEFT))))
        (_LOOP (APPEND _AC (LIST (CAR _LIST-LEFT))) ;Bug Fix
          (CDR _LIST-LEFT))
        (_LOOP (APPEND _AC NIL)
          (CDR _LIST-LEFT))))))

(DEFUN _CONTINUE (ANS LIS)
  (IF (NULL LIS)
    ANS
    (_CONTINUE (APPEND (LIST (CAR LIS) ANS)) ;Bug Fix
      (CDR LIS))))

and then traces the edits back through the program transformations to determine the edits in the original program:

(DEFUN RETURN-PALINDROMES (L)
  (MAPCAN
    (FUNCTION
      (LAMBDA (EL)
        (IF (AND (LISTP EL) (EQUAL EL (REV EL)))
          (LIST EL)
          NIL)))
    L)) ;Bug Fix

(DEFUN REV (LIS)
  (PROG (ANS)
    CONTINUE
    (IF (NULL LIS) (RETURN ANS) NIL)
    (SETQ ANS (APPEND (LIST (CAR LIS)) ANS)) ;Bug Fix
    (SETQ LIS (CDR LIS))
    (GO CONTINUE)) ))

Note that the bug correction (CAR _LIST-LEFT), which refers to the internal variable _LIST-LEFT, is inverted to the bug correction EL, as shown above. The default means of edit inversion, simply following pointers, was overruled by a demon associated with the PROG removal transform. This demon, referred to as an edit inversion method, intercepted the edit and determined the correction necessary to remove the reference to the internal variable _LIST-LEFT.

3.4.5 REVERSE Case Study

Although all functions written in the extended dialect can be simplified to the core dialect, not all can be simplified to the pure dialect. Consider the task and solution below,

REVERSE TASK: Write a function to reverse a proper list.
(DEFUN REV (L) (REV* L NIL))

(DEFUN REV* (L ANS)
 (IF (NULL L)
     NIL
     (REV* (CAR L) (RPLACD L (APPEND NIL ANS)))))

(RPLACD x y) destructively alters the CDR of x to be y. In REV*, RPLACD is not applied to fresh list structure and thus no program simplification transformation applies. Talus knows three algorithms for the REVERSE task: RECURSIVE-REVERSE, ITERATIVE-REVERSE, and DESTRUCTIVE-REVERSE. The last algorithm best matches this solution. The reference functions for that algorithm are:

(DEFUN REVERSE (LIS) (REVERSE1 LIS NIL))

(DEFUN REVERSE1 (LIS ANSWER)
 (IF (NULL LIS)
     ANSWER
     (REVERSE1 (CDR LIS) (RPLACD LIS ANSWER))))

These functions are normalized to:

(DEFUN REV (L) (REV* L NIL))

(DEFUN REV* (L ANS)
 (IF (NULL L)
     ANS
     (REV* (CDR L) (RPLACD L ANS)))))

Debugging proceeds as before for REV and REV*. No bugs are found in REV. There are two cases to consider for REV*: either (NULL L) is true or false. Talus checks the following functional equivalence verification condition for the first case:

(IMPLIES (NULL L) ;Case
    (EQUAL ANS ;From Stored Fn
      NIL)) ;From Student Fn

and discovers that it is false, indicating a bug: REV* should return ANS, not NIL when it terminates.

Debugging cannot proceed as usual for the second case, when (NULL L) is false. Both student and reference functions simplify to expressions containing RPLACD. The presence of RPLACD, which has no formal semantics in the Boyer-Moore Logic, prevents the generation of a verification condition that is a well formed formula in that logic. Furthermore it is not enough to compare the program fragments for functional equivalence since the correctness of the side effect performed in the student's code fragment must also be checked.

Talus isolates the s-expressions containing side effects and compares them heuristically while verification conditions that are wff's are generated for the remaining s-expressions. Consider the case when Talus symbolically evaluates the student and reference functions, assuming that (NULL L) is false. The student's function simplifies to

(REV* (CAR L) (RPLACD L (APPEND NIL ANS)))

and the reference function simplifies to
(REV* (CDR L) (RPLACD L ANS))

Since the first argument to REV* in each fragment contains no side effects, Talus generates a functional equivalence verification condition to perform the comparison. However, since RPLACD occurs in the second arguments, Talus must use heuristics to compare them.

The functional equivalence verification condition that compares the first arguments of REV* is:

(IMPLIES (NOT (NULL L)) ; Case
        (EQUAL (CDR L) ; From Reference Function
             (CAR L))) ; From Student Function

This conjecture is not true, indicating a bug, the use of CAR instead of CDR.

Heuristics are used to compare the fragments containing side effects. Talus checks that

(RPLACD L (APPEND NIL ANS)) ; From Student Function

and

(RPLACD L ANS) ; From Reference Function

would perform the same side effect by first comparing the main functions called in the two program fragments. These must be the same, as they are in this example, if either expression can cause a side effect. Otherwise the student’s code fragment will be considered buggy and replaced by the reference code fragment.

Next, Talus checks that the data structures being altered are identical, not just functionally equivalent. Since both expressions alter L they also pass this test. Finally, the remaining arguments must be functionally equivalent. Since ANS and (APPEND NIL ANS) are functionally equivalent, the final test is passed. The final debugged code is:

(DEFUN REV (L) (REV* L NIL))

(DEFUN REV* (L ANS)
             (IF (NULL L)
                 ANS
                 (REV* (CDR L) (RPLACD L (APPEND NIL ANS)))))

3.5 Computational Semantics in Debugging

The details and complexity of the program debugging process can obscure the kinds of reasoning that Talus performs about programs and programming language semantics. Three kinds of reasoning about computational semantics occur:

- **Heuristic Reasoning** derives plausible inferences based on the application of heuristics rather than formal rules of inference. The semantics of heuristic inferences is difficult to state formally, since such inferences are of the nature "x appears most similar to y." Talus uses heuristic reasoning to compare algorithms, functions, and formal variables in its algorithm recognition phase, and to localize bugs in its bug correction phase.

- **Conjecture-Based Reasoning** occurs when debugging decisions are made on the basis of the evaluation of conjectures. The conjectures are well-formed formulas in the Boyer-Moore Logic and their truth value is well defined by formal rules. Talus uses conjecture-based reasoning extensively in its bug detection and correction phases to perform:
  1. **Symbolic Evaluation.**
2. Checking for Functional Equivalence.


4. Disproving Heuristic Conjectures.

- **Reasoning using Encoded Rules of LISP Semantics** is procedural reasoning performed by Talus routines that exhibit knowledge about LISP programming language semantics, such as the ability to:
  1. Parse programs into frames.
  3. Perform Program Simplification.
  4. Invert Edits.
  5. Determine if Arguments are Fresh List Structure.
  6. Transform Algorithms into Algorithm Variants.

### 3.6 Summary

This chapter has demonstrated how Talus uses its abilities to reason about computational semantics to debug nontrivial recursive programs that vary significantly. There are four stages to program debugging:

- **Program Simplification.** Functions are reduced to a simpler form.

- **Algorithm Recognition.** The student’s algorithm is identified and the reduced functions are mapped to reference functions.

- **Bug Detection.** For each function pair, verification conditions are generated from an inductive proof of equivalence.

- **Bug Correction.** Violated verification conditions are repaired by altering student functions using code from reference functions.

In these four stages, Talus performs these three kinds of reasoning about computational semantics:

- **Heuristic Reasoning.** Examples:
  1. Algorithm Recognition
  2. Function Mapping
  3. Formal Variable Mapping
  4. Bug Localization

- **Conjecture-based Reasoning.** Examples:
  1. Symbolic Evaluation
  2. Functional Equivalence Verification Conditions
  3. Termination Verification Conditions

- **Reasoning using Encoded Rules of LISP Semantics.** Examples:
  1. Parsing programs into frames.
  3. Program Simplification.
4. Edit Inversion.
5. Determining if Arguments are Fresh List Structure.
6. Transforming Algorithms into Algorithm Variants.
Chapter Four

E-frames: A Frame Representation for Functions

Chapter 3 summarized the debugging process of Talus. This chapter focuses on the representations that Talus uses for programs and algorithms. These representations are important since they allow Talus to identify the stored algorithm that most closely matches the student's and then to pair that algorithm’s reference functions to the student’s. The key property of these representations that facilitates algorithm recognition is their ability to capture abstract computational features of programs that tend to remain invariant over alternate implementations of the same algorithm. Many of these features are related to the recursive data structures that are operated on and constructed by recursive functions. In contrast, features that are derived from more superficial aspects of the code such as the code structure, data flow, or control flow vary widely between implementations of the same algorithm and are more easily affected by the presence of bugs in programs.

Talus uses frames to represent both student functions and reference functions. Talus represents student solutions and stored algorithms as collections of E-frames. The "E" in "E-frame" stands for "enumeration". E-frames represent recursive functions as abstract enumerations of recursively defined data structures. This means that E-frames represent the way that recursive functions access and process the elements of data structures such as trees or lists. This will be explained more precisely in this chapter and examples will be given. E-frames extract features from recursive programs that are useful predictors of the algorithms that those programs implement, even if the implementations are buggy. This property of E-frames is important since the entire algorithm recognition process depends on it.

4.1 Recursively Defined Data Structures and Recursive Functions

E-frame slots represent abstract properties of recursive functions, such as the type of data structure enumerated or constructed by a function. In order to infer these properties it is necessary to recognize LISP primitives that play similar roles for different recursive data structures, such as:

- **Bottom Object Recognizers.** These are functions that recognize objects that cannot be decomposed. E.g. the bottom object of a proper list is NIL and is recognized by NULL; the bottom object of a natural number is 0 and is recognized by ZEROP.

- **Data Constructors.** These are primitive functions that construct the data structure. E.g. CONS returns a list; ADD1 returns a natural number.

- **Data Accessors.** These functions are complementary to data constructors, they retrieve the components of a composite data structure. E.g. CAR and CDR are data accessors for nonatomic s-expressions; SUB1 is a data accessor for a nonzero natural number.

While the reader may be used to thinking of lists and trees as recursively defined data structures, thinking of natural numbers in this way may be foreign. The list (A B C) can be thought of as a more convenient notation for (CONS 'A (CONS 'B (CONS 'C NIL))). Similarly with numbers, the number 3 can be considered an abbreviation for the data structure (ADD1 (ADD1 (ADD1 0))). For lists, CDR is the data accessor. For example, CDR of (CONS 'A (CONS 'B (CONS 'C NIL))) is (CONS 'B (CONS 'C NIL)). SUB1 plays the role of CDR for numbers since SUB1 is a data accessor and SUB1 of (ADD1 (ADD1 (ADD1 0))) is (ADD1 (ADD1 0)).
Knowing the roles of LISP primitives in implementing recursively defined data structures is important since it allows underlying similarities between apparently dissimilar recursive functions to be seen. For example, consider the two functions APPEND and PLUS:

```lisp
(DEFUN APPEND (X Y)  
  (IF (NLISTP X)  
      Y  
      (CONS (CAR X)  
             (APPEND (CDR X) Y))))

(DEFUN PLUS (X Y)  
  (IF (ZEROP X)  
      Y  
      (ADD1  
             (PLUS (SUB1 X) Y))))
```

These functions essentially copy the data structure x onto the data structure y. The primary difference between these two functions is that APPEND expects x and y to be lists while PLUS expects them to be numbers. Notice that both functions follow this pattern:

```lisp
(DEFUN fn (X Y)  
  (IF (bottom-object-recognizer X)  
      Y  
      (data-constructor  
                         (fn (data-accessor X) Y))))
```

Talus searches for LISP primitives that act as data accessors, constructors, and bottom object recognizers to infer a function's recursion type. A function's recursion type is one of:

- **List Recursion.** The elements of a CONS data structure are enumerated with the data accessor CDR. ATOM, LISTP, NLISTP, or NULL test for the bottom element.

- **Tree Recursion.** The elements of a CONS data structure are enumerated by both the CAR and CDR data accessors. ATOM, LISTP, or NLISTP test for bottom elements.

- **Number Recursion.** The elements of a NUMBER data structure are enumerated by the data accessor SUB1. ZEROP or EQUAL test for the bottom element.

- **Unknown.**

Notice that the buggy student function FLAT, taken from the MEMTREE case study in Section 3.4.1:

```lisp
(DEFUN FLAT (ANS TR)  
  (IF (ATOM TR)  
      ANS  
      (FLAT (FLAT ANS (CDR TR))  
             (CAR TR))))
```

and the correct reference function FLATTEN:

```lisp
(DEFUN FLATTEN (TREE)  
  (IF (NLISTP TREE)  
      (LIST TREE)  
      (APPEND (FLATTEN (CAR TREE))  
             (FLATTEN (CDR TREE))))))
```

both have the same recursion type, TREE-RECURSION, despite their many other differences. Talus easily establishes that both functions satisfy the TREE-RECURSION criteria above since FLAT terminates with ATOM, FLATTEN terminates with NLISTP, and both functions have occurrences of
CAR and CDR in recursive function calls.

4.2 Computational Features of Recursive Functions: The E-frame Slots

The last section provided an example of a computational feature that Talus extracts from recursive functions. Not surprisingly, RECURSION-TYPE is an E-frame slot. In general, E-frame slots represent abstract properties of recursive functions and the way they enumerate recursively defined data structures. This section discusses the other E-frame slots and their semantics.

4.2.1 Abstract Properties of Enumerations and Recursive Functions

E-frame slots emphasize fundamental similarities between recursive functions that implement the same algorithm. Superficial differences of alternate implementations are suppressed. Consider two functions that both solve the FLATTEN task, GET-ATOMS:

```
(DEFUN GET-ATOMS (TREE)
  (IF (ATOM TREE)
    NIL
    (CONS TREE NIL)
    (APPEND (GET-ATOMS (CAR TREE)) (GET-ATOMS (CDR TREE)))))
```

and SMASH:

```
(DEFUN SMASH (PAIR)
  (IF (LISTP PAIR)
    (APPEND (SMASH (CDR PAIR)) (SMASH (CAR PAIR)))
    (LIST PAIR)))
```

These two functions have the following superficial differences:

- **Different conditional tests.** GET-ATOMS uses ATOM; SMASH uses LISTP.
- **Different terminations.** GET-ATOMS calls CONS; SMASH calls LIST.
- **Different branching structure.** GET-ATOMS terminates when the if-true branch of its conditional is taken while SMASH terminates when its if-false branch is taken.
- **Recursive calls in differing orders.** GET-ATOMS calls itself recursively first on the CAR then the CDR, while SMASH operates in the reverse order.
- **Different identifiers.** The first function uses the identifiers GET-ATOMS and TREE while the second uses the identifiers SMASH and PAIR.

However, the deeper similarities that allow Talus to recognize these as alternate implementations of the same algorithm are the functions':

- **Recursion Type.** Each function performs a tree recursion.
- **Termination Criteria.** Each function terminates when given a non-list.
- **Recursive Calls.** There are two recursive calls in each function definition.
- **Construction of Output.** Each function uses APPEND to join the results of recursive calls together.
- **Output Data Type.** Each function returns a proper list.
- **Input Data Type.** Each function expects a CONS.

These similarities are represented in a group of E-frame slots called the recursion descriptor slots.
These slots characterize a recursive function's enumeration and construction of recursive data structures. The properties of enumeration and construction represented and the E-frame slots that represent them are:

- **The type of data structure enumeration.** The RECURSION-TYPE slot specifies a function's recursion type, as previously described.

- **The data types of all data structures referred to.** The VARIABLE-DATA-TYPES slot indicates the data types of the function's formal variables.

- **The type of data structure returned.** The OUTPUT-DATA-TYPE slot indicates the function's output data type.

- **When the enumeration stops.** The TERMINATIONS slot represents the logical conditions under which a function terminates and the values returned under those conditions.

- **When recursive calls occur.** The RECURSIONS slot represents the function's recursive calls and the logical conditions that must be true for the calls to occur.

- **How results of recursive calls are joined together.** The CONSTRUCTIONS slot represents how recursive calls are joined together to form the answer.

- **Whether elements are skipped over in recursive calls.** The VARIABLE-UPDATES slot represents the updates to formal variables and the logical conditions that govern the updates.

In addition to the recursive descriptor slots, additional bookkeeping slots represent a function's name, formal variables, and definition. Both types of slots are shown below in the E-frame for GET-ATOMS:

**Function Name - GET-ATOMS**

**Formals - (TREE)**

**Definition - (DEFUN GET-ATOMS (TREE))**

(IF (ATOM TREE)
   (CONS TREE NIL)
   (APPEND (GET-ATOMS (CAR TREE))
    (GET-ATOMS (CDR TREE))))

**Terminations - When (ATOM TREE) Return (CONS TREE NIL);**

**Recursions - When (NOT (ATOM TREE)) Call (GET-ATOMS (CAR TREE));**

**Constructions - When (NOT (ATOM TREE))**

Return (APPEND (GET-ATOMS (CAR TREE))
    (GET-ATOMS (CDR TREE))):
Variable-Updates - When (NOT (ATOM TREE))
   Update TREE To (CAR TREE);
When (NOT (ATOM TREE))
   Update TREE To (CDR TREE);

Variable-Data-Types - TREE Should-Be CONS

Output-Data-Type - PROPER-LIST

Recursion-Type - TREE-RECURSION

Notice how similar the recursion descriptor slots in the E-frame for SMASH are:

Function Name - SMASH

Formals - (PAIR)

Definition - (DEFUN SMASH (PAIR)
   (IF (LISTP PAIR)
      (APPEND (SMASH (CDR PAIR))
       (SMASH (CAR PAIR)))
   (LIST PAIR)))

Terminations - When (NOT (LISTP PAIR)) Return (LIST PAIR);

Recursions - When (LISTP PAIR) Call (SMASH (CDR PAIR));
   When (LISTP PAIR) Call (SMASH (CAR PAIR));

Constructions - When (LISTP PAIR)
   Return (APPEND (SMASH (CDR PAIR))
       (SMASH (CAR PAIR)));

Variable-Updates - When (LISTP PAIR)
   Update PAIR To (CDR PAIR);
   When (LISTP PAIR)
   Update PAIR To (CAR PAIR);

Variable-Data-Types - PAIR Should-Be CONS

Output-Data-Type - PROPER-LIST

Recursion-Type - TREE-RECURSION

4.2.2 The Role of Functions in Solutions

Additional E-frame slots capture the role of functions in solutions. These slots aid in determining the best function mapping between an algorithm’s reference functions and the student’s functions.

The static calling structure slots represent parent-child links in the static calling structure of a solution. The FUNCTIONS-CALLING slot in the E-frame for \( f \) lists those functions that call \( f \). The CONSTRUCTORS-CALLED slot lists the nonprimitive functions other than \( f \) that \( f \) calls outside of conditional tests. The PREDICATES-CALLED slot lists those nonprimitive functions that \( f \) calls inside
conditional tests exclusively. Recursive function calls are included since it is unusual for a function to call itself recursively as a predicate.

The FUNCTION-TYPE and FUNCTION-ROLE slots provide additional information about the role of functions in tasks. The FUNCTION-TYPE slot indicates whether a function is recursive (type RECURSIVE), calls only LISP primitives (type SIMPLE), or calls other nonprimitive LISP functions (type CALLING). This information is useful since SIMPLE and CALLING functions can frequently be eliminated by treating the functions as macros; thus these functions play a lesser role in task solutions than RECURSIVE functions. The FUNCTION-ROLE slot characterizes the role a function plays in a task as one of TOP, MAIN, SUPPORTING-CONSTRUCTOR, SUPPORTING-PREDICATE, or EXTRA. Nonrecursive functions with no callers are top-level functions and are assigned a role of TOP. Usually there is only one TOP and one MAIN function in any solution. In the MEMTREE example studied in Section 3.4.1, the function MEMTR is assigned the role TOP. The solution is repeated below with comments indicating each function's role and type:

```
(DEFUN MEMTR (AT CONS) ;TOP fn role
   (IN AT (FLAT NIL CONS))) ;CALLING fn type

(DEFUN FLAT (ANS TR) ;SUPPORTING-CONSTRUCTOR fn role
   (IF (ATOM TR) ANS
     (FLAT (FLAT ANS (CDR TR))
       (CAR TR)))) ;RECURSIVE fn type

(DEFUN IN (X L) ;MAIN fn role
   (IF (LISTP L)
     (IF (EQUAL L (LIST X))
       L
       (IF (NOT (EQUAL (CAR L) X))
         (IN X (CAR L))
         L)
     NIL))
```

The function IN receives the MAIN function role since it is the first recursive nonprimitive function occuring in MEMTR. FLAT receives the function role SUPPORTING-CONSTRUCTOR by default. Section 4.3 describes the method by which Talus determines function roles and types.

4.2.3 Making Subtle Distinctions
The remaining E-frame slots enable Talus to distinguish between functions that vary in more subtle ways than would be evident from the previous slots. Their utility has been determined primarily from empirical observation. These slots are:

- The TERMINATION-FORMALS slot contains those variables that occur anywhere in any of the conditional tests that govern function terminations.

- The OUTPUT-FORMALS slot contains those variables that occur in function terminations. If nonempty this is usually a single variable that accumulates a list. Such accumulator variables are frequent in tail recursive implementations.

- The CONDITIONS slot represents the predicates and variables that occur in conditional tests.
Each record in this slot essentially paraphrases a conditional test by stripping out boolean connectives (e.g. NOT) and primitives that are not predicates (e.g. CAR). For example the predicate

\[ \text{(OR (NOT (EQUAL (CAR L) X)) (GREATERP (CAR L) X))} \]

is represented by the record:

**Predicates:** (EQUAL GREATERP)

**Variables:** (L X)

- The **SIDE-EFFECTORS slot** lists all primitives in the function that can cause side effects. These primitives, such as NCONC or ASET are only present in the core and extended dialects.

- The **DB-FETCH-FNS slot** lists all occurrences of AREF and GET. This slot can be used, for example, to distinguish between those functions that place property list markers and those that retrieve them in an algorithm that performs set intersection.

- The **PROGSNS slot** is the number of nontrivial PROGNs that occur in a function definition. A nontrivial PROGN contains two or more nonatomic s-expressions. This slot is useful in detecting functions that call other functions for side effect only.

### 4.3 Parsing Functions into E-frames

This section explains how Talus parses functions into E-frames. First Talus collects values for the recursion descriptor slots in a tree walk over a function’s definition. Then Talus sets the values of other slots that depend on the recursion descriptor slots. Talus determines values for the static calling structure slots by examining the functions called by each function. Values for the function role slots are determined heuristically. The function type slot can be set by inspecting the function body. Talus sets the remaining slots either by examining conditional tests or by flattening the function body and then searching for primitives such as NCONC that are not present in pure LISP.

The first step is to set the recursion descriptor slots. These slot values can be determined by performing a tree walk over any function definition in IF-Normal Form. A function definition is in IF-Normal Form if the function body contains no special forms other than IF, QUOTE, and PROGN, and if each IF expression is not inside any function call or conditional test.

All functions in the MEMTREE example are in IF-Normal Form. The following is not since one IF expression occurs inside the conditional test of another:

```lisp
(DEFUN MEM (X LIS)
 (IF (NILSTP LIS) NIL
     (IF (IF (EQUAL LIS (LIST X))
           T
           (EQUAL X (CAR LIS)))
        LIS
        (MEM X (CDR LIS)))))
```

The following is not in IF-Normal Form since an IF expression occurs inside a function call:
(DEFUN MAX (N LIS)
  (IF (NILSTP LIS)
      N
      (MAX (IF (GREATERP N (CAR LIS)) N (CAR LIS))
           (CDR LIS))))

With functions such as these, Talus applies rewrite rules to coerce them into IF-Normal Form. This reduction to IF-Normal Form can always be performed in the LISP dialects of Talus. Until program transformations are discussed in Chapter 8, we will assume, without loss of generality, that all functions are in IF-Normal Form.

If a function is in IF-Normal Form then it can be represented as a binary tree, as discussed in Section 3.3.2. To determine the recursive descriptor slots for a function’s E-frame, Talus traverses each branch of each conditional test in the function’s binary tree representation. Whenever the if-true branch of a conditional is taken, Talus assumes the conditional test is true. Whenever an if-false branch is taken, Talus assumes the conditional test is false. Eventually Talus reaches the terminal nodes that represent the function’s symbolic values. Talus adds each symbolic value that contains no recursive calls to the TERMINATIONS slot. Those expressions containing recursive calls are added to the CONSTRUCTIONS slot. In addition, Talus adds each recursive call contained in these expressions to the RECURSIONS slot. For each recursive call, Talus compares the ith actual argument to the ith formal argument of the function to determine a variable update record to add to the VARIABLE-UPDATES slot.

Talus not only records a function’s terminations, recursions, and variable updates, but also the conditions (i.e. governing terms) under which they occur. When a function termination or recursion is reached assuming

(AND \(i_1\) \(\ldots\) \(i_n\))

Talus records this term as part of each record added to any of the TERMINATIONS, CONSTRUCTIONS, RECURSIONS, or VARIABLE-UPDATES slots.

An example will clarify this process. Consider the following definition:

(DEFUN MCFLATTEN (TREE ANSWER)
  (IF (ATOM TREE)
      (CONS TREE ANSWER)
      (MCFLATTEN (CAR TREE)
                  (MCFLATTEN (CDR TREE) ANSWER))))

Talus traverses both branches of the IF expression to set the recursive descriptor slots. When (ATOM TREE) is assumed, the expression (CONS TREE ANSWER) is reached. Since no recursive call to MCFLATTEN occurs, Talus adds (CONS TREE ANSWER) to the TERMINATIONS slot, recording the fact that (ATOM TREE) must be assumed to reach it.

Assuming (NOT (ATOM TREE)) leads to the expression

(MCFLATTEN (CAR TREE) (MCFLATTEN (CDR TREE) ANSWER))

Since this expression contains recursive calls Talus adds this expression to the CONSTRUCTIONS slot, again recording the assumption required to reach the expression. There are two recursive calls in this expression:

1. (MCFLATTEN (CAR TREE) (MCFLATTEN (CDR TREE) ANSWER))
2. \texttt{(MCFLATTEN (CDR TREE) ANSWER)}

Talus adds each recursive call to the \texttt{RECURSIONS} slot.

Talus also updates the \texttt{VARIABLE-UPDATES} slot according to the updates in each recursive call. In the first recursive call the formal variable \texttt{TREE} is updated to \texttt{(CAR TREE)} and \texttt{ANSWER} is updated to \texttt{(MCFLATTEN (CDR TREE) ANSWER)}. In the second recursive call \texttt{TREE} is updated to \texttt{(CDR TREE)} and \texttt{ANSWER} is unchanged. Each of these four updates occurs under the same assumptions. All four updates are added to the \texttt{VARIABLE-UPDATES} slot.

The resulting E-frame for \texttt{MCFLATTEN}, with just these slots, is:

\begin{verbatim}
TERMINATIONS - When (ATOM TREE)
  Return (CONS TREE ANSWER);

RECURSIONS - When (NOT (ATOM TREE))
  Call (MCFLATTEN (CAR TREE)
    (MCFLATTEN (CDR TREE) ANSWER));
  When (NOT (ATOM TREE))
    Call (MCFLATTEN (CDR TREE) ANSWER);

CONSTRUCTIONS - When (NOT (ATOM TREE))
  Call (MCFLATTEN (CAR TREE)
    (MCFLATTEN (CDR TREE) ANSWER));

VARIABLE-UPDATES - When (NOT (ATOM TREE))
  Update TREE to (CAR TREE);
  When (NOT (ATOM TREE))
    Update ANSWER to
      (MCFLATTEN (CDR TREE) ANSWER);
    When (NOT (ATOM TREE))
      Update TREE to (CDR TREE);
    When (NOT (ATOM TREE))
      Update ANSWER to ANSWER;
\end{verbatim}

After determining the values of these recursive descriptor slots, Talus can set the values of other slots that depend on these values. The \texttt{TERMINATION-FORMALS} slot is the union of all variable references in the terms governing all function terminations. The \texttt{OUTPUT-FORMALS} slot is the union of all variable references in all function terminations. Talus sets the \texttt{RECURSION-TYPE} slot of an E-frame according to the variable updates of the variables listed in the \texttt{TERMINATION-FORMALS} slot. Usually there is only one variable to consider. For each variable \texttt{v}, if:

1. All the updates to \texttt{v} in recursive calls are \texttt{(SUB1 v)} or \texttt{(op v)} where \texttt{op} is an arithmetic operator then \texttt{NUMBER-RECURSION} is added to the \texttt{RECURSION-TYPE} slot.

2. All the updates to \texttt{v} in recursive calls are either \texttt{(CAR v)} or \texttt{(CDR v)} and both updates occur somewhere in the function, then \texttt{TREE-RECURSION} is added to the \texttt{RECURSION-TYPE} slot.

3. All the updates to \texttt{v} in recursive calls are \texttt{(CDR v)} and an update to \texttt{(CAR v)} does not occur, then \texttt{LIST-RECURSION} is added to the \texttt{RECURSION-TYPE} slot.

Next Talus determines the value of the \texttt{OUTPUT-DATA-TYPE} slot from the values of the
CONSTRUCTIONS and TERMINATIONS slots. If the constructors, those functions that join together the results of recursive calls, are only CONS, LIST or APPEND and the only termination is NIL then the output data type is PROPER-LIST. Otherwise, if CONS is the only constructor then the output data type is CONS. If the only constructors are arithmetic operators, such as ADD1 or TIMES, and the only terminations are numbers then the output data type is NUMBER. If the only constructors are boolean operators then the output data type is BOOLEAN. If there are no constructors but only recursive calls, and if the only function termination returns the value of a formal variable, then the output data type is that variable’s data type.

Talus does not use the VARIABLE-UPDATES slot to set the VARIABLE-DATA-TYPES slot, instead it extracts all variable references, regardless of where they occur. If a variable \( v \) occurs as an actual argument to a LISP primitive whose domain requires data of type \( t \) then a pair \( (v . t) \) is added to the VARIABLE-DATA-TYPES slot. \( t \) is one of CONS, PROPER-LIST, BOOLEAN, or NUMBER. For example in the function below:

\[
(\text{DEFUN NTHCDR} \ (I \ L) \\
(\text{IF} \ (\text{ZEROP} \ I) \\
L \\
(\text{NTHCDR} \ (\text{SUB1} \ I) \ (\text{CDR} \ L))))
\]

Talus will add the pairs \( (I \ . \ \text{NUMBER}) \) and \( (L \ . \ \text{CONS}) \) to the VARIABLE-DATA-TYPES slot. Talus distinguishes between the variable data types PROPER-LIST and CONS by examining terms governing function terminations. When a pair \( (v \ . \ \text{CONS}) \) occurs in the VARIABLE-DATA-TYPES slot and there is a term \( (\text{NULL} \ v) \) governing a termination, then Talus replaces the pair \( (v \ . \ \text{CONS}) \) with \( (v \ . \ \text{PROPER-LIST}) \). It is not always possible to correctly determine a variable’s data type, especially when bugs are present in the student’s program. For example, if the variable \( v \) appears in both \( (\text{ADD1} \ v) \) and \( (\text{APPEND} \ v \ x) \) then Talus cannot determine the data type of \( v \).

Talus sets the static calling structure slots by examining function calls. If \( f_1 \) calls \( f_2 \) then Talus adds \( f_1 \) to the FUNCTIONS-CALLING slot of \( f_2 \). If the function call to \( f_2 \) occurs outside of a conditional test then Talus adds \( f_2 \) to the CONSTRUCTORS-CALLED slot of \( f_1 \) and removes occurrences of \( f_2 \) from the PREDICATES-CALLED slot. If the call to \( f_2 \) occurs inside a conditional test and \( f_2 \) is absent from both the CONSTRUCTORS-CALLED and PREDICATES-CALLED slots of \( f_1 \) then Talus adds \( f_2 \) to the latter slot.

Values for the FUNCTION-ROLE slots are determined heuristically. The first step is to determine the top-level functions of a solution, these are the functions that have empty FUNCTIONS-CALLING slots. In most cases there is only one top-level function. Talus marks each nonrecursive top-level function TOP, that is it sets the FUNCTION-ROLE slot of the function’s E-frame to TOP. Talus marks each recursive top-level function MAIN. If no function has been marked MAIN then Talus assigns a function role of MAIN to the first recursive nonprimitive function in each termination of a top-level function. Talus expands functions with FUNCTION-TYPE slots equal to CALLING since they only obscure the more important functions that they call. If no function has been marked MAIN then Talus changes the function roles of the top-level functions from TOP to MAIN. There will always at least one MAIN function and in most cases there is exactly one.

Talus marks functions as EXTRA if their FUNCTION-TYPE slot is CALLING. Any function not yet marked that occurs in the CALLED-PREDICATES slots of some other function but never in any CALLED-CONSTRUCTORS slot is marked as a SUPPORTING-PREDICATE. Talus marks any remaining unmarked functions as SUPPORTING-CONSTRUCTORs.
Talus sets the remaining E-frame slots of a function by inspecting the function's body. It first sets the FUNCTION-TYPE slot. If a function calls itself recursively its type is RECURSIVE. If it is nonrecursive and calls other nonprimitive functions its type is CALLING. Otherwise its type is SIMPLE. Talus determines the value of the CONDITIONS slot by flattening all conditional tests in the function, recording only the predicates and variable references for each test. Talus performs a tree walk over a function's definition to set the SIDE-EFFECTORS, NONTRIVIAL-PROGNS, and DB-FETCH-FNS slots. Each function that can cause a side effect (e.g. NCONC) is pushed onto the SIDE-EFFECTORS slot. Occurrences of GET and AREP are pushed on the DB-FETCH-FNS slot. If a PROGN containing two or more s-expressions is encountered, then the NONTRIVIAL-PROGNS slot, which is initially zero, is incremented.

4.4 Representing Algorithms and Student Solutions with E-frames

So far we have considered the use of E-frames to represent individual functions. Talus also represents both algorithms and student solutions as lists of E-frames. To represent algorithms, Talus parses each reference function of an algorithm into a separate E-frame to obtain the algorithm representation. Similarly, each student function in a solution is parsed into a separate E-frame to obtain the solution representation.

A more precise definition of "algorithm" will help distinguish between algorithms and their implementations and provide a justification for the particular algorithm representation that Talus uses. Informally, an algorithm is a particular way of solving a task. A strategy for a task's solution is specified but implementation details are omitted. To be more precise, the following are provided:

- A division of the overall task into smaller subtasks.
- The data flow in this division, i.e. the connections between input and output in each subtask.
- Input/output specifications for each subtask along with a specification describing the means by which the output is computed, such as variable updates or loop invariants.

The purpose of the subtasks are expressed in domain concepts. The following are implementation details and are not specified:

- Function Identifiers
- Formal Variable Identifiers
- Ordering and Nesting of Conditional Expressions
- Program Code in Conditional Expressions, Function Terminations, and Function Recursions.

In the domain of TALUS the algorithm recognition problem can be described in the following way: given function definitions \( f_1, \ldots, f_n \) and their formal variables \( v_{f_1}, \ldots, v_{f_n} \) for each function \( f_p \) determine the teleology of the functions and formal variables and determine specifications that constrain acceptable implementations of these functions. The following questions must be answered:

- What purpose does each function \( f_i \) play in solving the task?
- What are the input and output specifications of each function \( f_i \)?
- What purpose does each variable \( v \) play in implementing the function in which it occurs?
- What properties should be true of recursive function calls \( (f_i, v_{f_i}, \ldots, v_{f_i}) \) in functions?

The algorithm recognition procedure of TALUS answers all of these questions. Most other automated debugging systems, such as PROUST [Johnson 85] do not allow multiple procedures in the solution to a
task. To the extent that these systems perform algorithm recognition, it is much more limited than TALUS since the first two questions above are not addressed.

In TALUS an algorithm is represented primarily\textsuperscript{13} by a collection of E-frames. This representation specifies:

- A Functional Decomposition - Each E-frame represents a function that achieves one subtask in a task decomposition. The \textsc{functions-calling, constructors-called, and predicates-called} slots of the E-frames represent the static calling structure of the functions in the algorithm.

- Function Specifications - The function definition part of each E-frame specifies requirements for correct implementations. Recursive functions that correctly implement the subtask corresponding to an E-frame must be functionally equivalent to the E-frame's function definition part. The function definition is just the definition of the reference function that was parsed into the E-frame. In terms of input/output specifications the reference function provides the output specifications while the implicit data restrictions of the reference function provide the input preconditions.

- Constraints on Function Recursions - For each case, the recursions of an acceptable implementation must be functionally equivalent to the \textit{corresponding recursions}\textsuperscript{14} of the reference function. This constraint restricts acceptable implementations in the same way as if loop invariants had been specified for imperative programs.

Consider the MEMTREE example. The task is to determine if an atom is a member of an s-expression. TALUS has two algorithms represented in its task representation. The first algorithm, TREE-WALK, explores the CAR and CDR of the s-expression recursively to determine if the atom is present. The second algorithm, MEMTREE-FLATTEN, flattens the s-expression into a bag and then determines if the atom is a member of the bag. In this algorithm the MEMTREE task is broken down into three tasks. Each smaller task is to write a function that performs a separate part of the overall task. The three subtasks are:

1. Write a function to flatten an s-expression.
2. Write a function to determine if an atom is present in a list.
3. Assuming the first two functions are available, write a function to flatten an s-expression then determine if an atom is a member of the resulting bag.

These two algorithms are \textit{represented} by collections of E-frames. The first algorithm (TREE-WALK) is represented by the \textit{E-frame} of the following reference function:

\textsuperscript{13}Additional information occurs in the MEASURES and IMPLICIT-RESTRICTIONS slots of the task frame.

\textsuperscript{14}If the reference function and its paired student function symbolically evaluate to \( r_1 \) and \( r_2 \) assuming \textit{case} then \( r_1 \) and \( r_2 \) are corresponding symbolic values for \textit{case}. If \( r_1 \) and \( r_2 \) both contain recursive calls then they are corresponding recursions.
(DEFUN MEMTREE (ITEM TREE)
  (IF (ATOM TREE)
      T
      (IF (MEMTREE ITEM (CAR TREE))
          T
          F))))

The second algorithm (MEMTREE-FLATTEN) is represented by the E-frames of the following reference functions:

(DEFUN MEMBER (ITEM BAG)
  (IF (NLISTP BAG)
      F
      (IF (EQUAL ITEM (CAR BAG))
          T
          (MEMBER ITEM (CDR BAG)))))))

(DEFUN FLATTEN (TREE)
  (IF (ATOM TREE)
      (LIST TREE)
      (APPEND (FLATTEN (CAR TREE))
              (FLATTEN (CDR TREE)))))))

The actual E-frames are much more lengthy than the reference functions they represent. The E-frames of the functions above are shown in Appendix IV.

Implementations are actual code intended to implement an algorithm. A correct implementation of the TREE-WALK algorithm is:

(DEFUN MEMTR (ITEM S-EXP)
  (IF (LISTP S-EXP)
      (OR (MEMTR ITEM (CDR S-EXP))
          (MEMTR ITEM (CAR S-EXP)))
      (OR (AND (NUMBERP ITEM)
               (AND (NUMBERP S-EXP)
                    (= ITEM S-EXP)))
          (EQUAL ITEM S-EXP))))

If the first occurrence of OR is changed to AND then the code will be a buggy implementation of the TREE-WALK algorithm. The MEMTREE case study illustrates a buggy implementation of the MEMTREE-FLATTEN algorithm and its debugging.
4.5 The Task Representation

A key part of the information that Talus needs in order to debug a student's solution to a task are the algorithm representations of expected algorithms for that task. The task representation provides this and other information that Talus requires to debug student solutions to a task. Tasks are also represented by frames, but not by E-frames. Each slot is described below and then illustrated with that slot's value for the MEMTREE task representation. Some slot values are simplified for clarity.

1. NAME - the name of the task.

MEMTREE

2. DESCRIPTION - instructions to give the student.

"Write a function that determines whether an atom is one of the leaves of a tree."

3. ALGORITHMS - names of algorithms that are represented in the FUNCTIONS slot and which correctly solve the task.

(TREE-WALK MEMTREE-FLATTEN)

4. BUGGY-ALGORITHMS - names of the buggy algorithms represented in the FUNCTIONS slot. Each algorithm on this list does not correctly solve the task, but occurs frequently enough to be worth recognizing. These algorithms either have some design flaw or they solve a task similar to the one assigned but obtained by misreading the instructions.

() ;No buggy algorithms for MEMTREE

5. COMMENTARY - an association list that pairs algorithm names with commentary templates. Each commentary template is a textual description of the role of the reference functions and their formal variables in the algorithm's implementation. Talus replaces references to stored identifiers with student identifiers after pairing functions and formal variables. This substitution allows Talus to generate program commentary describing the expected role of the student's functions and formal variables in a correct implementation of the algorithm identified.

((TREE-WALK .

"If item is atomic then item is a member of tree, only if item is EQUAL to tree. Otherwise, item is a member of tree only if it is a member either of the CAR of tree, or a member of the CDR of tree."
(MEMTREE-FLATTEN).

"memtree_f determines if item_v is a member of the tree tree_v by first flattening tree_v using the function flatten_f, and then determining if item_v is an element of the resulting list. Note that it is possible to determine if item_v is a member of the tree tree_v without flattening tree_v, i.e. by determining if item_v is a member of the CAR or the CDR of nonatomic trees, or EQUAL to atomic trees."

;Note: Each variable identifier subscripted with ;v and each function identifier subscripted ;with f will be replaced by its paired student ;identifier.

6. FUNCTIONS - an association list that pairs algorithm names with algorithm representations. Each algorithm is represented by a list of E-frames, one E-frame for each reference function. In the slot value below e-frame[f] stands for the E-frame that Talus parses f into.

(TREE-WALK).

(e-frame
  [(DEFUN MEMTREE (ITEM TREE)
      (IF (ATOM TREE)
          (EQUAL ITEM TREE)
          (IF (MEMTREE ITEM (CAR TREE))
              T
              (IF (MEMTREE ITEM (CDR TREE))
                  T
                  F)))]))
(MEMTREE-FLATTEN .

(e-frame
  [(DEFUN MEMTREE (ITEM TREE)
      (MEMBER ITEM (FLATTEN TREE)))]

(e-frame
  [(DEFUN MEMBER (ITEM BAG)
      (IF (NLISTP BAG)
        F
        (IF (EQUAL ITEM (CAR BAG))
          T
          (MEMBER ITEM (CDR BAG)))))]

(e-frame
  [(DEFUN FLATTEN (TREE)
      (IF (ATOM TREE)
        (LIST TREE)
        (APPEND (FLATTEN (CAR TREE))
                    (FLATTEN (CDR TREE)))))]

))

7. RESTRICTIONS - an association list that pairs reference function names with LISP predicates that constrain legal values for the functions’ formal variables.

  ((MEMTREE . (NOT (LISTP ITEM)))
   (MEMBER . (AND (PLISTP BAG)
                   (NOT (LISTP ITEM)))))

8. MEASURES - an association list that pairs reference function names with measures. Each measure is a LISP function that can refer to the reference function’s formal variables. Talus can use these measures and the well-founded relation LESSP to prove that each reference function is total or to detect nonterminating loops in student programs.

  ((MEMTREE . (COUNT TREE))
   (MEMBER . (COUNT BAG))
   (FLATTEN . (COUNT TREE)))

EXAMPLES - is an association list that pairs reference function names with sets of variable value bindings. These bindings test typical and boundary conditions for conjectures involving the reference function’s formal variables. They are used to quickly search for counterexamples to conjectures that arise in the debugging process.

  ((MEMTREE ((ITEM . A) (TREE . NIL))
             ((ITEM . A) (TREE . A))
             ((ITEM . A) (TREE . X))
             ((ITEM . A) (TREE . (B C)))
             ((ITEM . A) (TREE . (A B C)))
             ((ITEM . A) (TREE . (B A C)))
             ((ITEM . 2) (TREE . (A B . 2)))
             ((ITEM . A) (TREE . (B A . A)))))
(MEMBER ((ITEM A) (BAG NIL))
  ((ITEM A) (BAG (B C)))
  ((ITEM A) (BAG (A))))
  ((ITEM NIL) (BAG (U NIL 9)))
  ((ITEM NIL) (BAG (U)))
  ((ITEM NIL) (BAG (NIL)))
  ((ITEM X) (BAG (((A X) Y Z)))
  ((ITEM X) (BAG (((A X) Y X)))
  ((ITEM A) (BAG (B))))
  ((ITEM A) (BAG (A B C)))
  ((ITEM A) (BAG (B A C)))
  ((ITEM A) (BAG (B (A) C))))

(FLATTEN ((TREE (A)))
  ((TREE (A B)))
  ((TREE (NIL)))
  ((TREE (NIL X)))
  ((TREE (A B C)))
  ((TREE Q))
  ((TREE ((1) (2 3) (4))))))

For instance the last set of bindings for FLATTEN is a counterexample to the conjecture:

(IMPLIES (LISTP TREE)
  (EQUAL (CONS (CAR TREE)
  (FLATTEN (CDR TREE)))
  (APPEND (FLATTEN (CAR TREE))
  (FLATTEN (CDR TREE)))))

The conjecture would arise in comparing the following buggy definition of FLATTEN:

(DEFUN FLATTEN (TREE)
  (IF (LISTP TREE)
  (CONS (CAR TREE)
  (FLATTEN (CDR TREE)))
  (LIST TREE)))

to the reference definition.

4.6 Alternate Program Representations

This section compares the E-frame program representation of Talus to alternate program representations in other debugging and program analysis systems. Alternative representations are inadequate for the algorithm recognition task of Talus since:

- the features extracted vary considerably over alternate implementations of the same algorithm, and

- these features do not reliably distinguish between buggy implementations of different algorithms.

Ideally, the features extracted from implementations of the same algorithm should fall into the same equivalence class and the features extracted from implementations of different algorithms should fall into different equivalence classes.

Program debuggers that use heuristic plan recognition commonly represent programs in a graph
representation where control or data flow, or both, are made explicit. LAURA [Adam 80] and MENO-II [Soloway 83] use graph representations of this nature. Specific programming constructs that implement control or data flow are suppressed in the representation. The virtue of these representations is that algorithm implementations having the same data or control flow have the same program representation.

The PLAN representation of the Programmer’s Apprentice Project [Rich 81] represents a function’s abstract control and data flow in addition to input preconditions and output specifications. However, the PLAN representation does not store explicit representations of how recursive functions enumerate or construct recursively defined data structures. Although these properties could be derived from the PLAN representation they are not represented explicitly as they are in the E-frame representation. Instead the PLAN representation emphasizes data and control flow rather than abstract properties of recursive functions related to data structure enumeration and construction. As a consequence, alternate implementations of the same algorithm can be parsed into quite different PLAN representations if their control or data flow varies.\(^{15}\) For example, the two functions below have considerably different PLAN representations even though Talus would accept these as correct implementations of the same algorithm, assuming stylistic bugs are ignored.

```lisp
(DEFUN FLAT (TREE)
  (IF (LISTP TREE)
      (IF (NULL (CAR TREE))
          (FLAT (CDR TREE))
          (IF (ATOM (CAR TREE))
              (CONS (CAR TREE)
                  (FLAT (CDR TREE))))
          (APPEND (FLAT (CAR TREE))
                (FLAT (CDR TREE))))
      (IF (ATOM TREE)
          (IF (NULL TREE)
              TREE
              (CONS TREE NIL))
          NIL)))

(DEFUN FLAT (TREE)
  (IF (NULL TREE)
      NIL
      (IF (LISTP TREE)
          (IF (LISTP (CAR TREE))
              (APPEND (FLAT (CAR TREE))
                      (FLAT (CDR TREE)))
              (IF (NOT (NULL (CAR TREE)))
                  (CONS (CAR TREE)
                      (FLAT (CDR TREE))))
              (LIST TREE))))
```

Both implement an algorithm that performs a TREE-RECURSION to flatten s-expressions while not

\(^{15}\) Some readers may prefer an alternate definition of algorithm where differences in data flow or control flow indicate different algorithms. With this view, the E-frame representation differs from the PLAN representation in that it captures similarities between algorithms other than data and control flow.
collecting occurrences of NIL. The E-frame’s of these two implementations are quite similar, reflecting the common algorithm underlying both function definitions.

Information that facilitates algorithm recognition in Talus is not represented explicitly in the PLAN representation. For example, a function’s E-frame specifically represents the type of data structure enumerated, the function’s role in the task solution, and the terms that govern function terminations and recursions. These features are represented in the RECURSION-TYPE, FUNCTION-ROLE, TERMINATIONS, and RECURSIONS slots. None of these features are explicit in the function’s PLAN representation, although all could be extracted.

Some aspects of data and control flow are represented by E-frames, but in a different manner than in the PLAN representation. E-frame slots represent data flow between the functions in an algorithm and control flow within individual functions, but the representation is more abstract than that found in the PLAN representation. The static calling structure slots represent some of the data flow information present in a solution by specifying each function’s position in the overall static calling structure. However, unlike the PLAN representation, they do not represent the control paths taken to reach individual function calls or data flow between function outputs and inputs. Similarly the E-frame representation of control flow is more abstract than that in the PLAN representation. Control paths are represented by governing terms rather than explicit arcs that bifurcate at conditional expressions. It is this more abstract representation of control paths that explains why the E-frame representations of the two FLAT implementations above are more similar than the PLAN representations of the same functions.

Empirical evidence presented in Chapter 9 demonstrates that the algorithm recognition process of Talus achieves high performance in correctly recognizing the algorithms of buggy student programs. This data indicates that the E-frame program representation and the best first search procedure discussed in Chapter 5 together provide an effective means of representing and detecting similar algorithms. The Programmer’s Apprentice System does not rely on heuristically guided best first search for algorithm recognition, instead the plan recognition component of the Programmer’s Apprentice System is a bottom-up web grammar parser [Brotsky 84]. This parsing scheme extends parsing techniques applicable to context-free grammars to flow graphs such as the Programmer’s Apprentice PLAN representation. Graph expansions replace context-free production rules. The parser is designed to parse correct program implementations and requires exact matching of flow graphs. It appears that this parser would be impractical for recognizing algorithms in buggy student programs since:

- **Partial matching would be required.** Discrepancies in graph matches could be attributed to bugs and the partial matching required would vastly increase the number of possible matches between graphs. Since subgraph matching is required even when programs are correct, combinatorial explosion is all the more likely to happen when bugs are allowed.

- **It ignores programming language semantics.** Computationally equivalent operators, such as "NLISTP" and "ATOM", will not match since they differ syntactically. Therefore unexpected implementations of algorithms will not be recognized, or additional rules to account for alternate implementations must be added. In the latter case, the multitude of alternate implementations to consider will contribute to combinatorial explosion.

The partial matching heuristics of Talus require no graph matching and do not explode combinatorially as the size of the student’s functions increase. The size of the best first search performed

---

16These are called splits and joins in the PLAN representation [Rich 81].
during algorithm recognition depends on three factors:

1. The number of algorithms present in the task representation.
2. The number of bugs present in the student’s program.
3. The number of recursive functions present in the student and stored solution.

The inner loop of the search algorithm requires the comparison of two E-frames. This comparison time grows according to the number of recursions and terminations present in student and reference functions, not according to overall program size.

Suppose program traces were examined to determine the algorithm used. This technique could work for programs that are largely correct, terminating, and executable but fails rapidly as programs become more buggy since the bugs obscure the program’s algorithms. Furthermore, programs that implement different algorithms can have identical traces. Algorithms that depend on side effects will be difficult to recognize unless a record of side effects is incorporated into the execution trace.

A program representation that depends on extracting loop invariants from student programs will be inappropriate for algorithm recognition when programs are buggy. Even if invariants can be mechanically derived, the presence of only a few bugs in the student’s program can cause the synthesized invariants to differ so largely from the correct invariants of the intended algorithm that recognition is impossible.

4.7 Summary

The key points of this chapter are:

- E-frame slots represent abstract computational properties of recursive functions. E-frames facilitate algorithm recognition since their slot values differ much less between alternate implementations of the same algorithm than between alternate implementations of different algorithms. A function’s E-frame slots specify:
  1. The recursive data structure it enumerates,
  2. The function’s formal variable and output data types,
  3. Function terminations and the terms governing them,
  4. Function recursions and the terms governing them,
  5. Variable updates and the terms governing them,
  6. The function’s parents and children in the static calling structure of the solution,
  7. The function’s relative importance and role in the solution, and
  8. Whether or not the function uses side effects.

- Each student solution is represented by a list of E-frames, one E-frame for each function.

- Each algorithm is represented by a list of E-frames, one E-frame for each reference function.

- Alternate program representations such as flow graphs, program traces, or loop invariants are less well suited than E-frames for recognizing algorithms in buggy student programs for two reasons:
  - They require correct or nearly correct programs.
  - They extract features (e.g., data flow and control flow) that vary significantly over correct implementations of the same algorithm.
Chapter Five
Algorithm Recognition

This chapter explains how Talus recognizes algorithms in buggy student programs. The algorithm recognition process not only identifies the student's algorithm, but also maps the student's solution to a stored solution. Talus pairs reference functions with student functions and then pairs the functions' formal variables. These mappings are important since they allow Talus to detect and correct bugs by repairing induction proofs. In each function pair, Talus generates verification conditions that establish the equivalence of the student function to the reference function. Failed verification conditions indicate the presence of bugs; this bug detection process is discussed further in Chapter 6.

The algorithm recognition process requires partial matching to allow for bugs in the student's program. However, Talus does not directly match student function definitions to reference function definitions, rather it matches their E-frames. A heuristic evaluation function computes a weighted sum of the E-frame slot differences to evaluate how well two functions match. It is worth emphasizing that no syntactic pattern matching or graph matching occur during the algorithm process. The partial matching that occurs in Talus is not syntactic, like the matching of unification, but semantic since the strength of a match depends on E-frame slot values and the semantics of the slots.

In Talus, the problem of algorithm recognition has been reduced to the problem of determining the best mapping from one set of E-frames to another. Recall from Chapter 4 that algorithms are represented in the task representation as collections of E-frames, one for each reference function. Similarly, E-frames represent student functions in the student's solution. Talus selects the stored algorithm whose E-frames best match the E-frames from the student's solution as the algorithm intended. The best match minimizes \( f \), a function that measures the plausibility of a particular mapping of one set of E-frames to another. Section 5.1.4 describes \( f \).

Talus could generate all possible mappings from the E-frames in the student's solution to those in the stored solutions. After evaluating \( f \) for each mapping, Talus would select the mapping with the lowest score. This mapping would determine not only the stored algorithm most similar to the student's, according to \( f \), but also the best mapping of reference functions to student functions. However, generating all possible mappings is unnecessary. Instead, Talus performs a best first search that always finds the best possible mapping while exploring a much smaller number of candidates in most cases. The next section describes the search space and the heuristics that guide the search.

5.1 Partial Matching through Best First Search

Each state in the search space represents a mapping from student functions to reference functions. The reference functions in each state are always associated with only one of the stored algorithms. Suppose the student wrote a solution consisting of \( n \) functions, \( f_1 \ldots f_n \), to a task with two stored algorithms A and B. Let \( a_1 \ldots a_j \) and \( b_1 \ldots b_k \) be the reference functions for A and B. Then some states in the search space will represent alternate mappings from \( f_1 \ldots f_n \) to \( a_1 \ldots a_j \), while other states will represent alternate mappings from \( f_1 \ldots f_n \) to \( b_1 \ldots b_k \).

States are represented by records called "nodes"; these are described in Section 5.1.1. All nodes are scored according to \( f \); this measure is described in Section 5.1.4. The nodes are maintained on an ordered
list called the active list. At each step in the search the top node on the ordered list, the node with the lowest \( f \) score, is removed and expanded. A node is expanded by mapping the next unmapped student function to all possible unmapped stored functions; this can generate \( n \) new nodes if there are \( n \) unmapped stored functions. Actually, Talus limits the number of nodes generated by applying constraints that allow only plausible mappings to be considered. These constraints are discussed in Section 5.1.2. Talus also generates mappings where student or stored functions map to EXTRA. When there are no more student functions left to map, stored functions are always mapped to EXTRA. Student functions mapped to EXTRA are superfluous to the correct solution; stored functions mapped to EXTRA are essential to a correct solution but missing from the student's solution. The search terminates when the top node, which is guaranteed to have the lowest possible \( f \) score, is a complete mapping.

### 5.1.1 The Search Space

This section explains how nodes represent function mappings, and how the search space is initialized and expanded. Each node in the search space is a record having the following slots:

- **ALGORITHM** - The name of the algorithm whose reference functions are being mapped.
- **STUDENT-FNS-UNMAPPED** - The student functions that have not yet been mapped.
- **STORED-FNS-UNMAPPED** - The reference functions that have not yet been mapped.
- **PAIRED-FNS** - An association list that pairs student functions with the stored functions they have been mapped to.
- **EXTRA-STUDENT-FNS** - Student functions mapped to EXTRA.
- **MISSING-STORED-FNS** - Stored functions mapped to EXTRA.
- **MATCHES-SCORE** - A match score based solely on the paired functions.
- **SCORE-SO-FAR** - The sum of MATCHES-SCORE plus penalties for student and stored functions mapped to EXTRA.
- **MIN-EXPECTED-SCORE** - The sum of SCORE-SO-FAR plus the minimum possible penalty that will result when the remaining unmapped functions are mapped. This is the \( f \) score for the node.

One initial node is created for each algorithm in the task representation, with the ALGORITHM slot instantiated to the algorithm name. The names of the student's functions are placed in the STUDENT-FNS-UNMAPPED slot. The names of the algorithm's reference functions are placed in the STORED-FNS-UNMAPPED slot. In certain cases (discussed in Section 5.1.2) some of the student or stored functions are initially mapped to EXTRA and the initial node scores are nonzero. Otherwise all scores are initialized to zero and the remaining slots are empty.

For example, in the MEMTREE example presented in Section 3.4.1 there are two algorithms, MEMTREE-FLATTEN and TREE-WALK. The reference functions for these algorithms are shown in Section 4.5. The initial node for MEMTREE-FLATTEN is shown below:
Node 1

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MEMTREE-FLATTEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN-EXPECTED-SCORE</td>
<td>0</td>
</tr>
<tr>
<td>SCORE-SO-FAR</td>
<td>0</td>
</tr>
<tr>
<td>MATCHES-SCORE</td>
<td>0</td>
</tr>
<tr>
<td>PAIRED-FNS</td>
<td>( )</td>
</tr>
<tr>
<td>STUDENT-FNS-UNMAPPED</td>
<td>(MEMTR FLAT IN)</td>
</tr>
<tr>
<td>STORED-FNS-UNMAPPED</td>
<td>(MEMTREE MEMBER FLATTEN)</td>
</tr>
<tr>
<td>EXTRA-STUDENT-FNS</td>
<td>()</td>
</tr>
<tr>
<td>MISSING-STORED-FNS</td>
<td>()</td>
</tr>
</tbody>
</table>

The TREE-WALK algorithm only has one reference function unlike the student’s solution which has three functions. Talus applies constraints, discussed in Section 5.1.2, to choose which student functions must be considered extra. These functions, MEMTR and FLAT, are placed in the EXTRA-STUDENT-FNS slot. The initial scores reflect penalties for the two student functions considered extra, resulting in this initial node for the TREE-WALK algorithm:

Node 2

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>TREE-WALK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN-EXPECTED-SCORE</td>
<td>8</td>
</tr>
<tr>
<td>SCORE-SO-FAR</td>
<td>8</td>
</tr>
<tr>
<td>MATCHES-SCORE</td>
<td>0</td>
</tr>
<tr>
<td>PAIRED-FNS</td>
<td>( )</td>
</tr>
<tr>
<td>STUDENT-FNS-UNMAPPED</td>
<td>(IN)</td>
</tr>
<tr>
<td>STORED-FNS-UNMAPPED</td>
<td>(MEMTREE)</td>
</tr>
<tr>
<td>EXTRA-STUDENT-FNS</td>
<td>(MEMTR FLAT)</td>
</tr>
<tr>
<td>MISSING-STORED-FNS</td>
<td>()</td>
</tr>
</tbody>
</table>

New nodes are created by removing the top node on the active list and creating new nodes that:

- Pair the next unmapped student function to one of the unmapped reference functions. In the example above three new nodes could be created from Node 1 by mapping MEMTR to one of MEMTREE, MEMBER, or FLATTEN.

- Map the next student function to EXTRA. Another node could be created by mapping MEMTR to EXTRA.

- Map the next stored function to EXTRA. However, this can only happen when there are no more student functions left to map. If the STUDENT-FNS-UNMAPPED slot of Node 1 was empty then a new node could be created by mapping MEMTREE to EXTRA.

Each newly created node is scored and inserted into the active list according to its \( f \) score. The search terminates when a node with empty STUDENT-FNS-UNMAPPED and STORED-FNS-UNMAPPED slots is at the top of the active list.

Figure 5-1 shows the best first search that occurs in the MEMTREE example. Node 1 and Node 2 are the initial nodes. In the first step MEMTR is mapped to MEMTREE and EXTRA, resulting in Node 3 and Node 4. Next Node 3 is expanded by mapping IN to MEMBER and EXTRA, resulting in Node 5 and Node 6. Then Node 5 is expanded by mapping FLAT to FLATTEN and EXTRA, resulting in nodes
7 and 8. At this point Node 4 is expanded since it has the lowest score, 7.0. That produces Node 9, which has a score of 19.0. Node 7 is now at the front of the active list, and the search terminates since it is a complete mapping.

![Diagram of search process]

**Figure 5-1:** Best First Search during Algorithm Recognition in MEMTREE Case Study

The active list has only five nodes at this point since the expanded nodes (Node 1, Node 3, Node 4, and Node 5) have been removed. Talus interprets the top node as identifying the student’s algorithm and the correct function mapping. The top node is shown here:

**Node 7**

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MEMTREE-FLATTEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN-EXPECTED-SCORE</td>
<td>7.5</td>
</tr>
<tr>
<td>SCORE-SO-FAR</td>
<td>7.5</td>
</tr>
<tr>
<td>MATCHES-SCORE</td>
<td>7.5</td>
</tr>
<tr>
<td>PAIRED-FNS</td>
<td>(FLAT . FLATTEN)</td>
</tr>
<tr>
<td></td>
<td>(IN . MEMBER)</td>
</tr>
<tr>
<td></td>
<td>(MEMTR . MEMTREE)</td>
</tr>
<tr>
<td>STUDENT-FNS-UNMAPPED</td>
<td>()</td>
</tr>
<tr>
<td>STORED-FNS-UNMAPPED</td>
<td>()</td>
</tr>
<tr>
<td>EXTRA-STUDENT-FNS</td>
<td>()</td>
</tr>
<tr>
<td>MISSING-STORED-FNS</td>
<td>()</td>
</tr>
</tbody>
</table>

Talus used the slot values above to produce this text in the MEMTREE case study:
Algorithm used: MEMTREE-FLATTEN.

Student Fns Matched to Reference Fns:
- FLAT to FLATTEN
- IN to MEMBER
- MEMTR to MEMTREE

The MIN-EXPECTED-SCORE of 7.5 in the top node is attributable to the poor matches between the student's functions and the reference functions. A perfect match has a score of 0. Two bugs in the student's program and an extra conditional test in IN contributed to the score.

The second node in the ordered list is the initial node created for the TREE-WALK algorithm. Node 2, shown below, was never expanded since there was always some other node with a lower score on the active list. Of course, this node would have been expanded if the student's solution had been even more buggy and Node 7 had received a score greater than Node 2's.

Node 2

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>TREE-WALK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN-EXPECTED-SCORE</td>
<td>8</td>
</tr>
<tr>
<td>SCORE-SO-FAR</td>
<td>8</td>
</tr>
<tr>
<td>MATCHES-SCORE</td>
<td>0</td>
</tr>
<tr>
<td>PAIRED-FNS</td>
<td>()</td>
</tr>
<tr>
<td>STUDENT-FNS-UNMAPPED</td>
<td>(IN)</td>
</tr>
<tr>
<td>STORED-FNS-UNMAPPED</td>
<td>(MEMTREE)</td>
</tr>
<tr>
<td>EXTRA-Student-FNS</td>
<td>(MEMTR FLAT)</td>
</tr>
<tr>
<td>MISSING-STORED-FNS</td>
<td>()</td>
</tr>
</tbody>
</table>

The remaining three nodes are all alternate function mappings of the MEMTREE-FLATTEN algorithm. In the first MEMTR and MEMTREE are paired, the student function FLAT and the reference functions MEMBER and FLATTEN have not yet been mapped, and the student function IN has been mapped to EXTRA:

Node 6

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MEMTREE-FLATTEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN-EXPECTED-SCORE</td>
<td>12.5</td>
</tr>
<tr>
<td>SCORE-SO-FAR</td>
<td>4.5</td>
</tr>
<tr>
<td>MATCHES-SCORE</td>
<td>0.5</td>
</tr>
<tr>
<td>PAIRED-FNS</td>
<td>((MEMTR . MEMTREE))</td>
</tr>
<tr>
<td>STUDENT-FNS-UNMAPPED</td>
<td>(FLAT)</td>
</tr>
<tr>
<td>STORED-FNS-UNMAPPED</td>
<td>(MEMBER FLATTEN)</td>
</tr>
<tr>
<td>EXTRA-Student-FNS</td>
<td>(IN)</td>
</tr>
<tr>
<td>MISSING-STORED-FNS</td>
<td>()</td>
</tr>
</tbody>
</table>

A small penalty of 0.5 is assessed in matching MEMTR to MEMTREE since the functions have different names. A penalty of 4.0 is assessed in mapping IN to EXTRA. An additional penalty of 8.0 is assessed since at least one reference function will be unmapped in any complete mapping derived from this node. Similar penalties result in the high score of the last two nodes:
5.1.2 Reducing the Search Space: Constraints on Mappings

Some function mappings were not generated during the best first search in the MEMTREE example above. For example, Talus did not generate nodes where the student function MEMTR mapped to MEMBER or FLATTEN. These mappings are considered implausible since MEMTR has a different function role than either MEMBER or FLATTEN: MEMTR has role TOP, MEMBER has role MAIN, and FLATTEN has role SUPPORTING-CONSTRUCTOR.

Talus applies plausibility constraints to reduce the search space. Two functions can be paired only if:

- *They have the same function role.*
- *They have the same function type.*
- *Their parent functions have already been paired.*

There are some exceptions to the last constraint. Two functions can be paired if neither has parents. If either the student or reference function have multiple parents then one parent from one set must map to one in the other set. During the process of determining a student function's parents, functions whose role is EXTRA are skipped over since these functions usually cannot be paired with reference functions.

Talus also applies these constraints when it creates initial nodes in the search space. For example, the initial node for the TREE-WALK algorithm is:
Talus mapped MEMTR and FLAT to EXTRA since their function roles (TOP and SUPPORTING-CONSTRUCTOR) do not match the function role of any of the reference functions. In this case there is only one reference function, MEMTREE, and it has function role MAIN. This initial node is never expanded and reappears in the final active list when the search terminates.

Figure 5-2 shows some of the additional nodes that would be generated if the plausibility constraints were not enforced. With constraints four nodes are expanded and a total of nine nodes are created. Without constraints still only four nodes are expanded but now a total of fifteen nodes are created. However in this case, there were only two algorithms to consider. In the SINGLETONS case study presented in Section 3.4.2 there are seven correct algorithms and three buggy algorithms to consider. Table 5-1 summarizes the number of nodes that are expanded, created, and on the final active list in the best first search for the student’s algorithm. The first numbers in each column are the results when constraints are enforced; the parenthesized numbers are the results when constraints are lifted. Without constraints the number of nodes created increases when only correct algorithms are considered increases by about 50%. The number of nodes created increases more than 360% and the active list increases more than 600% when constraints are lifted.

5.1.3 Measuring E-frame Mismatch

This section discusses the heuristic evaluation functions that guide the search. The best first search that Talus implements is a refinement of the A* algorithm discussed in [Rich 83] and first described in [Hart 68] and [Hart 72]. The "best" node is the node with the lowest $f$ score where

$$f[S] = g[S] + h[S]$$

and where

- $S$ is a node in the best first search. Each node represents a partial or complete mapping of student functions to reference functions for one particular algorithm.

- $f$ estimates the quality of the best complete mapping that can result from extending the partial mapping in $S$ to a complete mapping.

- $g$ directly measures the quality of the function mapping in $S$.

- $h$ estimates the cost, or loss of quality, that results from extending the mapping in $S$ to a complete mapping.

The function $h$ used by Talus never overestimates this cost, thus the best first search of Talus is admissible [Nilsson 80]; this means that the best function mapping will always be found.

The functions $f$, $g$, and $h$ are computed using two more primitive functions:
Best First Search in MEMTREE, No Constraints

The grey nodes are created when constraints are lifted.

Figure 5-2: Search Paths Eliminated by Constraints

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Expanded</th>
<th>Created</th>
<th>Final Active List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Only</td>
<td>21 (31)</td>
<td>42 (154)</td>
<td>20 (122)</td>
</tr>
<tr>
<td>Buggy Only</td>
<td>9 (11)</td>
<td>19 (48)</td>
<td>9 (36)</td>
</tr>
</tbody>
</table>

First number: nodes when constraints are enforced.

Parenthesized number: nodes when constraints are not enforced.

Table 5-1: Effect of Plausibility Constraints on Search in SINGLETONS Case Study

• \( \text{match}(E_1, E_2) \) measures how well the E-frame \( E_1 \) matches the E-frame \( E_2 \). This measure is computed by functions that measure the differences of the E-frames' slots and then weight
the differences according to the semantics of the slots.

- **discard[E]** assesses a penalty for mapping the E-frame E to EXTRA.

In these functions and \( f, g, \) and \( h, 0 \) indicates a perfect match or no penalty and higher numbers indicate worse matches and greater penalties.

The function **match** compares E-frame slots as explained below:

- **FUNCTION-NAME slots** - A slight penalty is given to function matches when the function identifiers are different.

- **RECURSION-TYPE slots** - A small penalty is incurred when two functions have differing recursion types.

- **FORMALS slots** - No penalty is assessed if the two functions have the same number of formal variables. A very large penalty is assessed if the number of formal variables in the reference function exceeds the number of formal variables in the student’s function. Smaller penalties are assessed if the number of formal variables in the student’s function exceeds the number of formal variables in the reference function, provided that the difference cannot be accounted for by transforming the reference function into a tail recursive implementation. Such a transformation adds an extra formal variable to the reference function.

- **OUTPUT-DATA-TYPE slots** - No penalty is assessed when both functions have the same output data types. A small penalty is assessed if the student’s output data type is unknown, or if one data type is CONS and the other is PROPER-LIST. A larger penalty is assessed for clear mismatches, such as NUMBER to PROPER-LIST.

- **TERMINATIONS and RECURSIONS slots** - Penalties are assessed if one function has more terminations or recursions than the other. These penalties are moderate and proportional to the difference between the number of terminations or recursions.

- **PREDICATES-CALLED and CONSTRUCTORS-CALLED slots** - Penalties accrue if one function calls more nonprimitive predicates or constructors than the other. These penalties are moderate but heavier than those for differing numbers of terminations or recursions. A student function that has fewer constructors than a stored function is penalized more than one that has more constructors.

- **CONDITIONS slots** - Talus calls a function **cond-diff** that compares each record in the stored function to see if there is a similar record in the student’s function. Two records are similar if their predicates fall in the same equivalence class and if their variables are equivalent sets. There are three equivalence classes of predicates:
  1. **Equality Predicates**. The set (EQUAL -).
  2. **Comparison Predicates**. The set (LESSP GREATERP).
  3. **Atomic Predicates**. The set (ATOM LISTP NLISTP NULL).

A moderate penalty is assessed for each record in the stored function that does not have a similar record in the student function. Each missing conditional test in the student function. A lesser penalty is assessed for each record in the student function that does not have a similar record in the stored function. Each of these records corresponds to an extra conditional test in the student function.

- **SIDE-EFFECTORS and DB-FETCH-FNS slots** - Moderate penalties are assessed if one function calls primitives that produce or rely upon side effects and the other does not.

- **PROGNS slots** - If neither of the two functions calls primitives that cause or rely upon side effects, then a moderate penalty is assessed if one has a nontrivial PROGN but the other does not. Recall that a nontrivial PROGN contains two or more s-expressions, usually indicating
that the nonprimitive functions in the PROGN are being called for their side effects only.

A close approximation to \textit{match} is shown below:

\begin{verbatim}
match (Student, Stored) =
  ;Compare the function names
  (if (function-name[Student] = function-name[Stored])
    then 0.0 else 0.5)
  +

  ;Compare the recursion types
  (if (recursion-type[Student] = recursion-type[Stored])
    then 0.0 else 1.0)
  +

  ;Compare the number of formal variables
  (if (l(formals[Student]) = l(formals[Stored])) ; l means length
    then 0.0
    else
      (if (l(formals[Student]) > l(formals[Stored]))
        then (3.5 * [(l(formals[Student]) - l(formals[Stored]))
                     - (if (tail-recursion-transform-possiblep[Stored])
                         then 1 else 0)])
        else 7 * [l(formals[Stored]) - l(formals[Student])])
    +

  ;Compare the output data type
  (if (output-data-type[Student] = output-data-type[Stored])
    then 0.0
    else
      (if unknownp(output-data-type[Student])
        then 1.0 else 2.5))
  +

  ;Compare the number of terminations
  |l(terminations[Student]) - l(terminations[Stored])|
  +

  ;Compare the number of recursions
  |l(recursions[Student]) - l(recursions[Stored])|
  +
\end{verbatim}
; Compare the number of nonprimitive predicates called
  2 * |l(predicates-called[Student]) - l(predicates-called[Stored])|

+ 

; Compare the number of nonprimitive constructors called
  2 * max(0, |l(constructors-called[Stored]) - l(constructors-called[Student])|)

+ 

; Compare the predicates and variables in the conditionals
  cond-diff[Student, Stored] + 2 * cond-diff[Stored, Student]

+ 

; Compare occurrences of functions that cause side effects
  (Note: the function set-diff computes symmetric set difference)
  2 * |l(set-diff(side-effectors[Student], side-effectors[Stored])|

+ 

; Compare occurrences of functions that depend on previous side effects
  2 * |l(set-diff(db-fetch-fns[Student], db-fetch-fns[Stored])|

+ 

; Compare number of nontrivial PROGNs
  (if (emptyp(side-effectors[Student])) and
       emptyp(db-fetch-fns[Student]) and
       emptyp(side-effectors[Stored]) and
       emptyp(db-fetch-fns[Stored]))
  then
  2 * |l(progons[Student]) - l(progons[Stored])|
  else
  0)

5.1.4 Evaluating Function Mappings

Some function mappings will map student or reference functions to EXTRA. The function discard computes penalties for these mappings. discard calls two other functions: discard1 to compute penalties for stored functions and discard2 to compute penalties for student functions. Functions penalized by discard1 are necessary to the correct solution but missing from the student’s solution while functions penalized by discard2 are merely superfluous to the correct solution. discard1 penalizes functions more heavily than discard2 in accordance with this view.

Penalties in both discard1 and discard2 depend on both the function type and function role of the functions mapped to EXTRA. For example, a very large penalty is assessed if a function with role MAIN is mapped to EXTRA since most solutions have exactly one MAIN function and the MAIN student function should always map to the MAIN reference function. However, only a small penalty is assessed if a function with role EXTRA is mapped to EXTRA since such functions could be eliminated by treating them as macros and expanding calls to them inline.
Additional penalties are added if the function mapped to EXTRA either relied on or performed side effects, since these functions should match to student functions that act similarly. The penalty function for mapping stored E-frames to EXTRA is:

\[
\text{discard1}[F] =
\]

\[
\begin{align*}
& (\text{if } (\text{function-type}[F] = \text{CALLING}) \\
& \quad \text{then } \\
& \quad \quad (\text{if } (\text{function-role}[F] = \text{EXTRA}) \\
& \quad \quad \quad \text{then } 2 \text{ else } 3) \\
& \quad \text{else } \\
& \quad (\text{if } (\text{function-type}[F] = \text{RECURSIVE}) \\
& \quad \quad \text{then } \\
& \quad \quad \quad (\text{if } (\text{function-role}[F] = \text{MAIN}) \\
& \quad \quad \quad \quad \text{then } 8 \\
& \quad \quad \quad \quad \text{else } 6) \\
& \quad \quad \text{else } 3) \\
& \quad + \\
& 3 \times l(\text{side-effectors}[F]) \\
& + \\
& 3 \times l(\text{db-fetch-fns}[F]) \\
& + \\
& (\text{if } \left(\text{emptyp}(\text{side-effectors}[F]) \\
& \quad \text{and } \text{emptyp}(\text{side-effectors}[F])\right) \\
& \quad \text{then } 3 \times \text{progs}[F] \\
& \quad \text{else } 0)
\end{align*}
\]

The function \textit{discard2} that penalizes student E-frames mapped to EXTRA assesses lighter penalties than \textit{discard1}. This reflects the view that these E-frames represent functions that are extraneous to a minimal correct solution while those being discarded in \textit{discard1} are essential to that solution. Some provision is also made for student functions that do not call other functions in the solution. Some of these functions can be explained by assuming that the student is missing some other function, usually the function with role TOP, that would link the isolated function to the remainder of the solution. Other functions may be unnecessary redefinitions of primitives, such as a function that acts like \texttt{APPEND} does. It is assumed that these redefinitions will only appear in the fringe of the static calling structure tree, i.e. they will call no nonprimitives.
\[ discard2[F] = \]
\[
(\text{if } (\text{function-type}[F] = \text{CALLING})
\quad \text{then}
\quad (\text{if } (\text{function-role}[F] = \text{EXTRA})
\quad \quad \text{then } 2 \text{ else } 4)
\quad \text{else}
\quad (\text{if } (\text{function-type}[F] = \text{RECURSIVE})
\quad \quad \text{then}
\quad \quad (\text{if } (\text{function-role}[F] = \text{MAIN})
\quad \quad \quad \text{then}
\quad \quad \quad (\text{if } [\text{empty}\text{-}\text{typ}(\text{predicates-called}[F])] \text{ and}
\quad \quad \quad \quad \text{empty}\text{-}\text{typ}(\text{constructors-called}[F])]
\quad \quad \quad \quad \text{then } 4
\quad \quad \quad \quad \text{else } 8)
\quad \quad \text{else}
\quad \quad (\text{if } (\text{function-role}[F] = \text{SUPPORTING-PREDICATE})
\quad \quad \quad \text{then}
\quad \quad \quad (\text{if } [\text{empty}\text{-}\text{typ}(\text{predicates-called}[F])] \text{ and}
\quad \quad \quad \quad \text{empty}\text{-}\text{typ}(\text{constructors-called}[F])]
\quad \quad \quad \quad \text{then } 5
\quad \quad \quad \quad \text{else } 6)
\quad \quad \text{else } ;\text{SUPPORTING-CONSTRUCTOR}
\quad \quad (\text{if } [\text{empty}\text{-}\text{typ}(\text{predicates-called}[F])] \text{ and}
\quad \quad \quad \text{empty}\text{-}\text{typ}(\text{constructors-called}[F])]
\quad \quad \quad \text{then } 4
\quad \quad \quad \text{else } 6))
\quad \quad \text{else } ;\text{Function type must be SIMPLE.}
\quad \text{(if } (\text{function-role}[F] = \text{MAIN}
\quad \quad \text{then } 8 \text{ else } 3)))
\]

The function \( g \), which measures the quality of the function mapping represented by node \( S \), can be computed with \( \text{match} \) and \( \text{discard} \). Let \( \text{penalty} \) be zero initially, then for each pair of matched functions \( (f_1, f_2) \) in \( S \), add \( \text{match}(f_1, f_2) \) to \( \text{penalty} \). Then for each reference function \( f \) mapped to \( \text{EXTRA} \), add in \( \text{discard1}(f) \) and for each student function \( f \) mapped to \( \text{EXTRA} \), add in \( \text{discard2}(f) \). The final value of \( \text{penalty} \) is \( g(S) \).

The function \( h \) measures the minimum additional penalty that can result from extending \( S \) to a complete mapping. It is computed by determining the set of unmapped functions in \( S \) that can never be paired with another function of the same role and type. \( S \) is equal to the penalties, computed with \( \text{discard} \), that come from mapping those functions to \( \text{EXTRA} \).

The function \( f \) is then the sum of \( g \) and \( h \). Consider Node 8, taken from the best first search example of Section 5.1.1:
Node 8

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MEMTREE-FLATTEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN-EXPECTED-SCORE</td>
<td>1.6 :f[Node 8]</td>
</tr>
<tr>
<td>SCORE-SO-FAR</td>
<td>1.0 :g[Node 8]</td>
</tr>
<tr>
<td>MATCHES-SCORE</td>
<td>6.0</td>
</tr>
<tr>
<td>PAIRED-FNS</td>
<td>(IN . MEMBER) (MEMTR . MEMTREE)</td>
</tr>
<tr>
<td>STUDENT-FNS-UNMAPPED</td>
<td>()</td>
</tr>
<tr>
<td>STORED-FNS-UNMAPPED</td>
<td>(FLATTEN)</td>
</tr>
<tr>
<td>EXTRA-STUDENT-FNS</td>
<td>(FLAT)</td>
</tr>
<tr>
<td>MISSING-STORED-FNS</td>
<td>()</td>
</tr>
</tbody>
</table>

The value for MATCHES-SCORE is

\[
match(e-frame\ [IN]\ ,\ e-frame\ [MEMBER]) +
match(e-frame\ [MEMTR]\ ,\ e-frame\ [MEMTREE]) = 6.0
\]

Since the penalty for mapping FLAT to EXTRA is 4.0 (i.e. \(\text{discard2}(e-frame[\text{FLAT}]) = 4\) ) then \(g[S] = 4 + 6 = 10\). There is no function for the reference function FLATTEN to map to, so

\[
k[S] = \text{discard1}(e-frame[\text{FLATTEN}]) = 6.0
\]

and

\[
f[S] = g[S] + k[S] = 10 + 6 = 16
\]

5.2 Improving the Match

5.2.1 Solution Transforms

Talus applies global solution transforms to reduce the differences between the student's solution and the stored solution that Talus has identified as its best match. These transforms intentionally represent common algorithm variants. Rather than explicitly storing two very similar algorithms, Talus stores only one and provides a rule to derive the other algorithm, provided that this rule is a general rule that can apply for many algorithms and many tasks.

These solution transforms are called "global" since they always alter more than one reference function when they are applied. In contrast, consider the plan transformation rules of PROUST [Johnson 85]. These account for implementation variants within a single procedure. Thus PROUST will have plan transformation rules to show that "X-Y" and "Y+X" or "NEW > MAX" and "MAX < NEW" are acceptable implementation variants. Talus does not have this kind of plan transformation rule since the conjecture evaluator can intentionally represent all such rules. Although global solution transforms can be considered a kind of plan transformation rule, solution transforms account more for variability at the algorithm and function level than at the implementation level. These transforms represent alternate methods of functionally decomposing algorithms and interfacing the functions within an algorithm.

Each solution transform has three components:

- **Triggering Conditions**: These conditions are differences between student and reference functions that the transformation can reduce or eliminate.
• **Preconditions.** These are conditions that must be true for the transformation to be equivalence preserving.

• **Actions.** These are the changes produced by the routines that rewrite the reference functions to improve the function mapping.

An example of a global solution transform occurred in the MEMTREE case study presented in Section 3.4.1. Talus first identified the student’s algorithm:

**Algorithm used:** MEMTREE-FLATTEN.

**Student Fns Matched to Reference Fns:**
- FLAT to FLATTEN
- IN to MEMBER
- MEMTR to MEMTREE

and then transformed the reference function FLATTEN to MCFLATTEN:

**Solution Transform Applied:**
- Transforming FLATTEN to MCFLATTEN to better match the student function FLAT.

The student’s buggy definition of FLAT:

```
(DEFUN FLAT (ANS TR)
  (IF (ATOM TR) ANS
    (FLAT (FLAT ANS (CDR TR))
        (CAR TR))))
```

has two formal variables while the reference function that it was paired to:

```
(DEFUN FLATTEN (TREE)
  (IF (ATOM TREE)
    (LIST TREE)
    (APPEND (FLATTEN (CAR TREE))
         (FLATTEN (CDR TREE))))
```

has only one formal variable. This difference triggers the TAIL-RECURSION-INTRODUCTION transformation that replaces FLATTEN by MCFLATTEN, shown below:

```
(DEFUN MCFLATTEN (TREE ANSWER)
  (IF (ATOM TREE)
    (CONS TREE ANSWER)
    (MCFLATTEN (CAR TREE)
                 (MCFLATTEN (CDR TREE) ANSWER))))
```

This transformation also changes calls in the remaining reference functions from

```
(FLATTEN exp)
```

to

```
(MCFLATTEN exp NIL).
```

Talus can transform non-tail-recursive reference functions that perform either list or number recursions into equivalent tail-recursive implementations. For example, consider the reference function REMOVE that occurs in the SINGLETONS scenario in Section 3.4.2:
(DEFUN REMOVE (ITEM BAG)
  (COND ((NLISTP BAG) NIL)
    ((EQUAL ITEM (CAR BAG))
      (REMOVE ITEM (CDR BAG)))
    (T (CONS (CAR BAG)
      (REMOVE ITEM (CDR BAG))))))

and assume that it has been matched against a student function that has three formal variables, not two. This difference will trigger the TAIL-RECURSION-INTRODUCTION transformation, causing Talus to transform REMOVE to

(DEFUN REMOVE-TAIL-RECURSIVE (ITEM BAG AC)
  (COND ((NLISTP BAG) AC)
    ((EQUAL ITEM (CAR BAG))
      (REMOVE-TAIL-RECURSIVE ITEM (CDR BAG) AC))
    (T (REMOVE-TAIL-RECURSIVE ITEM (CDR BAG) (APPEND AC (LIST (CAR BAG)))))))

and to change calls in other reference functions from

(REMOVE exp₁ exp₂)

to

(REMOVE-TAIL-RECURSIVE exp₁ exp₂ NIL)

The PREDICATE-INVERSION transform accounts for another common implementation variant. Suppose an algorithm requires a function that tests for set membership. The reference function that Talus stores for this function will test to see whether an element is present in a set, but the student’s implementation could test to see whether an element is *absent* instead. An example of this occurs in the SINGLETONS case study presented in Section 3.4.2. In that example the student’s function ABSENT is matched to the reference function MEMBER. Talus applies the PREDICATE-INVERSION transform to alter MEMBER to NOT-MEMBER, where

(NOT-MEMBER x y) = (NOT (MEMBER x y))

The transform is described below:

- **Triggering Conditions:** This transform applies when a student function \( f₁ \) is paired with a stored function \( f₂ \) and
  1. Both have the function role SUPPORTING-PREDICATE,
  2. Each termination \( t₁ \) in \( f₁ \) reached assuming *case* equals (NOT \( t₂ \)) where \( t₂ \) is the corresponding termination in \( f₂ \) reached assuming *case*.

- **Preconditions:** Just the triggering conditions.

- **Actions:** Each termination \( t₂ \) in \( f₂ \) is replaced by (NOT \( t₂ \)) and each call \( (f₂ \text{ arg₁ ... argₙ}) \) in reference functions other than \( f₂ \) is replaced by (NOT \( (f₂ \text{ arg₁ ... argₙ}) \)).

Sometimes algorithms require a function \( f \) to be applied to the input data and then the results of \( f \) are processed by some other function. For example, in SINGLETONS the first step is to apply a function to flatten the input into a tree, so that some other function can extract the atoms that are unique in the bag. A typical solution might begin as:
(DEFUN SING (TREE)
  (FILTER (FLATTEN TREE)))

(DEFUN FILTER (BAG)
  .Body) ;Body extracts unique atoms from BAG.

... but occasionally students will merge SING and FILTER into one function that repeatedly flattens its input:

(DEFUN FILTER (TREE)
  ((LAMBDA (BAG)
    .Body) ;Body extracts unique atoms from BAG.
    (FLATTEN TREE)))

...

rather than doing it once and for all as in the first solution. Note that the second solution is correct - in that it returns the right values - even though it is inefficient. Also, the correctness of the second solution depends on the fact that FLATTEN is idempotent. A function \( f \) is idempotent if \( f(f(x)) = f(x) \) is always true. FLATTEN is idempotent since

\[
\text{EQUAL (FLATTEN (FLATTEN TREE)) (FLATTEN TREE)}
\]

is true. The function LIST is not.

Here is the description of the IDEMPOTENT-TRANSFORM, which accounts for student solutions such as the second one above:

* **Triggering Conditions.** Talus expects a solution of the following form:

;Form A

(DEFUN top (x) (main (f x)) )

(DEFUN main (y) BODY)

... but gets a solution of the form:

;Form B

(defun main (y) BODY[ (f y) /y] )

where TERM[\(a/b\)] means substitute all occurrences of \(a\) for \(b\) in TERM. More specifically, Talus looks for the following two reference functions mapped to EXTRA in the best function mapping:

* A function \( \text{top} \) with role TOP that calls a function \( f \).

* A function \( f \) with role SUPPORTING-CONSTRUCTOR called by \( \text{top} \). This function is forced to map to EXTRA once its parent (\( \text{top} \)) is mapped to EXTRA.
• **Preconditions.** The following must be a theorem: \((\text{EQUAL } f(f(X))) f(X))\).

• **Actions.** Talus changes its stored solution from Form A to Form B by eliminating the function \(\text{top}\) and by replacing \(y\) by \(f(y)\) in \(\text{main}\).

The fourth transform that Talus can apply is the \text{PLIST-MARKER-TRANSFORM}. This allows Talus to transform reference solutions to use the specific property list markers that a student’s solution uses. For example, suppose a student implements a set intersection algorithm that intersects two sets \(x\) and \(y\) using property list markers. The student’s solution places a value MARKED on the property INTERSECTION of each element in \(x\) and then collects those elements of \(y\) that have non-NIL values on their INTERSECTION property. The \text{PLIST-MARKER-TRANSFORM} allows Talus to alter its solution to also use the value MARKED and the property INTERSECTION in its implementation. The transform is described below:

• **Triggering Conditions.** The student’s solution uses property list markers.

• **Preconditions.** Just the triggering conditions.

• **Actions.** The property list markers in the reference solution are replaced by the student’s property list markers.

### 5.2.2 Relaxing Constraints

Sometimes, but not frequently, the plausibility constraints that restrict allowable function mappings prevent a student and reference function from being paired when they should be. For example, the code below is a correct solution to \text{SINGLETONS}:

```lisp
(DEFUN SINGLETONS (IN)
  (SING NIL NIL (FLAT IN)))

(DEFUN SING (ONES TWOS LEFT)
  (IF (NULL LEFT)
    ONES
    (IF (MEMB (CAR LEFT) ONES)
      (SING (TAKE-OUT (CAR LEFT) ONES)
        (CONS (CAR LEFT) TWOS)
        (CDR LEFT))
      (IF (MEMB (CAR LEFT) TWOS)
        (SING ONES TWOS (CDR LEFT))
        (SING (CONS (CAR LEFT) ONES)
          TWOS
          (CDR LEFT))))))

(DEFUN TAKE-OUT (X L)
  (IF (NULL L) NIL
    (IF (EQUAL X (CAR L))
      (TAKE-OUT X (CDR L))
      (CONS (CAR L)
        (TAKE-OUT X (CDR L))))))

(DEFUN MEMB (X L)
  (IF (NULL L) NIL
    (IF (EQUAL X (CAR L)) L
      (MEMB X (CDR L))))))
```
(DEFUN FLAT (L)
  (IF (LISTP L)
      (APPEND (FLAT (CAR L))
              (FLAT (CDR L))))
      (IF (NULL L) NIL (LIST L))))

Talus will correctly identify the algorithm and check that there are no bugs in the code. Now suppose the function SINGLETIONS is omitted. In this case, FLAT is no longer called by any function. Consequently it will receive the function role MAIN and will not be able to match to the reference function FLATTEN, which has the role SUPPORTING-CONSTRUCTOR. In addition, the reference function that previously mapped to SINGLETIONS will map to EXTRA since it now has no counterpart. That function, FLAT, and FLATTEN will all map to EXTRA in the best function mapping. To remedy this situation and allow FLAT to be debugged by pairing it with FLATTEN, Talus relaxes the plausibility constraints that prevent FLAT and FLATTEN from mapping. The function mapping is altered so that FLAT and FLATTEN are paired.

In general, if \( f_1 \) is a student function mapped to EXTRA and \( f_2 \) is a reference function mapped to EXTRA, and if \( f_1 \) or \( f_2 \) have a parent function also mapped to EXTRA, Talus will revise the best function mapping so that \( f_1 \) and \( f_2 \) are now paired. Talus only applies this heuristic if there is one pair of functions that fits this description, otherwise more than one revised pairing would be possible.

5.3 Mapping Formal Variables

After refining the function mapping, Talus maps the formal variables of paired functions by applying heuristics that take into account variable data type and role. For example, if functions \( f_1 \) and \( f_2 \) have been paired then the variables in the TERMINATION-FORMALS slot of \( f_1 \) ’ s E-frame are less likely to be mapped to the variables in the OUTPUT-FORMALS slot of \( f_2 \) ’ s E-frame than to variables in the TERMINATION-FORMALS slot. In the MEMTREE case study the results of these heuristics were the following formal variable mappings:

<table>
<thead>
<tr>
<th>FLAT to MCFATTEN:</th>
<th>(TR/TREE, ANS/ANSWER).</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN to MEMBER:</td>
<td>(L/BAG, X/ITEM).</td>
</tr>
<tr>
<td>MEMTR to MEMTREE:</td>
<td>(CONS/TREE, AT/ITEM).</td>
</tr>
</tbody>
</table>

Talus can only map the formal variables of paired functions when both functions have the same number of formal variables. This is not a serious limitation of Talus as shown by the empirical results presented in Chapter 9. The use of the TAIL-RECURSION-INTRODUCTION transform accounts for most instances where the number of formal variables differs between a student and stored function.

Talus determines the best mapping of formal variables between \( f_1 \) and \( f_2 \) by generating all possible mappings, scoring them with a function \( s \), and then choosing the mapping that minimizes \( s \). This is not impractical since most functions have a small number of formal variables, typically one or two. Each mapping \( M \) is a list of variable pairs. \( s[M] \) is the sum of \( vmatch[v_1, v_2] \) for each pair \( (v_1, v_2) \) in \( M \).

The function \( vmatch \) measures how implausible it is for \( v_1 \) to map to \( v_2 \). Talus examines the values of four E-frame slots (VARIABLE-DATA-TYPES, VARIABLE-UPDATES, TERMINATION-FORMALS, and OUTPUT-FORMALS) to determine how likely it is that \( v_1 \) and \( v_2 \) play similar roles in functions \( f_1 \) and \( f_2 \). The following slots contribute information:

- VARIABLE-DATA-TYPES - A penalty is assessed if the two variables have different data types.
TERMINATION-FORMALS - A penalty is assessed if one variable appears in a term guarding a function termination and the other does not.

VARIABLE-UPDATES - A penalty is assessed if the updates to the two variables differ.

OUTPUT-FORMALS - A penalty is assessed if one variable accumulates results to be returned by function terminations while the other does not.

Additional heuristics allow Talus to select the most appropriate variable mapping when the E-frame slots provide insufficient information. A small penalty is assessed if two variables are not both actual arguments to functions that have been paired. For example, in the MEMTREE case study Talus pairs the functions

\[ \text{FLAT to FLATTEN} \]
\[ \text{IN to MEMBER} \]
\[ \text{MEMTR to MEMTREE} \]

and then attempts to pair formal variables. Consider what happens when Talus tries to determine a formal variable mapping from MEMTR to MEMTREE. The two function definitions are repeated below.

\[
\begin{align*}
\text{(DEFUN MEMTR (AT CONS))} \\
\text{(IN AT (FLAT NIL CONS)))}
\end{align*}
\]

\[
\begin{align*}
\text{(DEFUN MEMTREE (ITEM TREE) )} \\
\text{(MEMBER ITEM (FLATTEN TREE)))}
\end{align*}
\]

The two possible mappings are:

\[ M_1: ((\text{CONS . TREE}) (\text{AT . ITEM})) \]

\[ M_2: ((\text{CONS . ITEM}) (\text{AT . TREE})) \]

Talus cannot use E-frame slot values to distinguish these two mappings: no variable data types can be inferred, no variable updates occur, and there are no termination or output variables. However since AT is an actual argument to IN and ITEM is an actual argument to MEMBER and since IN and MEMBER have been paired, Talus prefers \( M_1 \) to \( M_2 \).

A very small penalty is assessed in a mapping for each pair \( (v_1, v_2) \) where \( v_1 \) and \( v_2 \) appear in different positions in their functions' variable lists. Consider the following buggy solution to the MEMBER task:

\[
\begin{align*}
\text{(DEFUN MEM (X L) (MEM X L))}
\end{align*}
\]

When Talus compares MEM to the reference function MEMBER:

\[
\begin{align*}
\text{(DEFUN MEMBER (ITEM BAG) )} \\
\text{(IF (NLISTP BAG) F \text{IF (EQUAL ITEM (CAR BAG))} T (MEMBER ITEM (CDR BAG)))))}
\end{align*}
\]

it would have no reason to prefer the mapping

\[ M_1: ((X . ITEM) (L . BAG)) \]
5.4 Normalizing the Stored Solution

Once function and formal variables have been paired, Talus can replace identifiers that occur in reference functions by their paired student identifiers. This process, called normalization, allows Talus to correct buggy student code with normalized reference code. To illustrate normalization, recall the following Talus output from the MEMTREE case study:

Algorithm used: MEMTREE-FLATTEN.

Student Fns Matched to Reference Fns:
- FLAT to FLATTEN
- IN to MEMBER
- MEMTR to MEMTREE

Solution Transform Applied:
- Transforming FLATTEN to MCFLATTEN to better match the student function FLAT.

; Formal Variable Mappings

FLAT to MCFLATTEN: (TR/TREE, ANS/ANSWER).
IN to MEMBER: (L/BAG, X/ITEM).
MEMTR to MEMTREE: (CONS/TREE, AT/ITEM).

At this point, before normalization, the reference functions for the MEMTREE-FLATTEN algorithm are:

(DEFUN MEMTREE (ITEM TREE) (MEMBER ITEM (MCFLATTEN TREE NIL)))

(DEFUN MEMBER (ITEM BAG)
 (IF (NLISTP BAG)
   F
   (IF (EQUAL ITEM (CAR BAG))
     T
     (MEMBER ITEM (CDR BAG)))))

(DEFUN MCFLATTEN (TREE ANSWER)
 (IF (ATOM TREE)
   (CONS TREE ANSWER)
   (MCFLATTEN (CAR TREE)
               (MCFLATTEN (CDR TREE)
                           ANSWER))))

After substituting student identifiers for stored identifiers, the reference functions are:

(DEFUN MEMTR (AT CONS)
   (IN AT (FLAT CONS NIL)))
(DEFUN IN (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
           T
           (IN X (CDR L)))))

(DEFUN FLAT (TR ANS)
  (IF (ATOM TR)
       (CONS TR ANS)
       (FLAT (CAR TR)
             (FLAT (CDR TR) ANS))))

Recall that the student’s solution was:

(DEFUN MEMTR (AT CONS)
  (IN AT (FLAT NIL CONS)))

(DEFUN IN (X L)
  (IF (LISTP L)
      (IF (EQUAL L (LIST X))
          L
          (IF (NOT (EQUAL (CAR L) X))
              (IN X (CAR L))
              L))
      NIL))

(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
      ANS
      (FLAT (FLAT ANS (CDR TR))
            (CAR TR))))

The two solutions differ in the order of variables in the variable list of FLAT and the order of actual arguments in function calls to FLAT. This discrepancy illustrates why Talus must also permute variable order in reference functions to match the student’s; this permutation is part of the normalization process. The final normalized code for the MEMTREE-FLATTEN algorithm is shown below. Changes due to variable order permutation are underlined:

(DEFUN MEMTR (AT CONS)
  (IN AT (FLAT NIL CONS)))

(DEFUN IN (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (IN X (CDR L))))

(DEFUN FLAT (ANS TR)
  (IF (ATOM TR)
      (CONS TR ANS)
      (FLAT (FLAT ANS (CDR TR))
            (CAR TR))))
5.5 Program Commentary

After pairing identifiers, Talus can generate program commentary. The commentary describes the role that Talus assumes the student functions and formal variables were meant to play in implementing the identified algorithm. The commentary describes a correct implementation of the algorithm. The commentary generated for the MEMTREE case study is shown here:

MEMTR determines if AT is a member of the tree CONS by first flattening CONS using the function FLAT, and then determining if AT is an element of the resulting list. Note that it is possible to determine if AT is a member of the tree CONS without flattening CONS, i.e. by determining if AT is a member of the CAR or the CDR of nonatomic trees, or EQUAL to atomic trees.

The commentary was easily produced by substituting student identifiers for the italicized variables in the commentary template for MEMTREE-FLATTEN:

"memtree determines if item is a member of the tree tree by first flattening tree using the function flatten, and then determining if item is an element of the resulting list. Note that it is possible to determine if item is a member of the tree tree without flattening tree, i.e. by determining if item is a member of the CAR or the CDR of nonatomic trees, or EQUAL to atomic trees."

The program commentary makes explicit the assumptions about code teleology that Talus applies in the remainder of the debugging process. Each commentary template for an algorithm describes the role of the reference functions and their formal variables in implementing that algorithm. Talus assumes that student and stored functions have the same role if they have been paired. The teleology for reference functions, which is given, is transferred to the student functions they are paired with. In this view, the function and formal variable mappings represent the student’s plan to solve the task and the commentary generated is a paraphrase of the inferred plan.

5.6 Summary

The key points of this chapter are:

- Partial matching can be implemented as best first search.
- The heuristic evaluation function $f$ that Talus uses in its implementation of the A* algorithm guarantees that the best solution will always be found.
- The function $f$ estimates the quality of a function mapping based on the functions mapped to EXTRA and the differences in the E-frame slot values of paired functions. These differences are measured and weighted according to the semantics of the slots.
- Function mapping constraints reduce the search space.
- Solution transforms improve the best function mapping by rewriting that mapping’s reference functions to more closely match the student’s.
• Formal variables are mapped according to heuristics that pair variables with similar variable roles. Variable roles are inferred from E-frame slot values.

• Talus normalizes the reference functions by replacing stored identifiers by paired student identifiers. Normalization allows Talus to replace student code by reference code to correct bugs.

• Talus generates program commentary by instantiating commentary templates with student identifiers; this commentary describes the code teleology assumed.
Chapter Six

Bug Detection

This chapter discusses bug detection in Talus. Section 6.1 explains the symbolic evaluation process and its use in detecting redundant, missing, and extra conditional tests. Section 6.2 explains how Talus generates verification conditions sufficient to establish that a student function and its paired reference function are equivalent. Section 6.3 discusses different means of evaluating these verification conditions. Section 6.4 compares the bug detection strategy of Talus to alternative approaches.

6.1 Symbolic Evaluation

This section explains how functions are symbolically evaluated in Talus. Symbolic evaluation is important since it allows Talus to break down the task of debugging a function into several disjoint cases. This problem decomposition allows bugs to be isolated to code fragments. Talus considers the same cases that would be considered in an induction proof of equivalence between a student function and its paired reference function. Symbolic evaluation also plays an important role in detecting missing conditional tests. This latter role will be discussed first since Talus must detect and insert missing conditional tests prior to generating verification conditions.

6.1.1 Detecting Redundant Conditional Tests

Before comparing student and stored code to detect missing and extra conditional tests, Talus first examines the student’s code in isolation for redundant conditional tests: these are predicates in IF expressions that will always be true or false when reached. Redundant conditional tests are stylistic bugs since the student’s code can be simplified by eliminating these tests. To detect these tests, Talus performs a tree walk over the binary tree representation of the student’s function and checks each predicate to see if it logically implied or falsified by the terms governing it. For example, consider the following solution to the FLATTEN task:

\[
\text{(DEFUN SMASH (L)}
\text{\hspace{1cm} (IF (LISTP L) \hspace{1cm}}
\text{\hspace{1cm} (IF (NULL L) \hspace{1cm} ; Always false when reached.}
\text{\hspace{1cm} NIL}
\text{\hspace{1cm} (APPEND (SMASH (CAR L)) \hspace{1cm} \text{(SMASH (CDR L)))})
\text{\hspace{1cm} (IF (ATOM L) \hspace{1cm} ; Always true when reached.}
\text{\hspace{1cm} (LIST L)}
\text{\hspace{1cm} NIL)))}
\]

Ignoring stylistic bugs, the solution is correct. But the test (NULL L) will always be false when it is reached, and the test (ATOM L) will always be true when it is reached. Talus generates the following conjectures during its tree walk over SMASH to detect redundant conditional tests:

\[
; Is \text{ (LISTP L) always true when reached?}
(\text{IMPLIES T \hspace{1cm} ; Terms Governing (LISTP L)}
\text{ \hspace{1cm} (LISTP L))) \hspace{1cm} ; Predicate \hspace{1cm}
\text{---> F. \hspace{1cm} ; Results of Evaluating Conjecture}
\]
(IS (LISTP L) always false when reached?
  (IMPLIES T
    (NOT (LISTP L)))
  \textit{Terms Governing (LISTP L)}
  \textit{Predicate Negation}
  \textit{Results of Evaluating Conjecture}
  \textit{T.}\n
(IS (NULL L) always false when reached?
  (IMPLIES (LISTP L)
    (NOT (NULL L)))
  \textit{Terms Governing (NULL L)}
  \textit{Predicate Negation}
  \textit{Results of Evaluating Conjecture}
  \textit{T.}\n
(IS (NULL L) always true when reached?
  \textit{Terms Governing (NULL L)}
  \textit{Predicate}
  \textit{Results of Evaluating Conjecture}\n
(IS (ATOM L) always true when reached?
  \textit{Terms Governing (ATOM L)}
  \textit{Predicate}
  \textit{Results of Evaluating Conjecture}\n
Only the last two conjectures are true, indicating that (NULL L) and (ATOM L) are redundant conditional tests. Talus displays the following warnings:

\textbf{EXTRA CONDITION:} \textit{It appears that the test (NULL L) can never be true when it is reached.}\n
\textbf{EXTRA CONDITION:} \textit{It appears that the test (ATOM L) will always be true when it is reached.}\n
6.1.2 Symbolic Evaluation using The Binary Tree Representation

After checking the student function for redundant conditional tests, Talus compares the student function to its paired reference function to detect missing conditional tests and extra conditional tests. Section 6.1.3 provides examples and explains the detection process. The detection process depends on symbolic evaluation, which is discussed in this section.

Recall that functions in IF-Normal Form can be represented as binary trees. This binary tree representation is particularly convenient for symbolic evaluation. Symbolic evaluation becomes a process of determining a path from the root node of the tree to one or more terminal nodes. For example, consider symbolically evaluating the reference function MEMBER:

\begin{verbatim}
(DEFUN MEMBER (ITEM BAG)
  (IF (NLISTP BAG)
    \textit{F}
    (IF (EQUAL ITEM (CAR BAG))
      \textit{T}
      (MEMBER ITEM (CDR BAG)))))
\end{verbatim}

assuming the term:

\begin{verbatim}
(AND (LISTP BAG)
  (EQUAL BAG (LIST ITEM))).
\end{verbatim}

The binary tree representation of MEMBER is depicted in Figure 6-1. Talus follows the left path leading from the root node since
(IMPLIES (AND (LISTP BAG)
               (EQUAL BAG (LIST ITEM)))
         (NOT (NLISTP BAG)))

is true. This path is shown in Figure 6-2. Then the right path is taken from that node to the leaf T, since

(IMPLIES (AND (LISTP BAG)
               (EQUAL BAG (LIST ITEM)))
         (EQUAL ITEM (CAR BAG)))

is true, i.e. if BAG is a list whose sole element is ITEM then ITEM is the first element of BAG. Thus MEMBER symbolically evaluates to T when the term

(AND (LISTP BAG)
      (EQUAL BAG (LIST ITEM))).

is assumed. The complete path from the root node to the tip node representing the s-expression T is shown in Figure 6-3.

Actually there are four possible situations that can occur when Talus reaches a nonterminal node node representing a predicate predicate. Let hypothesis be the terms assumed true for symbolic evaluation. Let assumptions be those terms that govern predicate whose truth or falsehood is not logically implied by hypothesis. These terms need not be computed prior to symbolic evaluation, they
will be collected incrementally whenever case splitting occurs. The initial value of assumptions is T. The four possibilities are:

- **Case 1.** The predicate will always be true. This case applies when the following conjecture is true:

\[(\text{implies} \ (\text{and} \ \text{hypothesis} \ \text{assumptions}) \ \text{predicate})\]

Symbolic evaluation continues by taking the right branch from node, as shown in Figure 6-4.

- **Case 2.** The predicate will always be false. This case applies when the following conjecture is true:

\[(\text{implies} \ (\text{and} \ \text{hypothesis} \ \text{assumptions}) \ \text{(not} \ \text{predicate}))\]

Symbolic evaluation continues by taking the left branch from node, as shown in Figure 6-5.
Case 2: (IMPLIES (AND HYPOTHESIS ASSUMPTIONS)
(NOT PREDICATE))

Figure 6-5: Path Traversal with a False Predicate

- Case 3. The predicate will always cause an error. This case applies when the following conjecture is true:

(IMPLIES (AND hypothesis assumptions)
(ERRORP predicate))

Symbolic evaluation terminates with the distinguished constant error returned, as shown in Figure 6-6.

Case 3: (IMPLIES (AND HYPOTHESIS ASSUMPTIONS)
(ERRORP PREDICATE))

Figure 6-6: A Predicate that Always Causes an Error

- Case 4. None of the above applies. Case splitting is required.

When case splitting occurs, Talus takes both paths from node, as shown in Figure 6-7. When the right branch is taken predicate becomes a new conjunct in assumptions:

assumptions <- (AND predicate assumptions)

When the left branch is taken (NOT predicate) becomes a new conjunct in assumptions:

assumptions <- (AND (NOT predicate) assumptions)
The results of both branches are collected and returned as the result of symbolically evaluating node. In addition to each symbolic value Talus returns the assumptions required to reach that value. These records are used in detecting and inserting missing conditional tests.

---

**Case 4: None of the Above**

![Predicate Diagram](image)

**Figure 6-7: Case Splitting**

---

### 6.1.3 Detecting Missing and Extra Conditional Tests

Talus performs a case analysis to detect missing and extra conditional tests. The terms governing a terminal node are defined to be a case. If a function's binary tree representation has n terminal nodes then there are n cases to consider for that function.

Talus determines cases for both student and reference functions. Consider the student function IN and the reference function it was paired with in the MEMTREE example. The student's definition is:

```
(defun in (x l)
  (if (listp l)
      (if (equal l (list x))
          l
          (if (not (equal (car l) x))
              (in x (car l))
              l))
      nil))
```

and the normalized reference function it was paired with is:

```
(defun in (x l)
  (if (nlistp l)
      f
      (if (equal x (car l))
          t
          (in x (cdr l))))
```

The binary trees of these two functions are shown in Figure 6-8. There are four cases for the student's definition:

- **Case 1:** (not (listp l))

  This term governs leaf \( l_1 \), which represents the s-expression NIL.
• **Case 2:** \((\text{AND} (\text{LISTP } L) \newline \text{(EQUAL } L \text{ (LIST } X)\text{)))\)

This term governs leaf \(l_2\), which represents the s-expression \(L\).

• **Case 3:** \((\text{AND} (\text{LISTP } L) \newline \text{(NOT} \text{ (EQUAL } L \text{ (LIST } X)\text{)))} \newline \text{(NOT} \text{ (EQUAL} \text{ (CAR } L\text{) } X)\text{)))}\)

This term governs leaf \(l_4\), which represents the s-expression \((\text{IN } X \text{ (CAR } L\text{)})\).

• **Case 4:** \((\text{AND} (\text{LISTP } L) \newline \text{(NOT} \text{ (EQUAL } L \text{ (LIST } X)\text{)))} \newline \text{(NOT} \text{ (NOT} \text{ (EQUAL} \text{ (CAR } L\text{) } X)\text{)))}\)

This term governs leaf \(l_5\), which represents the s-expression \(L\).

There are three cases for the normalized reference definition:

• **Case 1:** \((\text{NLISTP } L)\)

This term governs leaf \(l_7\), which represents the s-expression \(F\).

• **Case 2:** \((\text{AND} \text{ (NOT} \text{ (NLISTP } L\text{)))} \newline \text{(EQUAL } X \text{ (CAR } L\text{)\text{)))}\)

This term governs leaf \(l_8\), which represents the s-expression \(T\).

• **Case 3:** \((\text{AND} \text{ (NOT} \text{ (NLISTP } L\text{)))} \newline \text{(NOT} \text{ (EQUAL} \text{ (CAR } L\text{)\text{)))}\)

This term governs leaf \(l_9\), which represents the s-expression \((\text{IN } X \text{ (CDR } L\text{)})\).

Talus detects missing and extra conditions by symbolically evaluating each function using the other function's cases. This process can be viewed as determining mappings from the leaves of one tree to the other. A leaf \(l\) if mapped from tree \(t_1\) to \(t_2\) by symbolically evaluating tree \(t_2\) assuming the terms governing \(l\) in \(t_1\). If \(l\) maps to more than one leaf in \(t_2\) then \(t_2\) contains some conditional test that caused case splitting to occur. This case splitting indicates that \(t_2\) contains some conditional test that has no logical equivalent in the terms governing \(l\) in \(t_1\). Furthermore the conjunction of terms that govern \(l\) neither imply nor falsify this conditional test. When \(t_1\) is the student function and \(t_2\) the reference function this indicates a missing conditional test in the student function. When \(t_1\) is the reference function and \(t_2\) the student function this indicates an extra conditional test in the student function.

An example will clarify the process. Consider mapping the cases of the normalized reference function \(\text{IN}\) to the student's definition of \(\text{IN}\). This will detect any extra conditionals in the student's definition. Let \(t_1\) be the binary tree representation of the normalized reference function \(\text{IN}\) and let \(t_2\) be the binary tree representation of the student's definition of \(\text{IN}\). There are three cases for \(t_1\):

• **Case 1:** \((\text{NLISTP } L)\) ; Governs leaf \(l_7\)

The leaf \(l_7\) maps to the leaf \(l_1\), as shown in Figure 6-9. Talus evaluates the following conjectures to establish this path in \(t_2\):

\(\text{(IMPLIES} \text{ (NLISTP } L\text{) (LISTP } L\text{)\text{)}} \rightarrow F\)
\(\text{(IMPLIES} \text{ (NLISTP } L\text{) (NOT} \text{ (LISTP } L\text{)\text{)}} \rightarrow T\)

• **Case 2:** \((\text{AND} \text{ (NOT} \text{ (NLISTP } L\text{)\text{)))} \text{; Governs leaf } l_6\text{ (EQUAL } X \text{ (CAR } L\text{)\text{)))}\)
Figure 6-8: Binary Tree Representations for two definitions of IN

The leaf $l_6$ maps to two leaves: leaf $l_2$ and leaf $l_3$, as shown in Figure 6-10. Talus evaluates the following conjectures to establish these paths in $t_2$:

(IMPLIES (AND (NOT (NLISTP L))
          (EQUAL X (CAR L)))
       (LISTP L)) $\rightarrow$ T

(IMPLIES (AND (NOT (NLISTP L))
          (EQUAL X (CAR L)))
       (EQUAL L (LIST X))) $\rightarrow$ F

(IMPLIES (AND (NOT (NLISTP L))
          (EQUAL X (CAR L)))
       (NOT (EQUAL L (LIST X)))) $\rightarrow$ F

;Case Splitting Occurs Here

Case splitting occurs in $t_2$ on the predicate (EQUAL L (LIST X)) since the conjunction of terms governing the leaf $l_6$ in $t_1$ neither imply nor falsify (EQUAL L (LIST X)).
;Right Branch

(IMPLIES (AND (NOT (NLISTP L))
    (EQUAL X (CAR L)))
    (EQUAL L (LIST X))) ;Assumption
(EQUAL L (LIST X)) --> T

;Left Branch

(IMPLIES (AND (NOT (NLISTP L))
    (EQUAL X (CAR L))
    (NOT (EQUAL L (LIST X)))) ;Assumption
(NOT (EQUAL L (LIST X)))) --> T

(IMPLIES (AND (NOT (NLISTP L))
    (EQUAL X (CAR L))
    (NOT (EQUAL L (LIST X)))) ;Assumption
(NOT (EQUAL L (LIST X)))) --&gt; F

(IMPLIES (AND (NOT (NLISTP L))
    (EQUAL X (CAR L))
    (NOT (EQUAL L (LIST X)))) ;Assumption
(NOT (EQUAL L (LIST X)))) --> T

*Case 3: (AND (NOT (NLISTP L)) ; Governs leaf l3
    (NOT (EQUAL X (CAR L))))

The leaf l3 maps to the leaf l4, as shown in Figure 6-11. Talus evaluates the following conjectures to establish the path in l2:

(IMPLIES (AND (NOT (NLISTP L))
    (NOT (EQUAL X (CAR L))))
    (LISTP L)) --> T

(IMPLIES (AND (NOT (NLISTP L))
    (NOT (EQUAL X (CAR L))))
    (EQUAL L (LIST X))) --&gt; F

(IMPLIES (AND (NOT (NLISTP L))
    (NOT (EQUAL X (CAR L))))
    (NOT (EQUAL L (LIST X)))) --> T

;No case splitting required.

(IMPLIES (AND (NOT (NLISTP L))
    (NOT (EQUAL X (CAR L))))
    (NOT (EQUAL (CAR L) X))) --&gt; T

The case splitting that occurs when leaf l6 in l1 is mapped to two leaves in l2 indicates an extra conditional expression in the student's code. l6 maps to leaf l2 with the forced assumption that (EQUAL L (LIST X)) is true. It also maps to leaf l3 with the forced assumption that (NOT (EQUAL L (LIST X))) is true. The assumptions forced by case splitting indicate that the conditional test (EQUAL L (LIST X)) is unnecessary in the student's code. Talus generates the following warning after mapping cases:
Figure 6-9: Symbolically Evaluating IN, Case 1

NOTE: Extra condition in student's code reached when
(NOT (NLISTP L)) is true, and
(EQUAL X (CAR L)) is true.
The extra condition is (EQUAL L (LIST X)).

Missing conditions are detected when the direction of mapping is reversed. Assume that the student's definition of IN was the normalized reference function definition and vice versa. Then, using these swapped definitions, the direction of the previous mapping would be from student function to reference function. The case splitting that occurred for Case 2 now indicates a missing condition in the student's code, i.e. with the revised reference function the student's function should have a conditional test (EQUAL L (LIST X)). Of course this hypothetical scenario only illustrates the detection of missing conditional tests since reference functions should always be correct with respect to both stylistic and nonstylistic bugs; swapping the function definitions invalidates the assumption.

Extra conditions are considered stylistic bugs but missing conditions are considered nonstylistic bugs that must be corrected. If the student's function is missing a conditional test then Talus inserts that conditional test in the student's function before generating verification conditions. Chapter 7 explains how missing conditional tests are inserted. Talus only issues warnings for extra conditional tests, but it does not remove them. However, the symbolic values governed by extra conditionals will still be
checked for bugs, and bugs in those symbolic values will be corrected.

Sometimes leaves cannot be mapped since predicates evaluate to \textit{error} during the symbolic evaluation process. Suppose that leaf \( l \) from tree \( t_1 \) maps to \textit{error} in tree \( t_2 \). Then \( t_2 \) is missing some conditional test to filter out data outside the domain of some primitive. Since the reference function is assumed correct, this should only happen when \( t_1 \) is the reference function and \( t_2 \) the student function. For example, consider the MEMBER task first presented in Section 3.4.3. Assume that the student's buggy solution is:

\[
\begin{align*}
\text{(DEFUN MEM (X L)} \\
\text{  (IF (EQUAL X (CAR L))} \\
\text{    T} \\
\text{    (MEM X (CDR L)))))}
\end{align*}
\]

and that the normalized reference definition is:
(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (MEM X (CDR L)))))

The two binary tree representations of these functions are shown in Figure 6-12. When Talus attempts to map reference cases to the student's function, the first case to be considered is (NLISTP L). Talus detects that the predicate (EQUAL X (CAR L)) will always cause an error since the following conjecture is true:

(IMPLIES (NLISTP L) (ERRORP (EQUAL X (CAR L))))

This situation is shown in Figure 6-13. Talus generates the following bug report:

BUG: Missing condition. When (NLISTP L) is assumed, the IF-test (EQUAL X (CAR L)) always causes an error. The missing condition is (NLISTP L).

Chapter 7 explains how Talus corrects this kind of missing conditional test.

'Mappings must occur in both directions to detect all anomalies in the student's conditions. Talus
maps from student to stored code to detect missing conditional expressions in the student's code that do not cause errors. Talus maps from stored to student code to detect extra conditional expressions in the student's code and any missing conditional expressions that cause errors.

6.2 Detecting Bugs with Verification Conditions

Talus generates two kinds of verification conditions: functional equivalence verification conditions and termination verification conditions. The first kind of verification condition checks that the student and reference functions are equivalent for a particular case, assuming that the student function will be
correctly debugged for the remaining cases. The second kind of verification condition checks that recursive calls do not lead to infinite loops. This section provides examples of these verification conditions and then explains how they are generated.

Both kinds of verification conditions will be illustrated by examining the conjectures generated while debugging the student function IN. IN first appeared in the MEMTREE case study and was used by Section 6.1.3 to illustrate the detection of extra conditional tests. This section continues the detailed study of the debugging of IN. The functional equivalence verification conditions for IN are:

\[(\text{IMPLIES} \ (\text{NLISTP} \ L)) \quad ; \text{Case 1, from normalized reference code} \]
  \[(\text{IFF} \quad ; \text{Check logical equality} \]
  \[\quad \text{F} \quad ; \text{Symbolic value from reference function, leaf l7} \]
  \[\quad \text{NIL}) \quad ; \text{Symbolic value from student function, leaf l1} \]
  \[\quad \rightarrow \ T \quad ; \text{Results of evaluating conjecture} \]

\[(\text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ L)))) \quad ; \text{Case 2} \]
  \[(\text{EQUAL} \ X \ (\text{CAR} \ L))) \quad ; \text{Assumption} \]
  \[\quad (\text{EQUAL} \ L \ (\text{LIST} \ X))) \quad ; \text{Assumption} \]
  \[(\text{IFF} \ T \ L)) \quad ; \text{Compare l6 and l2} \]
  \[\quad \rightarrow \ T \]

\[(\text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ L)))) \quad ; \text{Case 2} \]
  \[(\text{EQUAL} \ X \ (\text{CAR} \ L))) \quad ; \text{Assumption} \]
  \[\quad (\text{NOT} \ (\text{EQUAL} \ L \ (\text{LIST} \ X))) \quad ; \text{Assumption} \]
  \[(\text{IFF} \ T \ L)) \quad ; \text{Compare l6 and l3} \]
  \[\quad \rightarrow \ T \]

\[(\text{IMPLIES} \ (\text{AND} \ (\text{NOT} \ (\text{NLISTP} \ L)))) \quad ; \text{Case 3} \]
  \[(\text{NOT} \ (\text{EQUAL} \ X \ (\text{CAR} \ L))) \quad ; \text{Compare l5 and l4} \]
  \[(\text{IFF} \ (\text{IN} \ X \ (\text{CDR} \ L)) \quad ; \text{Compare l5 and l4} \]
  \[\quad (\text{IN} \ X \ (\text{CAR} \ L))) \quad ; \text{Compare l5 and l4} \]
  \[\quad \rightarrow \ F \]

No termination verification conditions are generated since the falsehood of the last verification condition is sufficient to show that the only code fragment containing a recursive call, (IN X (CAR L)), is buggy.

The only termination verification conditions generated in the MEMTREE case study are for the function FLAT, whose definition is repeated below:

\[(\text{DEFUN} \ \text{FLAT} \ (\text{ANS} \ TR) \]
  \[(\text{IF} \ (\text{ATOM} \ TR) \]
  \[\quad \text{ANS} \]
  \[\quad (\text{FLAT} \ (\text{FLAT} \ ANS \ (\text{CDR} \ TR))) \]
  \[\quad (\text{CAR} \ TR)))) \]

The termination verification conditions for FLAT, both of which are true, are:

\[(\text{IMPLIES} \ (\text{NOT} \ (\text{ATOM} \ TR)) \]
  \[(\text{LESSLT} \ (\text{COUNT} \ (\text{CAR} \ TR)) \ (\text{COUNT} \ TR))))) \]

\[(\text{IMPLIES} \ (\text{NOT} \ (\text{ATOM} \ TR)) \]
  \[(\text{LESSLT} \ (\text{COUNT} \ (\text{CDR} \ TR)) \ (\text{COUNT} \ TR))))) \]
6.2.1 Functional Equivalence Verification Conditions

This section explains how functional equivalence verification conditions are generated and used in bug detection. Two terms \( x \) and \( y \) are functionally equivalent if:

1. \((\text{IFF } x \; y)\) is true and the reference function is a predicate,

2. \((\text{PERMUTATIONP } x \; y)\) is true and the reference function returns lists that represent sets, or

3. \((\text{EQUAL } x \; y)\) is true.

\(\text{IFF} \) is true only when both \( x \) and \( y \) are false or true simultaneously. \(\text{PERMUTATIONP} \) is true when both \( x \) and \( y \) are lists that are permutations of each other. Reference functions have properties indicating whether or not they operate as predicates or set constructors.\(^{17}\) Talus decides which predicate should test for functional equality based on these properties.

Suppose Talus is comparing a student function \( \text{buggy} \) and a reference function \( \text{ref} \) and each has \( n \) leaves. Assume the reference function is a predicate and that any missing conditional tests in the student's function have already been detected and corrected. Then Talus checks the following functional equivalence verification conditions:

\[
\text{For } i \text{ from } 1 \text{ to } n \text{ check }
\]

\[
(\text{IMPLIES } \text{ref-case}_i \\
(\text{IFF } \text{ref-code}_i \; \text{buggy-code}_i))
\]

where

- \( \text{ref-case}_i \) = The \( i \)th case in the reference function.
- \( \text{ref-leaf}_i \) = The leaf in the binary tree of \( \text{ref} \) that is governed by the terms in \( \text{ref-case}_i \).
- \( \text{ref-code}_i \) = The s-expression represented by \( \text{ref-leaf}_i \).
- \( \text{buggy-leaf}_i \) = The leaf that \( \text{ref-leaf}_i \) maps to in the binary tree representation of \( \text{buggy} \).
- \( \text{buggy-code}_i \) = The s-expression in the student function represented by \( \text{buggy-leaf}_i \).

If the reference function returns sets, as would occur in the SET-INTERSECTION task, then the functional equivalence verification conditions would be:

\[
(\text{IMPLIES } \text{ref-case}_i \\
(\text{PERMUTATIONP } \text{ref-code}_i \; \text{buggy-code}_i))
\]

By using \(\text{PERMUTATIONP}\) instead of \(\text{EQUAL}\), Talus can accept the following definition as a correct implementation of set intersection:

\(^{17}\)These properties are stored in global variables in Talus but they can be considered part of the task representation.
even though INTERSECT2 reverses the order of elements as they are collected.

In general, more than one verification condition can be generated when mapping ref-leaf: Assume that ref-leaf_i maps to c_i leaves in the student’s code: buggy-leaf_{i,1}... buggy-leaf_{i,c_i}. Since case splitting occurred each of these leaves buggy-leaf_{i,j} in the student’s functions has forced assumptions assumptions_{i,j} associated with it. These are the terms governing buggy-leaf_{i,j} that are not logically implied by ref-case_i. With this generalization the bug detection algorithm of Talus is:

\[
\begin{align*}
\text{For } i = 1 \text{ to } n & \text{ do } \quad \text{For each reference case} \\
\text{For } j = 1 \text{ to } c_i & \text{ do } \quad \text{For each leaf it maps to} \\
\text{:Check Functional Equivalence Verification Conditions:} & \\
\text{if } \text{theoremp}[ (\text{IMPLIES (AND ref-case}_i \text{ assumptions}_{i,j}) \text{ (equality-predicate ref-code}_i \text{ buggy-code}_{i,j}) ] & \text{ then continue} \\
\text{else} & \text{ Fix bugs with normalized reference code:} \\
& \text{correct[buggy-code}_{i,j} \text{, ref-code}_i \text{, (AND ref-case}_i \text{ assumptions}_{i,j}) ] .}
\end{align*}
\]

The function theoremp evaluates conjectures and will be discussed in Section 6.3. The function correct replaces parts of the student code with normalized reference code so that the verification condition will be true for the altered code. The term equality-predicate is IFF if the reference function is a predicate, PERMUTATIONP if it is a set constructor, or EQUAL otherwise.

### 6.2.2 Termination Verification Conditions

This algorithm is sufficient to explain all the debugging that occurred in the MEMTREE case study in Section 3.4.1. But it will also accept the following as a correct definition of IN:

\[
\begin{align*}
\text{(DEFUN IN } & (X  L) \text{)} \\
\text{(IF} & (\text{NILSTP } L) \\
\text{IN } & X  L) \\
\text{(IF} & (\text{EQUAL } X  (\text{CAR } L)) \\
\text{(IN} & X  L) \\
\text{(IN} & X L))
\end{align*}
\]

since IN is bound to a correct definition while conjectures are being evaluated. This is clearly undesirable.
since the IN defined above loops forever.

Each symbolic value is debugged assuming that the others will also be correctly debugged. As the example above illustrates, this can only be justified if we can establish that no recursive calls in the student’s function lead to infinite loops. Talus checks for nontermination with termination verification conditions. Each verification condition checks that the arguments in a recursive call diminish according to some measure and the well-founded relation LESSP. The measure is obtained from the task representation.

Two auxiliary functions will be needed to generate termination verification conditions. The function recursions returns a list of recursive calls in an s-expression taken from the student’s code. For example, consider the function FLAT that is part of the buggy solution to MEMTREE:

```lisp
(DEFUN FLAT (ANS TR)
  (IF (ATOM TR) ANS
       (FLAT (FLAT ANS (CDR TR))
             (CAR TR))))
```

Here are the results of applying the function recursions to the symbolic values of FLAT:

```
recursions [ANS] = ()
```

```
recursions [(FLAT (FLAT ANS (CDR TR)) (CAR TR))]
```

```
= ((FLAT (FLAT ANS (CDR TR)) (CAR TR)) ; First Recursive Call
    (FLAT ANS (CDR TR))) ; Second Recursive Call
```

The measure for the reference function FLATTEN is (COUNT TREE). This is normalized to (COUNT TR) during the normalization process. The second auxiliary function updates replaces the formal variables in the measure by their actual arguments in a call. It needs to know the order of variables in the variable list to do this. Here are two examples demonstrating the function updates:

```
updates [(FLAT (FLAT ANS (CDR TR)) (CAR TR)), ; Call
         (COUNT TR), ; Measure
         (ANS TR)] ; Formals
```

```
= (COUNT (CAR TR))
```

```
updates [(FLAT ANS (CDR TR))
         (COUNT TR), ; Measure
         (ANS TR)] ; Formals
```

```
= (COUNT (CDR TR))
```

With these two functions the procedure shown in Figure 6-14 checks termination verification conditions. Note that the procedure is a predicate: T is returned when all termination verification conditions are satisfied, otherwise F is returned. The call

```
check-terminations [buggy-code_{i,j}, (AND ref-case_{i} assumptions_{i,j})]
```

checks the termination verification conditions for the s-expression buggy-code_{i,j} reached by symbolically evaluating the student’s code assuming ref-case_{i} and assumptions_{i,j}. In the MEMTREE case study the
following conditions are true when the reference case (NOT (ATOM TR)) is mapped to the student's function FLAT:

\[ ref-case_2 = (\text{NOT (ATOM TR)}) \]

\[ buggy-code_{2,1} = (\text{FLAT FLAT ANS (CDR TR)}) (\text{CAR TR})) \]

\[ assumptions_{2,1} = \text{T ;No forced assumptions} \]

\[ measure = (\text{COUNT TR}) \]

\[ formals = (\text{ANS TR}) \]

The following termination verification conditions are generated:

\[ (\text{IMPLIEDS (NOT (ATOM TR))}) \]

\[ (\text{LESSP (COUNT (CAR TR)) (COUNT TR))) \]

\[ (\text{IMPLIEDS (NOT (ATOM TR))}) \]

\[ (\text{LESSP (COUNT (CDR TR)) (COUNT TR))) \]

Both conjectures are true so check-terminations would return \( \text{T} \).

\[ \text{check-terminations [code, assumptions]} = \]

;Check Termination Verification Conditions:
(For call in recursions [code]
  do
    (if theoremp
        [ (IMPLIEDS assumptions
            (LESSP measure
             updates [call, measure, formals]))]
        then continue
        else (return \text{F}))
    finally (return \text{T}))

---

**Figure 6-14: The Predicate that Checks Termination Verification Conditions**

Figure 6-15 presents the complete bug detection algorithm that Talus uses to detect bugs at the implementation level. This algorithm assumes that any missing conditional tests have already been inserted into the student's function definition. Both functional equivalence and termination verification conditions are generated. \textit{theoremp} evaluates conjectures and \textit{correct} replaces parts of student code with normalized reference code to enforce verification conditions.

The relationship between the debugging algorithm and an induction proof of functional equivalence is worth reemphasizing. The base steps in the induction proof correspond to those functional equivalence verification conditions generated for reference cases that govern function terminations. No termination verification conditions are generated since there are no recursive calls in the function terminations. Induction steps correspond to the functional equivalence and termination verification conditions generated for those cases in the reference function that govern recursive calls.
For $i = 1$ to $n$ do 
    For $j = 1$ to $c_i$ do 
        ;For each reference case
        ;Check Functional Equivalence Verification Conditions:
        if theorem?( (IMPLIES (AND ref-case$_i$ assumptions$_j$) 
                       (equality-predicate ref-code$_i$
                       buggy-code$_{ij}$) ) ) ]

and

;Check Termination Verification Conditions:
check-terminations [ buggy-code$_{ij}$, (AND ref-case$_i$ assumptions$_j$) ]
then continue
else
;Fix bugs with normalized reference code:
correct [buggy-code$_{ij}$, ref-code$_i$, (AND ref-case$_i$ assumptions$_j$) ] .

---

Figure 6-15: The Bug Detection Algorithm of Talus

The relationship between normalization and the use of induction hypotheses is subtle. Suppose we have a student function buggy and an unnormalized reference function ref that both have the same variable list formals. Consider a case ref-case governing a symbolic value in ref that contains a recursive call; call this code fragment ref-code. Assume that ref-code maps to buggy-code in buggy. For an induction proof of computational equivalence to succeed the following must be true:

; Verification Condition VC

(IMPLIES ref-case (EQUAL buggy-code ref-code))

Our induction hypotheses will establish that each recursive call (buggy ...updates$_i$...) in buggy-code will be EQUAL to (ref ...updates$_i$...), provided that we can show that some measure applied to updates$_i$ decreases by a well-founded relation compared to formals when case is assumed. That is the role of the termination verification conditions. The induction hypotheses will be

(EQUAL (buggy ...updates$_i$,...) (ref ...updates$_i$,...))

for $i = 1$ to $n$ where $n$ is the number of recursive calls in buggy-code. Now instantiate the ref-code, not buggy-code. Then the verification condition VC becomes:

(IMPLIES case (EQUAL buggy-code ref-code [buggy/ref]) )

where term$[f_1/f_2]$ means the results of replacing the function identifiers $f_1$ by $f_2$ in term. But this is exactly the same verification condition that would result if normalized reference code had been used in VC instead of unnormalized reference code.

---

$^{18}$The use of equality hypotheses to perform substitutions followed by their removal from a conjecture's hypothesis is called cross-fertilization by Boyer and Moore [Boyer 79].
For normalization to be equivalent to the instantiation of induction hypotheses it is also necessary for calls to buggy to refer to the normalized reference code, rather than the buggy student code, during conjecture evaluation. For example in evaluating the functional equivalence verification condition

\[(\text{IMPLIES} \; (\text{NOT} \; (\text{ATOM} \; \text{TR}))) \; (\text{EQUAL} \; (\text{APPEND} \; (\text{FLAT} \; (\text{CAR} \; \text{TR}))) \; (\text{FLAT} \; (\text{CDR} \; \text{TR}))) ; \; \text{Reference Code} \)\]

\[(\text{EQUAL} \; (\text{FLAT} \; (\text{CAR} \; \text{TR}))) \; (\text{FLAT} \; (\text{CDR} \; \text{TR}))) \; \text{From Student Code} \]

the function FLAT refers to the normalized reference code, even in the code fragment extracted from the student's code. It does not refer to the student's buggy definition of FLAT. If our original unnormalized reference function was FLATTEN then the verification condition above is equivalent to:

\[(\text{IMPLIES} \; (\text{AND} \; (\text{EQUAL} \; (\text{FLATTEN} \; (\text{CAR} \; \text{TR}))) \; (\text{FLAT} \; (\text{CAR} \; \text{TR}))) \; \text{Induction Hypothesis} \)
\[(\text{EQUAL} \; (\text{FLATTEN} \; (\text{CDR} \; \text{TR}))) \; (\text{FLAT} \; (\text{CDR} \; \text{TR}))) \; \text{Induction Hypothesis} \)
\[(\text{NOT} \; (\text{ATOM} \; \text{TR}))) \; \text{Case} \)
\[(\text{EQUAL} \; (\text{APPEND} \; (\text{FLATTEN} \; (\text{CAR} \; \text{TR}))) \; (\text{FLATTEN} \; (\text{CDR} \; \text{TR}))) ; \; \text{Reference Code} \)
\[(\text{CONS} \; (\text{FLAT} \; (\text{CAR} \; \text{TR}))) \; (\text{FLAT} \; (\text{CDR} \; \text{TR}))) \; \text{From Student Code} \]

In the conjecture above the induction hypotheses have not yet been instantiated so that the relationship between normalization and the use of induction hypotheses can be more clearly seen.

When a verification condition is invalid, Talus corrects the student's code by minimally altering the student's code so that the verification condition becomes a theorem. Essentially, Talus attempts to verify the student's program using the stored function both as its specification and as a source of corrections. Debugging consists of enforcing the verification conditions when necessary.

### 6.3 Evaluating Conjectures

The reader may be troubled by the function theoremp. Since the Boyer-Moore Logic is undecidable theoremp cannot reliably return T or NIL since then theoremp would be a decision procedure for the logic. But we can construct a version of theoremp that only returns T on theorems - this theoremp simply calls the Boyer-Moore Theorem Prover. NIL is returned on nontheorems and theorems that cannot be proven without the introduction of lemmas or further guidance in the use of existing lemmas. Section 6.3.3 discusses the implementation of theoremp with the Boyer-Moore Theorem Prover. With this version of theoremp the debugging algorithm in Figure 6-15 corrects all bugs in the student's program with respect to the reference function and measure in the task representation. However false alarms occur when theoremp returns NIL when it should return T. In that case student code fragments are unnecessarily replaced by stored code fragments.

The conjecture disprover is a different version of theoremp that only returns NIL on nontheorems. T is returned on conjectures that are theorems and conjectures that cannot be disproven. No false alarms occur when the conjecture disprover implements theoremp in the debugging algorithm in Figure 6-15. The conjecture disprover works by instantiating free variables in conjectures with stored examples. F is returned if a counterexample to a conjecture is discovered, otherwise T is returned. With this approach, Talus can always provide a counterexample for each failed verification condition. This counterexample
could be used by an ITS to demonstrate a program's buggy behaviour. However, bugs may be missed with this approach if no counterexamples can be found for invalid conjectures. Section 6.3.1 discusses this means of conjecture evaluation. Talus can also call a more advanced conjecture disposer called the counterexample generator. The counterexample generator applies heuristics to actively generate counterexamples. Section 6.3.2 discusses the counterexample generator.

Both the conjecture disposer and the counterexample generator can act as filters to the Boyer-Moore Theorem Prover (see Figure 6-16). In Talus, theorem p is implemented so that any combination of these three means of conjecture evaluation can be applied. In practice the conjecture disposer is most frequently used by itself since it is fast and generates no false alarms. For tasks the size of MEMTREE or smaller, the Boyer-Moore Theorem Prover performs quite well. Performance deteriorates for larger tasks such as SINGLETONS since additional lemmas must be introduced to prove that valid conjectures are theorems and to prove that reference functions can be admitted under the Principle of Definition.

6.3.1 The Conjecture Disprover

Counterexamples are stored sets of bindings of formal variables for each reference function in a stored task algorithm. For example, the counterexamples stored for the reference function FLATTEN are:

```lisp
((TREE . (A)))
((TREE . (A . B)))
((TREE . (NIL)))
((TREE . (NIL X)))
((TREE . (A B C)))
((TREE . Q))
((TREE . ((1) (2 3) ((4)))))
```

Counterexamples will be normalized during the algorithm recognition process. So if the identifier TREE is paired with the identifier TR from the student's code, then the counterexamples will be normalized to:

```lisp
((TR . (A)))
((TR . (A . B)))
((TR . (NIL)))
((TR . (NIL X)))
((TR . (A B C)))
((TR . Q))
((TR . ((1) (2 3) ((4)))))
```
If a conjecture evaluates true for all counterexamples then it is believed, otherwise it is definitely false. For each counterexample, Talus binds the free variables in the conjecture to their values in the counterexample and then evaluates the conjecture. Talus does not call the LISP evaluator EVAL, instead it calls its own evaluator that implements the core LISP dialect of Talus.

When conjectures are evaluated student function identifiers refer to correct function definitions obtained from the normalized reference functions. Student functions mapped to EXTRA are defined as in the student's solution since Talus has no paired reference function to supply a correct definition. These functions are evaluated with a clock to prevent infinite looping. If an excessive (e.g. greater than 100) calls to the evaluator are required then error is returned.

No error breaks can occur in evaluation. Instead error is returned if a primitive is applied to a data type outside of its domain. For example, CAR of NIL is error and SUB1 of 0 is error. All functions are strict: if any of their arguments is error the function returns error.

6.3.2 The Counterexample Generator
A more thorough means of refuting a conjecture is to generate counterexamples. To generate counterexamples for conjecture, Talus calls the EGS system [Kim 85]. The EGS system attempts to construct an example for (NOT conjecture). The example generator works by unfolding functions, merging stored examples, and reducing the conjecture by transformations and in the process building up examples. The EGS system was originally developed by Kim to increase the efficiency of the Boyer-Moore Theorem Prover. EGS is intended to reduce the search space for the correct proof of a conjecture by eliminating fruitless attempts to prove invalid conjectures.

6.3.3 The Boyer-Moore Theorem Prover
The Boyer-Moore Theorem Prover attempts to prove conjectures by rewriting terms with existing lemmas, by using information about term data types, and by using heuristics for generalizing variables and for induction. Induction schemas are generated when recursive functions are defined. Heuristics choose the most appropriate induction schemas and substitutions to prove a conjecture by induction. Induction is a last resort applied when simpler means fail.

Talus implements theoremp with the Boyer-Moore Theorem Prover by:

- **Defining primitives present in the pure LISP dialect of Talus.** For example, APPEND is not a primitive in the Boyer-Moore Logic and must be defined.

- **Defining all normalized reference functions prior to the evaluation of conjectures.** For example, FLAT must be defined before any conjecture involving FLAT can be evaluated.

- **Limiting the amount of resources that can be spent on a proof attempt.** The Boyer-Moore Theorem Prover has been modified so that the proof aborts after \( n \) inductions have been attempted. Currently \( n \) is set to five.

- **Defining ERRORP.** Since CAR and SUB1 return 0 when given illegal data types ERRORP is currently defined just to be ZEROP. A more rigorous treatment of error handling should define (ERROR) as a new data type.

- **Converting predicate logics.** Predicates in the Boyer-Moore Logic return F or non-F. Predicates in the pure LISP dialect of Talus and in most LISP implementations return NIL or non-NIL. Talus accounts for these differences when definitions or conjectures are passed to the Boyer-Moore Theorem Prover.
• Adding implicit data restrictions. Most functions operate correctly only on certain kinds of data. These domain restrictions must be made explicit when proving conjectures about these functions.

Implicit data restrictions are restrictions on the data that will be presented to a function that are a result of either the task specification or the function’s role in an algorithm implementation. For example a task description can specify that only natural numbers need be considered, as in the FACTORIAL task, or that only proper lists need be considered, as in the REVERSE task. For an example where a function’s role restricts the data types of its formal variables, consider the reference function MEMBER in the MEMTREE case study. Its second formal variable, BAG, will always be a proper list since the only actual argument for BAG is (FLATTEN TREE) and FLATTEN returns proper lists.

No special provisions need to be made for implicit data restrictions when only the conjecture disprover is used. Talus assumes that all stored counterexamples for a task obey the data restrictions for that task. Thus the examples for FACTORIAL should only be numbers and the examples for REVERSE should only be proper lists. However, the theorem prover requires these restrictions to be explicitly stated. Suppose REVERSE has its standard definition:

\[
\text{(DEFUN REVERSE} \ (X) \\
\text{ (IF} \ (\text{NILP} \ X) \\
\text{ NIL} \\
\text{ (APPEND} \ \text{(REVERSE} \ \text{CDR} \ X) \\
\text{ (LIST} \ \text{(CAR} \ X)))))
\]

then we might expect the following to be true:

\[
\text{(IMPLIES} \ \text{(LISTP} \ L) \\
\text{ (EQUAL} \ L \ \text{(REVERSE} \ \text{(REVERSE} \ L)))
\]

but it is not. Any CONS ending in a non-NIL atom, such as (A . B), provides a counterexample. When we restrict valid data to proper lists, i.e. NIL or CONSes ending in NIL, then reverse is its own inverse and the following conjecture can be proven correct:

\[
\text{(IMPLIES} \ \text{(PLISTP} \ L) ; L is NIL or a CONS ending in NIL \\
\text{ (EQUAL} \ L \ \text{(REVERSE} \ \text{(REVERSE} \ L)))
\]

Talus stores implicit data restrictions as part of its task representation. In the MEMTREE case study the implicit data restrictions for the reference function MEMBER are:

\[
\text{(AND} \ \text{(PLISTP} \ \text{BAG}) \ \text{(NOT} \ \text{(LISTP} \ \text{ITEM})))
\]

During algorithm recognition the student function IN, with formal variables X and L is paired to the reference function MEMBER and its formal variables ITEM and BAG. The implicit data restriction is normalized along with the reference functions. It becomes:

\[
\text{(AND} \ \text{(PLISTP} \ L) \ \text{(NOT} \ \text{(LISTP} \ \text{X})))
\]

All conjectures passed to theorem will have implicit data restrictions added to their hypotheses. Suppose the student has correctly defined IN as follows:
(DEFUN IN (X L)
  (IF (NULL L)
    F
    (IF (EQUAL X (CAR L))
      T
      (IN X (CDR L)))))

When Talus is checking for extra conditional tests it will symbolically evaluate IN assuming the reference case (NLISTP L). Talus will pass the following conjecture to theoremp:

(IMPLIES (NLISTP L) (NULL L))

Theoremp will add the implicit data restrictions to the conjecture and pass the following conjecture to the Boyer-Moore Theorem Prover:

(IMPLIES (AND (NLISTP L) ;L is an atom.
  (PLISTP L) ;L is a proper list.
  (NOT (LISTP X))) ;X is not a CONS.
  (NULL L))

This conjecture will be proven to be a theorem since if L is an atom and if L is a proper list then it must be the atom NIL since that is the only atomic proper list. Note that

(IMPLIES (NLISTP L) (NULL L))

is false since L could be some non-NIL atom; thus implicit data restrictions are crucial when the Boyer-
Moore Theorem Prover implements theoremp.

6.4 Alternate Bug Detection Strategies

This section considers alternate approaches to bug detection. Talus uses a program verification approach where reference function definitions act as specifications. The Programmer's Apprentice can also detect bugs by a program verification methodology. Programs are represented using the PLAN [Rich 81] representation. A plan calculus formalizes the semantics of plans. Program correctness is checked by proving that the input specifications of a plan imply the output specifications [Shrobe 79]. When a plan cannot be proven to meet its specifications, a bug may be present in the plan.

The Programmer's Apprentice is intended to aid expert users, not students or novice programmers. It was not designed to support intelligent tutoring systems. As discussed in Section 4.6, the algorithm recognition component does not appear amenable to recognizing algorithms in buggy student programs. Another disadvantage when considering the Programmer's Apprentice for intelligent tutoring systems is its inability to correct bugs. The Programmer's Apprentice can only notify the user of inconsistencies between plans and specifications; it expects the user to resolve the conflicts.

Another means of detecting bugs is to set up demons that detect specific bugs during or after a program's execution. For example, SNIFTER [Shapiro 81] relies on bug specialists called sniffers to look for specific bugs in a program's execution history. If any bugs are found the sniffers can provide highly specific diagnosis. The main drawback with this approach is the inadequate coverage of the sniffers. If no sniffers fire, there can still be bugs in the student's program that are unknown to the sniffers. Since this approach relies on a program's execution history, many stylistic errors will be undetectable and program context surrounding nonstylistic bugs will be lost. This is undesirable when this means of bug detection is incorporated in a complete ITS that relies on the context surrounding a bug to infer misconceptions.
The bug specialists of SNIFFER detect bugs that are rational form violations, i.e. the bug specialists do not take into account the semantics of tasks. What may be a bug for one task may be correct for another task, but in SNIFFER task information is not incorporated. Bug specialists for each task can be implemented but this requires the curriculum author to enumerate the expected bugs for each task and to code bug specialists for each bug.

Bugs can also be detected by interpreting match failures as bugs. Debuggers that follow the heuristic plan recognition approach interpret plan match failures as possible symptoms of bugs. However, unanticipated implementations can also be interpreted as bugs, resulting in false alarms. Plan-difference rules [Johnson 84] that recognize common implementation variants and that can recognize specific kinds of bugs mitigate, but do not solve the problem. Code that cannot be accounted for by plan-difference rules remains problematic.

None of the debuggers that adopt a heuristic plan recognition approach actually correct code. The problem with false alarms, inability to handle unanticipated implementations, and an inability to correct code are all due to restricted capabilities to reason about computational semantics. Most knowledge about computational semantics is hard-wired into plan-difference rules and the plan database. Talus, in contrast, emphasizes dynamic reasoning about computational semantics during the debugging process through its use of conjecture evaluation.

A final means of bug detection to consider is trace analysis. Discrepancies between an actual and expected execution trace indicate bugs. No bug specialists are required. This bug detection approach enables PDS6 [Shapiro 83] to detect bugs in pure PROLOG when a user correctly answers all questions about the correctness of traces. However, the trace analysis approach of PDS6 does not detect stylistic bugs, such as dead or superfluous code, and does not provide for the debugging of programs with side effects. Furthermore, since programs are only debugged with respect to the examples provided by the user, bugs may remain undetected due to the choice of examples.

6.5 Summary

The key points of this chapter are:

- The binary tree representation facilitates symbolic evaluation. Functions in IF-Normal Form can be symbolically evaluated by representing the functions as binary trees and then determining paths from the root node to terminal nodes.
- Symbolic evaluation has two purposes in Talus:
  1. To detect redundant, missing, and extra conditional tests in student code, and
  2. To allow bugs to be isolated to code fragments by decomposing the debugging of a function's body into disjoint cases.
- Student and reference functions need only be functionally equivalent. Predicates need only return NIL and non-NIL values under the same conditions. Set constructors need only return values that are permutations of each other.
- There are two kinds of verification conditions:
  1. Functional equivalence verification conditions, to check for functional equivalence.
  2. Termination verification conditions, to check for termination.
Conjectures can be evaluated by:

- The Conjecture Disprover. The conjecture disprover searches for a counterexample to a conjecture among examples stored in the task representation.

- The Counterexample Generator. The counterexample generator attempts to construct a counterexample to a conjecture by applying heuristics.

- The Boyer-Moore Theorem Prover. The Boyer-Moore Theorem Prover applies heuristics to search for a formal proof of correctness of a conjecture.

- Any combination of the above.
Chapter Seven

Bug Correction

This chapter explains how Talus corrects bugs. Talus can correct bugs at three levels of program design. These levels are:

1. **The Algorithm Level.** The design of the task solution. Section 7.1 explains how bugs are corrected at the algorithm level.

2. **The Function Level.** The functional decomposition of the chosen algorithm. Section 7.2 explains how bugs are corrected at the function level.

3. **The Implementation Level.** The code that implements each function. Section 7.3 explains how bugs are corrected at the implementation level.

Warnings or commentary are generated for stylistic bugs at all three levels. Most nonstylistic bugs are corrected by replacing student code by normalized reference code. The next section considers bugs at the highest level of program design, the algorithm level.

7.1 Inefficient and Buggy Algorithms

The student's algorithm can be either correct, buggy, or inefficient. Correct algorithms solve the assigned task when correctly implemented. Correct implementations of buggy algorithms solve some task closely related to the one assigned, but they do not solve the assigned task. Buggy algorithms arise when task descriptions are not correctly understood or when the student's algorithm has a design flaw. The final category, inefficient algorithms, correctly solve the assigned task but in an inefficient manner.

The SINGLETONS case study in Section 3.4.2 demonstrated the use of a buggy algorithm. The student was instructed to write a program that returns those atoms that occur uniquely in an s-expression. Instead the student wrote a program that returns unique occurrences of every atom in the s-expression. Talus generated the following program commentary to explain the underlying misconception:

```
Looks like you misunderstood the task. The function SINGLETONS1 flattens its input and passes the resulting bag to the function SINGFLAT1. The function SINGFLAT1 unifies its input. The result is that one copy of every atom in the input to SINGLETONS1 will be returned. This is not the same as returning only those atoms that occur exactly once.

For example,

(SINGFLAT1 '(A B A C C E)) should be (B E), not (A B C E).
```

Program debugging can only continue with a correct algorithm so Talus selects the correct algorithm that best matches the student's code:
NOTE: I will continue the debugging process using the SINGLTONS-W-MEMBER algorithm, which is the correct algorithm that most closely matches your algorithm.

The MEMTREE case study in Section 3.4.1 demonstrated the use of an inefficient algorithm. The student wrote a solution with three functions when one would suffice. Talus generated the following commentary which mentions the more efficient solution:

MEMTR determines if AT is a member of the tree CONS by first flattening CONS using the function FLAT, and then determining if AT is an element of the resulting list. Note that it is possible to determine if AT is a member of the tree CONS without flattening CONS, i.e. by determining if AT is a member of the CAR or the CDR of nonatomic trees, or EQUAL to atomic trees.

In general it is up to the curriculum author to decide whether more efficient algorithms should be mentioned in commentary templates or not.

These examples illustrate that commentary templates are used to point out both stylistic and nonstylistic errors at the algorithm level. Inefficient algorithms are stylistic errors; buggy algorithms are nonstylistic errors. In the latter case, Talus can only continue debugging a program based on a buggy algorithm by reinterpreting the program as implementing a correct algorithm. If Talus continued with the buggy algorithm the final debugged program would solve a different task then the one assigned. So in order to continue, Talus selects the correct algorithm that best matches the student’s solution.

7.2 Missing and Unnecessary Function Definitions

Even if the algorithm is correct, errors can still occur at the next level of program design. Students can omit functions necessary to an algorithm or add functions unnecessarily. The first kind of error, missing functions, are reference functions that cannot be paired with any student functions in the best function mapping. Talus interprets these functions as essential to a minimal correct solution but missing from the student’s solution. Thus missing functions are nonstylistic bugs.

Missing function bugs are corrected by adding any unpaired normalized reference functions to the student’s solution. For example, in the SINGLTONS case study the student’s solution does not define a function capable of removing atoms from lists. This function is necessary to the correct implementation of the SINGLTONS-W-MEMBER algorithm, which is the correct algorithm that best matches the student’s solution. Talus supplies the correct definition from its reference function REMOVE:

**Missing Definition for REMOVE,**

```
(DEFUN REMOVE (ITEM BAG)
  (COND ((NILSP BAG) NIL)
     ((EQUAL ITEM (CAR BAG))
       (REMOVE ITEM (CDR BAG))))
    (T (CONS (CAR BAG)
      (REMOVE ITEM (CDR BAG))))
  )
```
The second kind of error that can occur at the function level are extra functions. These are student functions that cannot be paired with any reference functions in the best function mapping. Talus interprets these functions as superfluous to a minimal correct solution. Thus extra functions are only stylistic bugs and need not be corrected. However, Talus does comment on extra functions. For example suppose the student function CONCAT has the same definition as the primitive APPEND. Then Talus will generate a warning:

The function CONCAT appears unnecessary.

when CONCAT maps to EXTRA in the best function mapping.

7.3 Bug Correction at the Implementation Level

Even if the algorithm and its functional decomposition are correct, many bugs can still occur in the implementation of the functions. The bugs that can occur at the implementation level include:

- **Anomalous Conditional Tests.**
  - **Redundant Conditional Tests.** These are conditional tests logically implied or falsified by the terms governing them.
  - **Missing Conditional Tests.** A student function definition has a missing conditional test if there is some case student-case and some case ref-case such that neither
    
    \[ (\text{IMPLIES} \; \text{student-case} \; \text{ref-case}) \]
    
    nor
    
    \[ (\text{IMPLIES} \; \text{student-case} \; (\text{NOT} \; \text{ref-case}) ) \]
    
    is true. Chapter 6 explains how missing and extra conditional tests are detected. Section 7.3 discusses how missing conditional tests are inserted into the student's function prior to the generation of verification condition.
  
- **Extra Conditional Tests.** A student function definition has an extra conditional test if there is some case student-case and some case ref-case such that neither
    
    \[ (\text{IMPLIES} \; \text{ref-case} \; \text{student-case}) \]
    
    nor
    
    \[ (\text{IMPLIES} \; \text{ref-case} \; (\text{NOT} \; \text{student-case}) ) \]
    
    is true.

- **Buggy Symbolic Values.** These are detected as described in Chapter 6. Section 7.3.2 discusses the correction of these errors. Two common bugs within this category are:
  
  - **Wrong Function Calls.** The wrong function was called.
  
  - **Wrong Variable Updates.** The expression for an actual argument in a function call is incorrect.

7.3.1 Anomalous Conditional Tests

Anomalous conditional tests will be discussed first. These include redundant, extra, and missing conditional tests. Talus only generates warnings for redundant and extra conditional tests since they are only stylistic errors. Missing conditional tests are nonstylistic errors that must be corrected. There are three steps required in this process:
• *Determine the missing conditional test.* What test is missing?

• *Determine the insertion point for the test.* Where should it go?

• *Determine the code to place under the inserted test.* What should be placed in the branches of the new IF-expression?

Talus distinguishes between missing conditional tests that contribute to errors in symbolic evaluation and those that do not. The means of isolating and correcting each kind of missing conditional test is different. The first kind are called *missing guards*; the second kind are called *missing conditions*.

The means of detecting and correcting each kind of missing conditional test is different. Missing guards are detected by errors that occur when evaluating predicates in a student function. Missing conditions are detected by case splitting in reference functions. Chapter 6 explains in more detail how both types of missing conditional tests are detected. An example of a missing guard occurs below:

```
(DEFUN MEM (X L)
  (IF (EQUAL X (CAR L))
      T
      (MEM X (CDR L)))))
```

MEM returns *error* when symbolically evaluated assuming (NLISTP L). The missing guard, obtained from the normalized reference code, is (NLISTP L). An example of the second kind of missing conditional test, a missing condition, is shown here:

```
(DEFUN MEM (X L)
  (IF (NLISTP L)
      NIL
      (MEM X (CDR L)))))
```

MEM returns NIL for all inputs. A test logically equivalent to the missing condition (EQUAL X (CAR L)) is required for a correct solution to the MEMBER task.

### 7.3.1.1 Missing Guards

Missing guards are detected by mapping leaves from the reference code to the student code. As discussed in Chapter 6 a missing guard is indicated whenever

```
(IMPLIES (AND ref-case assumptions)
  (ERRORP test))
```

is true for the case ref-case from the reference function and some predicate test in the student's function. The terms in assumptions govern test but are not implied by ref-case.

The missing condition to be inserted is the first term in ref-case that is sufficient to imply that test causes an error. For example if the student code is

```
(DEFUN MEM (X L)
  (IF (EQUAL X (CAR L))
      L
      (MEM X (CDR L)))))
```

and the normalized reference code is
(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (MEM X (CDR L)))))

then Talus will detect a missing guard in the student’s code when (NLISTP L) is assumed. The truth of the following conjecture indicates that the missing guard is (NLISTP L):

(IMPLIES (NLISTP L) ;ref-case
          (ERRORP (EQUAL X (CAR L))));test

To correct the missing guard Talus creates a new IF-expression

(IF predicate if-true-branch if-false-branch)

where

• predicate = the missing guard. In the example above predicate is (NLISTP L).
• if-true-branch = the reference code governed by ref-case. The if-true-branch in the current example is the atom F.
• if-false-branch = the IF-expression in the student’s code whose predicate is test. In the example above the if-false-branch is

(IF (EQUAL X (CAR L))
  L
  (MEM X (CDR L))))

Continuing with the MEM example, the new IF-expression is:

(IF (NLISTP L)
  F
  (IF (EQUAL X (CAR L))
      L
      (MEM X (CDR L))))

The new IF-expression replaces the IF-expression in the student’s code whose predicate is test. The final student code with the missing guard added is:

(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          L
          (MEM X (CDR L)))))

7.3.1.2 Missing Conditions

Missing conditional tests that sometimes cause errors, or that cause no errors but contribute to incorrect program results are handled by slightly different means. The MEMBER case study presented in Section 3.4.3 provides an example of a missing condition. In that example, the student’s code is:
(DEFUN MEM (X L)
  (IF (LISTP L)
       (MEM X (CDR L))
       NIL))

and the normalized reference code is

(DEFUN MEM (X L)
  (IF (NLISTP L)
      F
      (IF (EQUAL X (CAR L))
          T
          (MEM X (CDR L)))))

When the case (LISTP L) is mapped to the reference function, case splitting occurs on the predicate (EQUAL X (CAR L)), as shown earlier in Figure 3-9.

Suppose case splitting occurs when mapping student-case to the reference function. To correct the missing condition, Talus creates a new IF-expression:

(IF predicate if-true-branch if-false-branch)

where

- \textit{predicate} = \textit{test}, the predicate in the reference function that causes case splitting. In the example above \textit{test} is (EQUAL X (CAR L)).

- \textit{if-true-branch} = the student code governed by student-case. In the current example student-case is (LISTP L) and the code governed by that term is (MEM X (CDR L)).

- \textit{if-false-branch} = the student code governed by student-case (another copy).

The new IF-expression for the example above is:

(IFDEF X (CAR L))
  (MEM X (CDR L))
  (MEM X (CDR L)))

The new IF-expression replaces the student code governed by student-case. In the buggy MEM function, the IF-expression replaces the underlined code below:

(DEFINE MEM (X L)
  (IF (LISTP L)
       (MEM X (CDR L))
       NIL))

resulting in:

(DEFINE MEM (X L)
  (IF (LISTP L)
      (IF (EQUAL X (CAR L))
          (MEM X (CDR L))
          (MEM X (CDR L))))
      NIL))

After symbolic values are debugged the final debugged code will be:
(DEFUN MEM (X L)
  (IF (LISTP L)
    (IF (EQUAL X (CAR L))
      T
      (MEM X (CDR L)))
    NIL)))

Multiple missing conditions and missing guards can also be detected and inserted. Once Talus detects and corrects one missing conditional test of either kind, it evaluates the code that results for any other missing guards or conditions. Talus maps leaves between the new code and the normalized reference function to detect additional missing guards or conditions. This process repeats until all missing conditional tests have been inserted. Consider the example in the MEMBER case study where the following definition was debugged:

(DEFUN MEM (X L) (MEM X L))

Talus first detects and corrects the missing condition (NLISTP L):

(DEFUN MEM (X L)
  (IF (NLISTP L)
    (MEM X L)
    (MEM X L))))

Then Talus reanalyzes the partially debugged code. Another missing condition is detected and then inserted, resulting in:

(DEFUN MEM (X L)
  (IF (NLISTP L)
    (MEM X L)
    (IF (EQUAL X (CAR L))
      (MEM X L)
      (MEM X L))))

After symbolic values are debugged the result will be:

(DEFUN MEM (X L)
  (IF (NLISTP L)
    F
    (IF (EQUAL X (CAR L))
      T
      (MEM X (CDR L)))))

7.3.1.3 Logically Overlapping Predicates

Sometimes Talus inserts a missing condition that did not really need to be inserted. Talus usually discovers the unnecessary insertion and suppresses the bug report that would have been issued otherwise. This section explains when this occurs and the reason why Talus must make these insertions.

Two predicates logically overlap if they are not logically equivalent but the truth or falsehood of one implies the truth or falsehood of the other. Consider a predicate ref-test from the normalized reference code, governed by terms and a predicate student-test from the student’s code. There are four cases when the two predicates logically overlap:
;Case 1

(IMPLIES (AND terms ref-test) student-test)
    --> T
and
(IMPLIES (AND terms (NOT ref-test)) (NOT student-test))
    --> F

;Case 2

(IMPLIES (AND terms ref-test) (NOT student-test))
    --> T
and
(IMPLIES (AND terms (NOT ref-test)) student-test)
    --> F

;Case 3

(IMPLIES (AND terms (NOT ref-test)) student-test)
    --> T
and
(IMPLIES (AND terms ref-test) (NOT student-test))
    --> F

;Case 4

(IMPLIES (AND terms (NOT ref-test)) (NOT student-test))
    --> T
and
(IMPLIES (AND terms ref-test) student-test)
    --> F

For example, consider the MAX task and the correct solution below:

Write a function that determines the maximum of a list of natural numbers.

(DEFUN MAX (L) (MAX1 0 L))

(DEFUN MAX1 (N L)
  (IF (NLISTP L)
      N
      (IF (LESSP (CAR L) N)
       (MAX1 N (CDR L))
       (MAX1 (CAR L) (CDR L))))

This student solution is compared to the following normalized reference code:
(DEFUN MAX (L) (MAXI 0 L))

(DEFUN MAXI (N L)
  (IF (NLISTP L)
      N
      (IF (GREATERP (CAR L) N)
          (MAXI (CAR L) (CDR L))
          (MAXI N (CDR L)))))

The predicate

(GREATERP (CAR L) N)

logically overlaps the predicate

(LESSP (CAR L) N)

since if (CAR L) is greater than N it follows that (CAR L) is not less than N:

(IMPLIES (AND (NOT (NLISTP L))
              (GREATERP (CAR L) N))
          (NOT (LESSP (CAR L) N)))

--> T

but if (CAR L) is not greater than N it does not follow that (CAR L) is less than N:

(IMPLIES (AND (NOT (NLISTP L))
              (NOT (GREATERP (CAR L) N)))
          (LESSP (CAR L) N))

--> F

For a counterexample to the last conjecture let N be 3 and L the list (3 2).

Now consider what happens when Talus maps cases from the student code to the reference code in order to detect missing conditions. When the case

(AND (NOT (NLISTP L))
      (NOT (LESSP (CAR L) N)))

is mapped to the reference code, case splitting will occur on the predicate

(GREATERP (CAR L) N))

since neither

(IMPLIES (AND (NOT (NLISTP L))
              (NOT (LESSP (CAR L) N)))
          (GREATERP (CAR L) N))

or

(IMPLIES (AND (NOT (NLISTP L))
              (NOT (LESSP (CAR L) N)))
          (NOT (GREATERP (CAR L) N)))

is true. Talus will consider (GREATERP (CAR L) N) a missing condition and modify the student's code to be:
(DEFUN MAX (L) (MAX1 0 L))

(DEFUN MAX1 (N L)
 (IF (NLISTP L)
  N
  (IF (LESSP (CAR L) N)
   (MAX1 N (CDR L)) ;(CAR L) less than N.
   (IF (GREATERP (CAR L) N)
    (MAX1 (CAR L) (CDR L)) ;(CAR L) greater than N.
    (MAX1 (CAR L) (CDR L)))))) ;(CAR L) equal to N.

Talus will debug the symbolic values and find no bugs in the resulting modified program.

In the example above the insertion of the missing condition was unnecessary. Talus detects unnecessary insertions by comparing the branches of inserted IF-expressions after symbolic values have been debugged. If the two branches of an inserted IF-expression are identical then Talus suppresses bug warnings about the missing condition and explains the faulty assumption that it made. In the case above, Talus would generate the following report after debugging the symbolic values:

[Note: The condition (GREATERP (CAR L) N), which Talus had assumed was a missing condition, has been determined not to be missing after all.]

The insertion of missing conditions is sometimes necessary for logically overlapping cases. Suppose MAX was incorrectly defined as:

(DEFUN MAX (L) (MAX1 0 L))

(DEFUN MAX1 (N L)
 (IF (NLISTP L)
  N
  (IF (EQUAL (CAR L) N)
   (MAX1 N (CDR L))
   (MAX1 (CAR L) (CDR L))))))

The result after inserting the missing condition is:

(DEFUN MAX (L) (MAX1 0 L))

(DEFUN MAX1 (N L)
 (IF (NLISTP L)
  N
  (IF (EQUAL (CAR L) N)
   (MAX1 N (CDR L))
   (IF (GREATERP (CAR L) N) ;Inserted Missing Condition
    (MAX1 (CAR L) (CDR L))
    (MAX1 (CAR L) (CDR L)))))) ;Bug

Talus will detect a bug in the last call to MAX1 above: when (CAR L) is not greater than N then N should not be updated to (CAR L), instead it should remain unchanged. With this correction the final debugged code is:
(DEFUN MAX (L) (MAXI 0 L))

(DEFUN MAXI (N L)
  (IF (NILSTP L)
    N
    (IF (EQUAL (CAR L) N)
      (MAXI N (CDR L))
      (IF (GREATERP (CAR L) N) ;Inserted Missing Condition
       (MAXI (CAR L) (CDR L))
       (MAXI N (CDR L)))))) ;Bug Fix

Talus found a bug in the code underneath the inserted condition and repaired it. So in this case the inserted condition (GREATERP (CAR L) N) was necessary, since the two possibilities:

(GREATERP (CAR L) N)

and

(NOT (GREATERP (CAR L) N))

must be distinguished.

These two examples illustrate that Talus cannot know prior to debugging symbolic values if a missing condition should be inserted when there are logically overlapping predicates in the student and reference code. The solution adopted by Talus is to always insert missing conditions and then to detect when the insertion was unnecessary.

7.3.2 Isolating Bugs in Symbolic Values

After correcting missing guards and missing conditions, Talus detects bugs in symbolic values by generating and evaluating verification conditions as described in Chapter 6. The bug detection algorithm of Talus, shown earlier in Figure 6-15, calls a function correct to correct student code fragments with normalized reference code fragments.

The function correct is defined in pseudocode below:

```
correct[ref-code, student-code, assumptions] =

;Isolate the bug and determine code to replace the student’s.
  bug-fix = isolate[ref-code, student-code, assumptions];

;Replace the buggy code with the new code.
  replace[student-code, assumptions, bug-fix].
```

where

- ref-code is the reference code fragment that is being mapped;
- student-code is the buggy student code fragment that it was mapped to. It is considered buggy since it failed either the functional equivalence or termination verification conditions that were generated for it.
- assumptions are the terms in ref-case that govern ref-code and any terms governing student-code that are not logically implied by ref-case. Viewed another way, the reference
code symbolically evaluates to ref-code and the student code symbolically evaluates to student-code when assumptions are true.

The function isolate attempts to isolate the bug in the student’s code to a single top-level\(^{19}\) s-expression in student-code. If it can do so then it calls itself recursively. Eventually either the bug is isolated to an atomic s-expression or isolate cannot find the bug in a single top-level s-expression in student-code. The function isolate returns a code fragment that combines parts of student-code and ref-code and that satisfies both functional equivalence and termination verification conditions. In the worse case isolate returns ref-code. In the best case isolate returns an s-expression that differs from student-code only by a single atom. The function replace changes student-code to be identical to bug-fix. Section 7.3.3 discusses how replace is implemented.

The algorithm for isolate systematically replaces s-expressions in student-code by s-expressions in ref-code. If student-code satisfies both termination and functional equivalence verification conditions after a replacement then all bugs were in the s-expression replaced. The bug isolation algorithm can then be applied recursively to the code replaced. The algorithm for isolate is shown in Figure 7-1.

\[\text{isolate}[\text{ref-code}, \text{student-code}, \text{assumptions}] =\]

IF ref-code and student-code are not both function calls
THEN (RETURN ref-code)
ELSE
   \[i \leftarrow 0;\]
   LOOP: \[i \leftarrow i + 1;\]
      candidate \leftarrow copy[student-code];
      IF \(i > \text{length}[\text{student-code}]\) THEN (RETURN ref-code);
      candidate[\(i\)] \leftarrow \text{ref-code}[\(i\)];
      IF
      ;Check Functional Equivalence Verification Conditions:
      \[\text{theoremp}\left[\text{IMPLIES assumptions}
      \quad \text{equality-predicate ref-code candidate}\right]\]
      AND
      ;Check Termination Verification Conditions:
      \[\text{check-terminations}[\text{candidate}, \text{assumptions}]\]
      THEN
      ;Bug isolated to the \(i\)th expression. Call isolate recursively.
      \[\text{fix} \leftarrow \text{isolate}[\text{ref-code}[\(i\)], \text{student-code}[\(i\)], \text{assumptions}]\];
      candidate[\(i\)] \leftarrow fix;
      RETURN candidate;
      ELSE (GO TO LOOP); ;Bug not isolated, try the next subexpression.

Figure 7-1: The Bug Isolation Algorithm isolate

---

\(^{19}\)The s-expression \(s\) that occur in code is top-level if the only s-expression containing \(s\) is code. E.g. if code is \((\text{MEM X (CDR L)})\) then MEM, X, and (CDR L) are top-level s-expressions but CDR and L are not.
The notation \texttt{candidate[i] <- ref-code[i]} means that the \texttt{i}th top-level s-expression of \texttt{candidate} is replaced by the \texttt{i}th top level s-expression of \texttt{ref-code}. For example if:

\begin{verbatim}
candidate = (APPEND (CAR L) (FLATTEN (CDR L)))

ref-code = (APPEND (FLATTEN (CAR L)) (FLATTEN (CDR L)))
\end{verbatim}

then

\begin{verbatim}
\end{verbatim}

changes \texttt{candidate} to be identical to \texttt{ref-code}.

The \texttt{MEMTREE} case study demonstrates bug isolation to a single atom in a buggy student code fragment. \texttt{Talus} determines that the code fragment

\begin{verbatim}
(IN X (CAR L))
\end{verbatim}

is buggy when the following functional equivalence verification condition fails:

\begin{verbatim}
(IMPLIES (AND (NOT (NLISTP L)) ;Case from Reference Code
           (NOT (EQUAL X (CAR L))))
          (IFF (IN X (CDR L)) ;Reference Code
               (IN X (CAR L)))) ;Student Code
\end{verbatim}

so the \texttt{isolate} algorithm is called with:

\begin{verbatim}
student-code = (IN X (CAR L))

ref-code = (IN X (CDR L))
\end{verbatim}

\begin{verbatim}
assumptions = (AND (NOT (NLISTP L))
                (NOT (EQUAL X (CAR L))))
\end{verbatim}

Replacing the first or second s-expression in \texttt{student-code} by the first or second s-expression in \texttt{ref-code} has no effect. However when the third s-expression, \texttt{(CAR L)}, is replaced by \texttt{(CDR L)} both functional equivalence and termination verification conditions are satisfied. \texttt{Talus} issues the following report:

\texttt{NOTE: There is a bug in the expression (CAR L), which is bound to the parameter L of the function IN, in the expression (IN X (CAR L)) inside your function IN.}

Then \texttt{isolate} is called recursively with

\begin{verbatim}
student-code = (CAR L)

ref-code = (CDR L)
\end{verbatim}

\begin{verbatim}
assumptions = (AND (NOT (NLISTP L))
               (NOT (EQUAL X (CAR L))))
\end{verbatim}

This time replacing the first s-expression in \texttt{student-code} by the first s-expression in \texttt{ref-code} fixes the bug. Since the bug cannot be further localized \texttt{Talus} issues this bug report:

\texttt{HINT: Looks like you called function CAR instead of function CDR in the expression (CAR L) in your function IN.}
The innermost call to *isolate* returns (CDR L) as its bug fix. The outermost call to *isolate* returns (IN X (CDR L)) as its bug fix.

In the example above it might appear as if *isolate* merely alters *student-code* to be the same as *ref-code*. This is not correct. Suppose the buggy student code fragment had been:

(IN (CAR (LIST X)) (CAR L))

The bug isolation algorithm would operate as before: first isolating the bug to (CAR L) and then to CAR. The bug fix returned would be:

(IN (CAR (LIST X)) (CDR L))

There are some improvements to *isolate* that could be made but which are not currently implemented. One improvement allows more specific bug isolation for commutative operators. For example, assume that *isolate* is called with the following parameters:

ref-code = (TIMES (FACT (SUB1 N)) N)

student-code = (TIMES N (FACT (ADD1 N)))

assumptions = (NOT (ZEROP N))

Talus will skip over the first possible replacement of TIMES for TIMES since nothing will change in the student expression. Then Talus replaces the second s-expression in the student's code by the second s-expression in the normalized reference code and checks to see whether the new code satisfies the required functional equivalence verification condition:

(IMPLIES (NOT (ZEROP N))
           (EQUAL (TIMES (FACT (SUB1 N)) (FACT (ADD1 N)))
                   (TIMES (FACT (SUB1 N)) N)))

but the conjecture is invalid. Replacing the third expression also fails:

(IMPLIES (NOT (ZEROP N))
           (EQUAL (TIMES N N)
                   (TIMES (FACT (SUB1 N)) N)))

Since Talus fails to isolate the bug to a single s-expression within *student-code* the function *isolate* can do no better than return *ref-code*:

(TIMES (FACT (SUB1 N)) N)

Adding information about commutative operators and special heuristics to correct bugs in the variable updates of these operators would allow Talus to return:

(TIMES N (FACT (SUB1 N)))

where only ADD1 has been changed to SUB1, rather than

(TIMES (FACT (SUB1 N)) N)

where both variable updates are altered.

The algorithm in Figure 7-2 illustrates a bug isolation algorithm *isolate* with special provisions for commutative operators. *isolate* is a front end to *isolate*: *isolate* calls *isolate* when its rules for commutative operators do not apply, and recursive calls in *isolate* are now replaced by calls to *isolate*. 
The predicate `commutativep` returns true if an operator is commutative, false otherwise. Talus can either store information about commutative operators or prove that the operators are commutative.

\[
isolate*[\text{ref-code, student-code, assumptions}] =
\]

;Note: Term[i] selects the ith element of Term.

\[
\text{IF } \text{commutativep}[\text{ref-code}[1]] \\
\quad \text{AND} \\
\quad \text{ref-code}[1] = \text{student-code}[1] \\
\quad \text{AND} \ \text{length}[\text{ref-code}] = \text{length}[\text{student-code}] = 3
\]

\[
\text{THEN}
\]

\[
; \text{student-code} = (\text{fn } x \ y) \\
; \text{ref-code} = (\text{fn } u \ v)
\]

\[
\text{IF theorem}( \ (\text{IMPLIES assumptions} \\
\quad (\text{EQUAL student-code}[2] \ \text{ref-code}[3])))
\]

\[
\text{THEN}
\]

\[
; \text{assumptions} \to x = v \text{ so compare } y \text{ and } u:
\]

\[
\text{isolate*}[\text{ref-code}[2], \quad ;\text{New ref code: } u \\
\quad \text{student-code}[3], \quad ;\text{New student code: } y \\
\quad \text{assumptions}] \quad ;\text{Same assumptions}
\]

\[
\text{ELSE}
\]

\[
\text{IF theorem}( \ (\text{IMPLIES assumptions} \\
\quad (\text{EQUAL student-code}[3] \ \text{ref-code}[2])))
\]

\[
\text{THEN}
\]

\[
; \text{assumptions} \to y = u \text{ so compare } x \text{ and } v:
\]

\[
\text{isolate*}[\text{ref-code}[3], \quad ;\text{New ref code: } v \\
\quad \text{student-code}[2], \quad ;\text{New student code: } x \\
\quad \text{assumptions}] \quad ;\text{Same assumptions}
\]

\[
\text{ELSE} \ \text{isolate}[\text{ref-code, student-code, assumptions}]
\]

Figure 7-2: The Improved Bug Isolation Algorithm \textit{isolate*}

---

The limitation of the original \textit{isolate} algorithm regarding commutative operators is symptomatic of a more general problem. Talus may not be able to isolate a bug to an s-expression because the buggy s-expression and its correction in the stored code occur at the wrong level of embedding. Or they could occur in the wrong order, as in the example with commutative operators. Here is an example where the bug and bug fix occur at a different level of embedding:

\[
\text{student-code} = (\text{APPEND } (\text{LIST } (\text{CDR } y)) \ z) ;\text{Bug: CDR should be CAR}
\]

\[
\text{ref-code} = (\text{CONS } (\text{CAR } y) \ z)
\]

\[
\text{assumptions} = (\text{AND } (\text{LISTP } y) (\text{LISTP } z))
\]

The bug and bug fix are underlined. Talus will not be able to discover the bug fix unless it has a specific
rule to handle CONS/APPEND comparisons or unless isolate attempts substituting reference code s-expressions into student code s-expressions that are not at the top level. In the latter case the search space of possible edits can be much larger than the linear substitution scheme.

Talus does not presently have a bug catalog with specific rules for handling CONS, APPEND, etc. A bug catalog with rules to recognize specific bugs could augment, but not replace, the existing uniform bug detection and correction algorithms. These rules could be used to aid the bug isolation algorithm, as in the example above, or to generate hints to correct specific misconceptions.

7.3.3 Editing Code to Fix Bugs

This section describes how Talus produces a debugged version of the student’s program by editing a copy of the student’s solution. This editing requires some means of uniquely identifying s-expressions within code. It is not sufficient for an edit to merely specify “Replace CAR by CDR in function MEMBER” since that function may have more than one occurrence as in the definition below:

```
(DEFUN MEMBER (EL SET)
  (IF (NULL SET)
    F
    (IF (EQUAL (CAR SET) EL) ; First CAR
      T
      (MEMBER EL (CAR SET)))))) ; Second CAR
```

Two means have been used in Talus. The first strategy identifies s-expressions by the commands that would be given to a structure editor to select the s-expression. This strategy is adequate for performing bug edits on programs that are in IF-Normal form. The second strategy associates s-expressions with unique nodes in a graph representation. Bug edits consist of replacing one node (representing the buggy code) by another node (representing a bug fix). The first strategy is no longer used since this second strategy more readily lends itself to tracing edits through program transformations. Edit tracing will be discussed in Chapter 8.

7.3.3.1 Automating a Structure Editor

Edit paths can uniquely identify s-expressions. An edit path is a series of numbers (s_1,...,s_n) that instruct the editor to repeatedly select the s_i-th s-expression from the current s-expression. For example in the definition of MEMBER above the first CAR can be identified by an edit path (4 4 2 2 1) and the second CAR can be identified by an edit path (4 4 3 1). To choose the second CAR, following the edit path (4 4 3 1), first choose the fourth s-expression in:

```
(DEFUN MEMBER (EL SET)
  (IF (NULL SET)
    F
    (IF (EQUAL (CAR SET) EL)
      T
      (MEMBER EL (CAR SET))))))
```

Next choose the fourth s-expression in:
(IF (NULL SET)
F
(IF (EQUAL (CAR SET) EL)
T
(MEMBER EL (CAR SET)))
)

Now choose the 4th s-expression in:

(IF (EQUAL (CAR SET) EL)
T
(MEMBER EL (CAR SET)))

Next choose the 3rd s-expression in:

(MEMBER EL (CAR SET))

Finally select the first s-expression in:

(CAR SET)

to reach the CAR.

Bug edits are performed on a copy of the student’s code by specifying the buggy code with an edit path and also the s-expression which should replace it. Edit paths required to reach buggy s-expressions are recorded by isolate. To repair multiple bugs, Talus need only repair one bug at a time. Each bug fix is applied to the current partially debugged program. The final program contains all the bug fixes. For example given the following buggy solution to the MEMBER task:

(DEFUN MEM (X L)
  (IF (NLISTP L)
   'FALSE
   (IF (EQUAL X (CAR L))
     'TRUE
     (MEM X (CAR L)))))

Talus detects two bugs that can be corrected by these bug edits:

(REPLACE (4 3) BY F)
(REPLACE (4 4 3 1) BY CDR)

After these edits, the final debugged code will be:

(DEFUN MEM (X L)
  (IF (NLISTP L)
   F
   (IF (EQUAL X (CAR L))
     'TRUE
     (MEM X (CDR L)))))

Missing conditional tests are also corrected by replacing one s-expression by another. The s-expression replaced is the student code that should be, but is not, governed by the missing guard or missing condition. The s-expression that replaces it is the new IF-expression containing the missing conditional test and the code replaced in one or both of its branches. For example given the following buggy definition of MEM:

(DEFUN MEM (X L) (MEM X L))
Talus would first detect one missing condition and generate a bug edit command to fix it:

(REPLACE (4) BY (IF (NLISP L) (MEM X L) (MEM X L)))

This bug edit would produce:

(DEFUN MEM (X L)
   (IF (NLISP L)
       (MEM X L))
   (MEM X L)))

Then the remaining missing condition would be detected. The following bug edit command would fix the second missing condition:

(REPLACE (4 4) BY (IF (EQUAL X (CAR L)) (MEM X L) (MEM X L)))

At this point the partially debugged code would appear as:

(DEFUN MEM (X L)
   (IF (NLISP L)
       (MEM X L))
   (IF (EQUAL X (CAR L))
       (MEM X L)
       (MEM X L)))

Then Talus debugs the symbolic values. The final debugged code will be:

(DEFUN MEM (X L)
   (IF (NLISP L)
       (IF (EQUAL X (CAR L))
           T
           (MEM X (CDR L)))))

7.3.3.2 Editing Graph Representations of Programs

The edit path strategy has now been superseded by the graph edit strategy where programs are represented as graphs. These graphs are altered to perform bug edits. The main advantage of the graph representation is the ease by which program transformations can be represented by graphs where copied or transformed s-expressions are linked together. These links facilitate the back propagation of edits from a simplified program to the student’s original program. This is discussed in more detail in Chapter 8.

In the graph edit strategy, programs are represented by program graphs. Each CONS cell in a LISP function corresponds to a node in a program graph. Each node has both CAR and CDR pointers that duplicate the role of the CAR and CDR pointers of the CONS cell represented. In addition, there are PARENT pointers that point to a node’s parent and an SEXP property that caches the node’s list representation. For example, the following function:

(DEFUN REV (L) ;Buggy Reverse Function
   (IF (LISTP L)
       (CONS (REV (CDR L)) (LIST (CAR L)))
       NIL))

has the graph representation shown in Figure 7-3. The program graph representation is drawn in a similar fashion to the conventional box representation of LISP CONS cells, but note the double-headed arrows
indicating the backward (PARENT) pointers between CONS cells and their parents.

Figure 7-3: Graph Representation of REV

Bug fixes are performed by replacing buggy s-expression nodes with bug fix nodes. Each bug fix node has two additional pointers:

1. **BUGGY-NODE**, which points to the node representation of the buggy s-expression replaced.

2. **BUG-FIX-NODE**, which points to the node representation of the bug fix that replaces the buggy s-expression.

and two additional properties:

1. **BUGGY-SEXP**, which caches the list representation of the buggy s-expression replaced.

2. **BUG-FIX-SEXP**, which caches the list representation of the code that replaced the buggy s-expression.

Talus keeps track of the node associated with each s-expression during symbolic evaluation and bug detection. If Talus discovers that s-expression student-code, represented by node student-node, is buggy and needs to be replaced by s-expression ref-code, represented by node ref-node, Talus replaces student-node by a bug fix node with the following pointers:

- **BUGGY-NODE**, which points to student-node, and
- **BUG-FIX-NODE**, which points to ref-node.

The bug fix node also has the following two properties:

- **BUGGY-SEXP**, whose value is student-code, and
• BUG-FIX-SEXP, whose value is ref-code.

After inserting a bug fix node, Talus updates the SEXP properties of all nodes above the inserted node so that they reflect the bug fix. For example, Figure 7-4 illustrates the graph representation that would result when Talus replaces CONS by APPEND in the definition of REV, producing this code:

```
(DEFUN REV (L)
 (IF (LISTP L)
   (APPEND (REV (CDR L)) (LIST (CAR L)))) ; Bug Fix: APPEND/CONS NIL))
```

![Diagram of graph representation of a bug fix to REV](image)

**Figure 7-4: Graph Representation of a Bug Fix to REV**

Talus retains the original buggy code along with the corrections so that Talus can selectively highlight program bugs or bug fixes in its graphics displays. Bugs are highlighted by displaying them in reverse video or a special font. Figure 7-5 shows the bug highlighting that occurs after debugging in the MEMTREE case study.

Missing conditional tests are inserted similarly. Suppose Talus inserts the missing condition (NLISTP L) into the definition below:

```
(DEFUN REV (L)
 (APPEND (REV (CDR L)) (LIST (CAR L)))))
```
to produce this code:

\[
\text{(DEFUN REV (L))}
\text{(IF (NLISTP L) NIL (APPEND (REV (CDR L)) (LIST (CAR L)))))}
\]

This bug edit is shown if Figure 7-6.

7.4 Hint Generation

Presently Talus generates hints, i.e. bug reports, that are highly specific. They indicate both the bug detected and the bug fix that Talus uses to correct the bug. These hints are not intended to be pedagogically optimal or to address misconceptions, rather the intent is to show the specificity of advice that can be given. An ITS using Talus might choose to generate less specific hints that do not include the exact location or correction of a bug.

Hints are generated by instantiating templates with specific information about a particular bug. The templates used for generating hints are shown below. The following abbreviations are used:

- \( \$ \)term stands for a string containing the value of \( \text{term} \).
- \( \% \)term stands for a string generated from the list \( \text{term} \). If \( \text{term} \) has elements \( t_1 \ldots t_n \) then the string has the form "\( t_1, t_2, \) and \( t_3 \)" or "\( t_1 \) and \( t_2 \)" or "\( t_1 \)", according to the length of \( \text{term} \).

For example if \( \text{term}_1 = (\text{CAR L}) \) and \( \text{term}_2 = ((\text{NULL L}) (\text{EQUAL} (\text{CAR L}) X)) \) then "When \( \$ \text{term}_2 \) is true, there is a bug in \( \% \text{term}_1 \)" abbreviates "When \( (\text{NULL L}) \) and \( (\text{EQUAL} (\text{CAR L}) X) \) are true, then there is a bug in \( (\text{CAR L}) \)."

Here are the templates for anomalous conditional tests:

- **Redundant Conditional Tests**
  - \( \% \)Always True. "It appears that the test \( \% \text{predicate} \) will always be true when it is reached."
• *Always False.* "It appears that the test \( ?\text{predicate} \) will always be false when it is reached."

• *Extra Conditional Tests.* "NOTE: Extra condition in student's code reached when \$assumption \$ is true. The extra condition is \$\text{predicate} \$.*

• *Missing Conditional Tests.*
  
  • *Missing Guards.* "BUG: Missing condition. When \$assumption \$ is assumed, the IF-test \$\text{predicate} \$ always causes an error. The missing condition is \$\text{missing} \$."

  • *Missing Conditions.* "BUG: Missing condition in student's code reached when \$assumptions \$ are true. The missing condition is \$\text{missing} \$."

• *Incorrect S-Expressions in Symbolic Values.*

[The bug occurs in the expression ?exp_1,
 which is in the expression ?exp_2,
 ... which is in the expression ?exp_n.

The clause containing this expression is reached when the following conditions are true: $assumptions.
"

$student-code is a buggy s-expression in the student's function fn, which can be corrected by the s-expression $ref-code, obtained from normalized reference code. $type_1 and $type_2 are the types of these two s-expressions. The types are one of "number", "constant", "variable", "function call", or "quote", as appropriate. $exp_1...$exp_n are the terms enclosing the buggy term $student-code. The terms in $assumptions logically imply the terms governing $student-code in fn.

For a bug where function ?fn_1 is called instead of function ?fn_2 in expression $student-code, the hint template is:

"HINT: Looks like you called function ?fn_1 instead of function ?fn_2 in the expression ?exp_1 in your function ?fn.

[The bug occurs in the expression ?exp_1,
 which is in the expression ?exp_2,
 ... which is in the expression ?exp_n.

The clause containing this expression is reached when the following conditions are true: $assumptions.
"

- **Missing Function Calls that Omit a Necessary Side Effect.** "HINT: The expression ?exp is missing from the function ?fn and should be inserted."

- **Extra Function Calls that Perform an Unnecessary Side Effect.** "HINT: The expression ?exp is unnecessary and should be deleted from the function ?ln."

- **Extra Functions.** "The function ?fn appears unnecessary."

- **Missing Functions.** "Here is a correct definition for the function ?fn, which was not defined in your solution."

---

20 Chapter 8 discusses some common side effects that are not strictly necessary to an algorithm's implementation but which are not considered bugs. For example NCONC can replace APPEND anywhere, provided that its arguments can be shown to be fresh list structure.
7.5 Alternate Strategies
This section considers alternative strategies for bug correction. Section 7.5.1 considers the use of heuristics to search for plausible bug edits. Section 7.5.2 considers bug correction strategies that rely on program synthesis. Section 7.5.3 considers a search strategy that searches bug equivalence classes.

7.5.1 Repair Heuristics
An alternative approach to bug correction is to have repair heuristics that suggest plausible bug edits for specific bug manifestations. The bug edits are undetermined - for a specific bug there may be more than one type of plausible edit and the edits may apply in several locations. The wrong bug edits can increase the number of bugs in the program. Search is required to find those edits that minimize the number of bugs in the program.

MYCROFT [Goldstein 74], a design for a LOGO program debugger, corrects bugs with repair heuristics. The tasks are to draw simple pictures such as the well in Figure 7-7. The LOGO program being debugged is run and a trace is kept of the lines drawn during its execution. The task is described with a picture model, a geometric description describing shapes in the expected picture and their interrelationships. Lines drawn by the LOGO program are mapped to picture model parts. Violations of model predicates - such as incorrect spatial relationships or connections between parts - are interpreted as manifestations of underlying bugs. Suppose a student wrote a LOGO program to draw the well in Figure 7-7 by calling three subroutines: ROOF, POLE, and BASE. Assume that the execution of the program results in the picture shown in Figure 7-8. (Numbers and dashed lines indicate the path of the turtle while labels indicate the parts drawn by the three subroutines. These are for the reader's convenience; only the solid lines are actually drawn by the program.) Then an example of an underlying bug that can explain this result is the omission of code to bring about the proper interface between the subroutines to draw the triangle and the line that connects it to the square. If these are the ROOF and POLE subroutines then the bug heuristics will suggest the following plausible bug edits:

- Modify existing interface code. E.g. change an existing TURN command.
- Add new interface code. E.g. add a new TURN command.

The edits can occur at:
- The end of ROOF.
- The beginning of POLE.
- In between calls to ROOF and POLE in the main program.

The use of repair heuristics to correct bugs has the following drawbacks:
- Significant search is required to find bug fixes for nontrivial programs and nontrivial tasks. The LOGO domain of MYCROFT contains few commands (TURN, MOVE, DRAW, PEN UP, PEN DOWN) and tasks are simple.
- It does not work well when bug manifestations and underlying bugs are loosely coupled. E.g. many bugs can cause nontermination so repair heuristics for nontermination can suggest many alternative bug edits.
- The search space will be vastly increased in programs that contain multiple interacting bugs. Each repair heuristic may suggest bug edits that only degrade the performance of the program due to the presence of other bugs. Only the right combination of multiple edits will remove all bugs.
- Repair heuristics for unexpected kinds of bugs will be missing.
Figure 7-7: The Wishing Well Task

The large search space is symptomatic of a lack of knowledge about the actual program bugs and how they should be corrected. Since Talus can isolate bugs to specific code fragments, it has no need for heuristics that relate manifestations to possible causes. In addition, there is no need for heuristics to suggest plausible bug edits since Talus can correct program bugs with normalized reference code.

7.5.2 Program Synthesis

Buggy code can also be repaired by synthesizing correct code that satisfies the specifications that the buggy code should have. The inductive inference system (MIS) of Shapiro [Shapiro 83] can synthesize PROLOG clauses to replace buggy clauses. The synthesized clauses cover goals that should be satisfied (e.g. \texttt{append([a],[b],[a,b]))} while not covering goals that should not be satisfied (e.g. \texttt{append([a,d],[b,c],[a,b]))}. Other systems with debugging capabilities that can synthesize bug corrections are TURTLE [Miller 82] and the LISP tutor GREATERP [Reiser 85]. TURTLE can synthesize LOGO code to draw simple pictures such as that shown in Figure 7-7. Synthesized LOGO code can be used to correct buggy code in completed student programs. However, TURTLE cannot provide assistance as the student writes his program. In contrast, GREATERP can synthesize LISP code to continue a task solution when a student is at an impasse.

There are two main problems with the program synthesis approach:

- \textit{It requires a program synthesis capability.} Program synthesis in nontrivial domains is difficult. If the program synthesizer fails some bugs may not be corrected.
- \textit{Partially correct solutions are discarded.} Students may learn more by seeing the mistakes in
their solutions corrected than by seeing synthesized correct solutions. Merely presenting synthesized code fails to address misconceptions that can be discovered from bugs in student code. The repair performed may be far removed from the student’s code, even when a simple fix would correct that code. The resulting bug fix may only confuse the student.

If the task size is small or decomposed into many small parts, as in GREATERP, the last criticism does not apply. With GREATERP only small amounts of code are entered at a time. Consequently, the amount of buggy code replaced or supplied by program synthesis is minimal and surrounding correct code is retained. On the other hand suppose Shapiro’s model inference system (MIS) is used to synthesize correct PROLOG clauses to replace buggy clauses. Then entire PROLOG clauses may be replaced when all that needs to be changed is an arithmetic test or a variable name.

7.5.3 Searching Bug Equivalence Classes

Shapiro’s PROLOG debugger PDS6 avoids the problem just mentioned of unnecessarily replacing entire PROLOG clauses when smaller corrections are sufficient. Buggy PROLOG clauses are not discarded. Instead buggy clauses are perturbed to see if a correct clause can be derived from the buggy clause by some small change. The means of perturbation are specified by bug classes. For example in one bug class, variable misspellings, the variables in PROLOG clauses are altered. In another arithmetic tests are altered - the operator > may be replaced by < for example. The buggy clause acts as a seed in a search space of possible corrections. The operators that produce new states in the search space, called refinement operators, depend on the bug class being searched.

Several assumptions severely limit the applicability of this approach. This approach assumes that
the clause being replaced is nearly correct. The search space will be very large for clauses with several
conjuncts and multiple bugs. Secondly it is assumed that all bugs in a clause belong to only one bug
class. Thus a clause with only two bugs from different bug classes cannot be corrected. Finally, note that
bugs outside of known bug equivalence classes cannot be corrected at all.

7.6 Summary
The key points of this chapter are:
• Talus corrects nonstylistic bugs at all three levels of program design: the algorithm level, the
  function level, and the implementation level.
• Stylistic bugs produce only warnings or commentary.
• Talus can isolate bugs to s-expressions contained in symbolic values. Frequently these s-
  expressions are atomic.
• Talus isolates bugs in symbolic values by tentatively replacing s-expressions in student code
  by s-expressions in reference code. When a replacement satisfies all verification conditions
  then only the s-expression replaced was buggy. Then the isolation algorithm is called
  recursively until no further isolation can occur.
• Bugs are corrected by replacing nodes in a graph representation of programs. Both buggy
  s-expressions and missing conditional tests can be corrected by this means.
• Alternative means of bug correction are:
  • Program Synthesis, but this requires a program synthesis capability and may discard
    partially correct solutions.
  • Repair Heuristics, but this is impractical for nontrivial domains and programs with
    multiple bugs.
  • Searching Bug Equivalence Classes, but this requires nearly correct programs and the
    prediction of expected bug types.
Chapter Eight
Beyond Pure LISP

8.1 Introduction

This chapter explains how Talus debugs programs that are not in IF-Normal Form or in pure LISP. Extensions are introduced to debug programs that can cause side effects to shared list structure, property lists and arrays. In addition, these extensions allow additional special forms such as COND, LAMBDA, and PROG, and the use of mapping functions such as MAPCAR and MAPCAN.

Figure 8-1 illustrates the relationship between these extensions and the algorithm recognition, bug detection, and bug correction methods of Chapters 5, 6, and 7. New capabilities introduced in this chapter are drawn outside of the dashed line; debugging capabilities already explained are drawn inside the dashed line. With these extensions, the first step in program debugging is to simplify programs by applying equivalence preserving transformations. Then the simplified programs are parsed into E-frames. Algorithm recognition can be performed as discussed in Chapter 5. No modifications are necessary for algorithm recognition, but the bug detection techniques of Chapter 6 must be extended to allow for programs with side effects. After bugs have been detected, the bug correction techniques of Chapter 7 are sufficient to correct bugs in the simplified program, but not in the original program. Special techniques for tracing bug corrections back through program transformations are required.

---

The increased debugging capabilities introduced in this chapter allow Talus to debug programs with special forms such as COND and primitives such as NCONC; these constructs are not present in the Boyer-Moore Logic. These capabilities are supported by:

- *Heuristics that allow comparison of symbolic values when side effects can occur.*  Section 8.2
explains how Talus debugs programs that are not restricted to pure LISP.

- Program simplification transforms that simplify programs for debugging. Section 8.3 explains how Talus simplifies programs to:
  - Reduce them to the core dialect.
  - Place them in IF-Normal Form.
  - Facilitate algorithm recognition.
- Heuristics that trace bug edits back to original programs. Section 8.4 explains how bug corrections to simplified programs are regressed through program transformations to the programs originally entered.

8.1.1 The Core and Extended LISP Dialects

With the extensions introduced in this chapter, task solutions are no longer restricted to the pure LISP dialect of Figure 3-2. Either the core or extended dialect can be used. The core LISP dialect of Talus extends the pure LISP dialect by introducing primitives that:

- Alter shared list structure. RPLACA, RPLACD, and NCONC.
- Allow property list manipulation. GET, PUTPROP, and REMPROP.
- Allow array assignment and reference. ASET, and AREF.

These primitives will be referred to as side-effectors since they cause or rely upon side effects and are not part of pure LISP. The core dialect also allows PROGN so functions can be called for side effect only. The complete core dialect is shown in Figure 8-2.

The extended LISP dialect expands the core dialect by adding special forms for:

- Mapping Functions. MAPCAR, MAPCAN, MAP, MAPLIST, MAPPING, and MAPC. FUNCTION is allowed inside mapping constructs.
- Imperative Programming. PROG with SETQ, GO, and RETURN allowed inside.
- Convenience. COND and LAMBDA.

Figure 8-3 shows the extended LISP dialect of Talus.

8.1.2 Program Simplification and Edit Inversion

This section provides a simple example to illustrate program simplification transforms and edit inversion. A brief review of program simplification and edit inversion will be given first.

Talus applies equivalence preserving transformations, called program simplification transforms, to simplify programs into IF-Normal Form and the core dialect. Talus can debug these simplified programs with the extensions discussed in Section 8.2 that allow for side effects. In all other respects, Talus applies the same algorithm recognition, bug detection, and bug correction techniques developed in the previous chapters. Once the simplified program has been debugged, Talus attempts to determine the bug corrections in the original program that correspond to the bug edits of the simplified program.

The process of tracing edits from the simplified program back to the original program is called edit inversion. Edit inversion is facilitated by a graph representation of programs that links together s-expressions copied and transformed during program transformations. Each program transformation applied results in a new program version. The graph representation can be viewed as a semantic network
Special Forms: DEFUN, IF, QUOTE, PROGN

List Operators: CONS, CAR, CDR, LIST, APPEND

Arithmetic Operators: ADD1, SUB1, PLUS,
                   DIFFERENCE, TIMES, QUOTIENT

Boolean Connectives: AND, OR, NOT, IMPLIES

Predicates: ATOM, LISTP, NILSP, ZEROP, NUMBERP,
              EQUAL, GREATERP, LESSP, =, NULL

Constants: T, NIL, F

Numbers: Natural Numbers Only (0, 1, 2...)

List Side-Effectors: REPLAC, REPLACD, NCONC

Property List Side-Effectors: PUTPROP, GET, REMPROP

Array Side-Effectors: ASET, AREF

NOTE: Side effects and assignment are not allowed in
      conditional tests.

Figure 8-2: The Core Dialect

where nodes represent CONS cells in LISP function definitions. Nodes are linked by CAR and CDR
pointers. The semantic network is partitioned by program transformations: each partition represents a
different program version. Links between partitions represent s-expressions copied and transformed
during program transformations. Because of this similarity between partitioned semantic networks and
the program graph representation, the graph representation of each program version is called a partition.

The links between program partitions allow edits to s-expressions to be traced across program
transformations. The original student program is called the initial version and is represented by the initial
partition. The simplified program is the final version and is represented by the final partition. The goal
of edit inversion is to determine the bug edits in the initial version that correspond to bug edits in the final
version. In many cases the replacement of a buggy s-expression by the correct code in the final version
corresponds to the same bug edit in the initial version, with different code surrounding the bug edit.

To illustrate program simplification and edit inversion consider the following task and buggy
solution:

21Unfortunately, the analogy between the program graph representation and partitioned semantic networks breaks down when
quantification or class hierarchies are considered; the graph representation has neither.
Special Forms: DEFUN, IF, QUOTE, PROGN, MAPCAR, MAPCAR, MAP, MAPLIST, MAPCON, MAPC, PROG with SETQ, GO, and RETURN inside, COND, LAMBDA, FUNCTION inside mapping constructs.

List Operators: CONS, CAR, CDR, LIST, APPEND

Arithmetic Operators: ADD1, SUB1, PLUS, DIFFERENCE, TIMES, QUOTIENT

Boolean Connectives: AND, OR, NOT, IMPLIES

Predicates: ATOM, LISTP, NLISTP, ZEROP, NUMBERP, EQUAL, GREATERP, LESSP, =, NULL

Constants: T, NIL, F

Numbers: Natural Numbers Only (0, 1, 2...)

List Side-Effectors: RPLACA, RPLACD, NCONC

Property List Side-Effectors: PUTPROP, GET, REMPROP

Array Side-Effectors: ASET, AREF

NOTE: Side effects and assignment are not allowed in conditional tests or in the actual parameters of a LAMBDA expression.

Figure 8-3: The Extended Dialect

Write a program that determines the length of a proper list.

(DEFUN LENGTH (L)
 (COND ((NLISTP L) 1)
        (T (SUB1 (LENGTH (CDR L))))))

Talus applies a COND-TO-IF program simplification transform to rewrite the function body to IF-Normal Form. This transformation results in the new program version shown here:

(DEFUN LENGTH (L)
 (IF (NLISTP L)
    1
    (SUB1 (LENGTH (CDR L)))))

Figure 8-4 illustrates the initial and final partitions representing the two program versions. Dark lines in the figure are CAR and CDR pointers. PARENT pointers are not shown. Dashed lines indicate s-expressions copied during the COND-TO-IF transformation and dotted lines indicate transformed s-
expressions. Talus debugs the simplified code as described in previous chapters. The bug fixes in the final program version are:

```
(DEFUN LENGTH (L)
  (IF (NLISTP L)
    0
    (ADD1 (LENGTH (CDR L)))) ) ;Bug Fix: Replaced 1 by 0
(COND ( (NLISTP L) 1
          (T (SUB1 (LEN (CDR L)))))) ;Bug Fix: Replaced SUB1 by ADD1
```

Figure 8-4: Graph Representation of a COND-TO-IF Transformation

Figure 8-5 illustrates how edits are inverted through this COND-TO-IF transformation. The two bug corrections that apply to the simplified code, shown in the lower left hand corner, are traced back to the original code by following the COPIED-FROM links along the paths indicated by dark arrows. The atoms 0 and ADD1 replace the atoms 1 and SUB1 in both the original and transformed code. The bug fixes in the initial program version are the same as in the final program version, but the code surrounding each bug fix differs. The bug fixes in the original program are:

```
(DEFUN LENGTH (L)
  (COND ( (NLISTP L) 0
          (T (ADD1 (LENGTH (CDR L)))))) ) ;Bug Fix: Replaced 1 by 0
```

```
(COND ( (NLISTP L) 1
          (T (SUB1 (LEN (CDR L)))))) ;Bug Fix: Replaced SUB1 by ADD1
```
Edit inversion is not always this simple; inversion through procedural program transformations is more difficult than inversion through simple rewrites such as the COND-TO-IF transformation. Section 8.4 discusses how Talus takes into account the semantics of program transformations in more difficult cases of edit inversion.

![Diagram](image)

**Figure 8-5: Edit Inversion Through a COND-TO-IF Transform**

### 8.2 Debugging Programs with Side Effects

Program simplification transforms can always simplify programs from the extended dialect to the core dialect and IF-Normal Form. However, the bug detection process of Chapter 6 is not equipped to debug programs that can cause side effects. This section explains how the bug detection process has been extended to allow for side-effectors such as NCONC and PUTPROP.

Some primitives that alter shared list structure can be replaced by nondestructive equivalents. Talus checks that arguments to the primitives are fresh list structure before performing any replacements. For instance, consider the correct definition of FLAT below:
(DEFUN FLAT (L)
  (IF (NLISTP L)
      (LIST L)
      (NCONC (FLAT (CAR L)))
      (FLAT (CDR L))))

NCONC can be replaced by APPEND since its arguments are always fresh list structure. The replacement cannot be performed in the buggy solution below:

(DEFUN FLAT (L)
  (IF (NLISTP L)
      (LIST L)
      (IF (AND (ATOM (CAR L))
                (NULL (CDR L)))
          L
          (NCONC (FLAT (CAR L)) ;Bug: NCONC may alter L
                 (FLAT (CDR L))))))

since the arguments to NCONC are not always fresh list structure. In particular, if x is the list (A) then (FLAT (CONS x 'B)) alters x to (A B). The DEFUSE-DESTRUCTIVE-FNS program simplification transform, discussed in Section 8.3.3.1, determines if the replacement is safe or not.

Destructive functions cannot always be eliminated by the DEFUSE-DESTRUCTIVE-FNS transform. Other functions which perform side effects such as PUTPROP and ASET also require a different approach. For these cases, Talus applies heuristics to check for the correct use of side effects. Talus checks that side effects in student functions occur under the same conditions and to the same data structures as any side effects in paired reference functions. The data structures altered by side-effectors must be identical, but the new values to be stored in the data structures need only be functionally equivalent.

An example will clarify the checks performed. Consider the UPDATE-ALIST task and the buggy solution shown below:

Write a function that will update an alist so that the new value of the pair (KEY . VALUE) is NEW. If no pair with the key KEY exists in the alist then the pair (KEY . NEW) should be added to the end of the alist. If such a pair exists in alist it is unique. The actual alist should be modified. Each alist begins with a dummy header node NIL.

E.g.  (update 'size 'large '(NIL (color . red)
                (size . small)
                (price . expensive)))

->  ((color . red) (size . large) (price . expensive))

---

22FLAT returns (LIST L), which is fresh list structure, in the base case. Assume that recursive calls return fresh list structure. Then when L is a list FLAT returns fresh list structure since it splices together the results of two recursive calls.
(update 'size 'large' (NIL (color . red)
   (price . expensive)))

-> ((color . red) (price . expensive) (size . large))

;Buggy Solution:

(DEFUN UPDATE (AL KEY VALUE)
   (COND ((NULL (CDR AL))
           (RPLACD AL (CONS (CONS KEY VALUE) NIL)))
           ((EQUAL (CAR (CAR (CDR AL))) KEY)
               (RPLACD (CAR AL) VALUE))
           (T (UPDATE (CAR AL) KEY VALUE)))))

Both occurrences of RPLACD are not applied to fresh list structure and so no program
simplification transforms apply. UPDATE is paired with the following reference function:

(DEFUN UPDATE (KEY NEW ALIST)
   (IF (NULL (CDR ALIST))
       (RPLACD ALIST (LIST (CONS KEY NEW)))
       (IF (EQUAL (CAR (CAR (CDR ALIST))) KEY)
           (RPLACD (CAR (CDR ALIST))) NEW)
           (UPDATE KEY NEW (CDR ALIST)))))

which is normalized to:

(DEFUN UPDATE (AL KEY VALUE)
   (IF (NULL (CDR AL))
       (RPLACD AL (LIST (CONS KEY VALUE)))
       (IF (EQUAL (CAR (CAR (CDR AL))) KEY)
           (RPLACD (CAR (CDR AL)) VALUE)
           (UPDATE (CDR AL) KEY VALUE ))))

Since side effects and assignment are not allowed in conditional tests, Talus can detect missing and
extra conditional tests as described in Chapter 6. Missing conditional tests can be inserted as described in
Chapter 7. In this example there are no missing or extra conditional tests.

Chapter 6 explains how Talus determines a mapping of leaves from one binary tree to another. The
first tree represents a normalized reference function; the second represents the student function paired
with it. Symbolic values are paired for comparison by mapping leaves from the first tree to the second.

In that chapter the paired symbolic values were "compared" with functional equivalence and
termination verification conditions. However, when symbolic values contain side-effectors these
verification conditions cannot be generated. In that case, Talus applies heuristic methods to compare
symbolic values that contain side-effectors. This situation arises when we consider the base case
(NULL (CDR AL)) for the current example.

Both base cases in the student and normalized reference function govern symbolic values containing
RPLACD. The presence of RPLACD, which has no formal semantics in the Boyer-Moore Logic,
prevents the generation of verification conditions that are wffs in that logic. Even if verification
conditions could be generated, it is not enough to compare the program fragments for functional
equivalence since the correctness of the side effects must also be checked.
For the case (NULL (CDR AL)), the student's code symbolically evaluates to:

\[
\text{(RPLACD AL (CONS (CONS KEY VALUE) NIL))}
\]

The normalized reference code symbolically evaluates to

\[
\text{(RPLACD AL (LIST (CONS KEY VALUE)))}
\]

Both symbolic values contain side effects. Talus checks that that these two code fragments perform the same side effect by first comparing the top-level functions called in the two program fragments. These must be the same if either expression can cause a side effect. In this case both are RPLACD so this test is passed.

Next, Talus checks that the data structures being altered are identical, not just functionally equivalent. Since both expressions alter AL they also pass this test. Finally, the remaining arguments must be functionally equivalent. Talus generates the following functional equivalence verification condition to check that the new value of the CDR of AL will be the same in both code fragments:

\[
\text{(IMPLIES (NULL (CDR AL))}
\]
\[
\text{ (EQUAL (CONS (CONS KEY VALUE) NIL) (LIST (CONS KEY VALUE)))}
\]

The conjecture is true so Talus finds no bugs in the student's code for this case.

In the next case,

\[
\text{(AND (NOT (NULL (CDR AL))) (EQUAL (CAR (CAR (CDR AL))) KEY))}
\]

the student's code symbolically evaluates to:

\[
\text{(RPLACD (CAR AL) VALUE)}
\]

The normalized reference code symbolically evaluates to:

\[
\text{(RPLACD (CAR (CDR AL)) VALUE)}
\]

These two code fragments call the same top-level function, RPLACD, so the first test is passed. However they alter the CDRs of different data structures. Since (CAR AL) and (CAR (CDR AL)) are not identical Talus considers the student code fragment (CAR AL) buggy and replaces it with (CAR (CDR AL)). Talus also checks that the values being stored into the CDR's are functionally equivalent. Since each code fragment stores VALUE into the CDR, there are no further bugs for this case.

Talus generates functional equivalence and termination verification conditions when paired symbolic values contain no side effects. For the last case,

\[
\text{(AND (NOT (NULL (CDR AL))) (NOT (EQUAL (CAR (CAR (CDR AL))) KEY)))}
\]

the student's code symbolically evaluates to:

\[
\text{(UPDATE (CAR AL) KEY VALUE )}
\]

and the reference code symbolically evaluates to:

\[
\text{(UPDATE (CDR AL) KEY VALUE )}
\]

Talus generates the following functional equivalence verification condition:
(IMPLIES (AND (NOT (NULL (CDR AL)))
    (NOT (EQUAL (CAR (CAR (CDR AL))) KEY)))
  (EQUAL (UPDATE (CDR AL) KEY VALUE ))
  (UPDATE (CAR AL) KEY VALUE ))

which is false since the student code uses CAR instead of CDR. With this bug correction and the earlier one, the final debugged code is:

(DEFUN UPDATE (AL KEY VALUE)
  (COND ((NULL (CDR AL))
    (RPLACD AL (CONS (CONS KEY VALUE) NIL)))
    ((EQUAL (CAR (CAR (CDR AL))) KEY)
      (RPLACD (CAR (CDR AL)) VALUE))
    (T (UPDATE (CDR AL) KEY VALUE))))

The heuristics Talus uses to compare symbolic values containing side effects are overly restrictive. According to these heuristics, Talus would correctly decide that (RPLACA (APPEND X NIL) Y) and (RPLACA X Y) do not perform the same side effect. But the heuristics go awry when (RPLACA (APPEND NIL X) Y) and (RPLACA X Y) are compared: Talus incorrectly decides that these code fragments perform different side effects.

The heuristics for comparing functions with side effects are the most recent and least developed extension that has been implemented in Talus. Much more work remains for an adequate and robust handling of programs with side effects. As another example of the limitations of the current heuristics consider the following correct solution to UPDATES:

(DEFUN UPDATE (AL KEY VALUE)
  (IF (NULL (CDR AL))
    (CAR (LIST (RPLACD AL (LIST (CONS KEY VALUE)))))
    (IF (EQUAL (CAR (CAR (CDR AL))) KEY)
      (RPLACD (CAR (CDR AL)) VALUE)
      (UPDATE (CDR AL) KEY VALUE )))))

When Talus considers the case (NULL (CDR AL)) the student code fragment

(CAR (LIST (RPLACD AL (LIST (CONS KEY VALUE))))))

will be compared to the normalized code fragment

(RPLACD AL (LIST (CONS KEY VALUE))))

Talus will consider the student’s code fragment to be buggy since both functions contain side-effectors (RPLACD) but the top-level function in the student’s code is CAR while the top-level function in the normalized reference code fragment is RPLACD.

Heuristics for comparing PROGN expressions have also been implemented. Suppose Talus is comparing the student code fragment:

(PROGN s₁ . . . sₙ)

to the reference code fragment

---

23In the first code fragment X is copied by APPEND so it is not altered by RPLACA.
(PROGN \(r_1 \ldots r_m\))

Talus pairs expressions in the student code fragment with those in the reference code fragment. Then each pair containing side effectors is compared with the heuristics described previously. The remaining pairs are compared with functional equivalence and termination verification conditions. Student expressions \(s_i\) that are unpaired are considered unnecessary; reference expressions \(r_i\) that are unpaired are considered missing.

Before explaining how the heuristics pair terms in PROGN expressions, an example will be given to clarify the overall process. Consider the INTERSECTION task and the buggy solution shown below:

Write a function that takes two sets, represented as lists, and returns their intersection.

(DEFUN INTERSECT (X Y)
  (PROGN
    (MARK X)
    (FILTER Y)))

(DEFUN MARK (L)
  (COND ((NULL L) NIL)
    (T (PROGN (PUTPROP
                     (CAR L) ;Symbol
                     T ;Value
                     'FLAG) ;Property
                     (MARK (CDR L)))))

(DEFUN FILTER (L)
  (COND ((NULL L) NIL)
    ((NULL (GET (CAR L) 'FLAG))
     (FILTER (CDR L)))
    (T (FILTER (CDR L)))))

Talus knows of three algorithms for the INTERSECTION task: RECURSIVE-INTERSECTION, ITERATIVE-INTERSECTION, and MARKER-INTERSECTION. Talus selects the last algorithm as best matching the student’s solution. That algorithm’s reference functions are shown below:

(DEFUN INTERSECT (A B)
  (PROGN
    (CLEAR-MARKERS B)
    (SET-MARKERS A)
    (COLLECT-MARKED-ELEMENTS B)))

(DEFUN CLEAR-MARKERS (LIS)
  (COND ((NULL LIS) NIL)
    (T (PROGN (REMPROP (CAR LIS) '?MARKER)
                  (CLEAR-MARKERS (CDR LIS)))))

(DEFUN SET-MARKERS (LIS)
  (COND ((NULL LIS) NIL)
    (T (PROGN (PUTPROP (CAR LIS) '?VALUE '?MARKER)
                  (SET-MARKERS (CDR LIS))))))
(DEFUN COLLECT-MARKED-ELEMENTS (LIS)
  (COND ((NULL LIS) NIL)
    ((GET (CAR LIS) 'MARKER)
      (CONS (CAR LIS) (COLLECT-MARKED-ELEMENTS (CDR LIS))))
    (T (COLLECT-MARKED-ELEMENTS (CDR LIS)))))

The identifiers 'MARKER and 'VALUE in the functions are placeholders for the property list indicators in the student's solution. They are bound to the values FLAG and T in the student's solution during the normalization process. Talus examines occurrences of GET and PUTPROP in the student solution to determine the bindings to use. The normalized reference functions are:

(DEFUN INTERSECT (X Y)
  (PROGN
    (CLEAR-MARKERS Y)
    (MARK X)
    (FILTER Y)))

(DEFUN CLEAR-MARKERS (L)
  (COND ((NULL L) NIL)
    (T (PROGN (REMPROP (CAR L) 'FLAG)
      (CLEAR-MARKERS (CDR L))))))

(DEFUN MARK (L)
  (COND ((NULL L) NIL)
    (T (PROGN (PUTPROP (CAR L) 'T 'FLAG)
      (MARK (CDR L))))))

(DEFUN FILTER (L)
  (COND ((NULL L) NIL)
    ((GET (CAR L) 'FLAG)
      (CONS (CAR L) (FILTER (CDR L))))
    (T (FILTER (CDR L))))))

There is only one case to consider for the function INTERSECT. For that case the student function symbolically evaluates to:

(PROGN
  (MARK X)
  (FILTER Y))

and the reference function symbolically evaluates to:

(PROGN
  (CLEAR-MARKERS Y)
  (MARK X)
  (FILTER Y))

In comparing these two PROGNs Talus maps the expression (CLEAR-MARKERS Y) in the reference code to EXTRA and then pairs the expressions (MARK X) and (FILTER Y) in each code fragment to its identical s-expression in the other code fragment. (CLEAR-MARKERS Y) is considered missing from the student's code so the following hint is generated:

HINT: The expression (CLEAR-MARKERS Y) is missing from the function INTERSECT and should be inserted.

Since the other paired s-expressions are identical there are no further bugs in INTERSECT.
The function CLEAR-MARKERS is considered missing from the student's solution. Without this function, INTERSECT can return incorrect values when applied more than once since old property list markings are never cleared away. Talus supplies the definition for CLEAR-MARKERS from the normalized reference function of the same name:

```
(defun clear-markers (l)
  (cond ((null l) nil)
         (t (progn (remprop (car l) 'flag)
                 (clear-markers (cdr l))))))
```

There are two cases to consider for the function MARK: either L is NULL or it is not. For the first case both reference and student functions symbolically evaluate to NIL. Since both code fragments are side effect free and identical they obviously satisfy all verification conditions.

When L is non-NIL the student code's symbolically evaluates to

```
(progn (putprop (car l) t 'flag)
        (mark (cdr l)))
```

and the reference code symbolically evaluates to

```
(progn (putprop (car l) 't 'flag)
        (mark (cdr l)))
```

Talus applies heuristics to match up the top-level s-expressions in each PROGN. The two expressions containing PUTPROP and the two recursive calls to MARK are paired. Talus finds no bugs for the first pair:

```
(putprop (car l) t 'flag) ; Student Code Fragment
(putprop (car l) 't 'flag) ; Reference Code Fragment
```

since they both begin with PUTPROP, both place properties on (CAR L), and since the remaining arguments are functionally equivalent:

```
(implies (not (null l))
          (equal t 't))
--> t
```

```
(implies (not (null l))
          (equal 'flag 'flag))
--> t
```

Talus finds no bugs in the last two expressions (both (MARK (CDR L))) since they are identical. If they had not been identical functional equivalence and termination verification conditions would have been generated to compare them.

In the last function, FILTER, the symbolic values in both the reference function and the student function cannot cause side effects. Since no side-effectors can occur, Talus generates functional equivalence and termination verification conditions as usual. One bug is found:

**HINT:** There is a bug in the function call (filter (cdr l)) in the expression (filter (cdr l)) inside the function FILTER. If you use the function call (cons (car l) (filter (cdr l))) instead, the bug will go away.
The final debugged code is shown below:

(DEFUN INTERSECT (X Y)
  (PROGN
    (PROGN (CLEAR-MARKERS Y) (MARK X))
    (FILTER Y)))

(DEFUN MARK (L)
  (COND ((NULL L) NIL)
    (T (PROGN (PUTPROP (CAR L) T 'FLAG)
             (MARK (CDR L))))))

(DEFUN FILTER (L)
  (COND ((NULL L) NIL)
    ((NULL (GET (CAR L) 'FLAG))
     (FILTER (CDR L)))
    (T
     (CONS (CAR L) (FILTER (CDR L))))))

**Missing Definition for CLEAR-MARKERS.**

(DEFUN CLEAR-MARKERS (L)
  (COND ((NULL L) NIL)
    (T (PROGN (REMPROP (CAR L) 'FLAG)
             (CLEAR-MARKERS (CDR L))))))

Talus determines the best mapping of top-level expressions between two paired PROGNs

(PROGN \(s_1, \ldots, s_n\))

and

(PROGN \(r_1, \ldots, r_m\))
in a two step process. The first step is to generate all possible mappings, subject to certain constraints. The second step is to score each mapping and then to choose the best mapping.

In the first step all possible pairings are generated, subject to an ordering constraint:

\textbf{If} \(s_i\) \textbf{maps to} \(r_u\), \textbf{and} \(s_j\) \textbf{maps to} \(r_v\), where \(j > i\), \textbf{then} \(v > u\).

This constraint requires the side effects that occur in the first PROGN occur in the same sequence as the second.

The heuristic scoring function measures how bad each mapping is. 0 is a perfect mapping. The following penalties are assessed for each pair \((s_i, r_u)\) in the mapping:

- 0.0 for identical expressions
- 0.5 for expressions that are not identical but that use the same side-effectors
- 1.0 for expressions that use different side-effectors (times the number of different side-effectors)

Top-level expressions in the reference PROGN can map to EXTRA with these penalties:

- 1.5 for missing expressions at the beginning of an expression sequence
2.0 for missing expressions not at the beginning of an expression sequence
Top-level expressions in the student's PROGN can map to EXTRA with a penalty of 3.

8.3 Program Simplification Transforms
This section discusses the program simplification transforms of Talus. These transforms have three
purposes in Talus:

- To Rewrite Function Definitions to IF-Normal Form. Discussed in Section 8.3.1.
- To Reduce Programs from the Extended Dialect to the Core Dialect. Discussed in Section
  8.3.2.
- To Simplify Algorithm Recognition. Discussed in Section 8.3.3.
All transforms are equivalence-preserving. Section 8.3.4 discusses the order in which these transforms
are applied.

8.3.1 Rewriting Conditional Expressions to IF-Normal Form
The transforms in this section rewrite conditional expressions so that they are in IF-Normal Form.
The COND-TO-IF transform rewrites conditional expressions with COND to equivalent expressions with
IF. The REMOVE-IFS-EMBEDDED-IN-FN-CALLS transform moves IF expressions inside function
calls outside of them. The REMOVE-IFS-EMBEDDED-IN-IF-TESTS transform removes an IF
expression that is inside the conditional test of another IF expression by rewriting the outer IF expression.

8.3.1.1 Replacing COND with IF
The COND-TO-IF transform rewrites COND expressions to IF expressions. An example was
provided in Figure 8-4. In general, the rewrite looks like:

(COND (test_1 value_1)
       (test_2 value_2)
       ...
       (test_n value_n))

----> (IF test_1 value_1
         (IF test_2 value_2
            ...
            (IF test_n value_n NIL))))

8.3.1.2 Removing IF from Function Calls
The REMOVE-IFS-EMBEDDED-IN-FN-CALLS transform rewrites IF expressions so that they
occur outside of function calls. The following rewrite rule is applied to expressions:

(fn (IF test if-true if-false))
----> (IF test (fn if-true) (fn if-false))

---

Equivalence is preserved with respect to program values returned; efficiency may be lost.
Since functions can have any number of arguments, the general form of this rule is:

\[(fn \ arg_1 \ldots \ arg_i (if \ test \ if-true \ if-false) \ arg_{i+2} \ldots \ arg_n)\]

\[\Rightarrow\]

\[(if \ test \ (fn \ arg_1 \ldots \ arg_i \ if-true \ arg_{i+2} \ldots \ arg_n)
  \quad (fn \ arg_1 \ldots \ arg_i \ if-false \ arg_{i+2} \ldots \ arg_n))\]

8.3.1.3 Removing IF from Conditional Tests

The REMOVE-IFS-EMBEDDED-IN-IF-TESTS transform rewrites expressions where one IF expression occurs in the conditional test part of another IF expression. In general, the rewrite rule is:

\[(if \ (if \ test \ if-true_{inner} \ if-false_{inner})
  \quad if-true_{outer}
  \quad if-false_{outer})\]

\[\Rightarrow\]

\[(if \ (or \ (and \ test \ if-true_{inner})
  \quad (and \ (not \ test) \ if-false_{inner}))
  \quad if-true_{outer}
  \quad if-false_{outer})\]

but Talus also recognizes two special cases:

\[(if \ (if \ test \ T \ NIL) \ if-true_{outer} \ if-false_{outer})\]

\[\Rightarrow\]

\[(if \ test \ if-true_{outer} \ if-false_{outer})\]

and

\[(if \ (if \ test \ NIL \ T) \ if-true_{outer} \ if-false_{outer})\]

\[\Rightarrow\]

\[(if \ (not \ test) \ if-true_{outer} \ if-false_{outer})\]

8.3.2 Reduction from Extended to Core Dialect

The transforms in this section simplify programs in the extended dialect to the core dialect. The PROG-REMOVAL transform rewrites expressions containing PROG into equivalent expressions that do not contain PROG. Auxiliary function definitions are introduced in the process. The MAP-TO-PROG transform rewrites expressions containing mapping constructs into equivalent expressions that contain PROG. (These in turn are simplified by the PROG-REMOVAL transform.) Finally the LAMBDA-EXPANSION transform expands out LAMBDA expressions.

8.3.2.1 PROG Removal

PROG Expressions can be algorithmically translated to equivalent recursive functions that do not use PROG by the PROG-REMOVAL transform. The algorithm breaks a PROG expression apart into blocks that are demarcated by the boundaries of the PROG expression and by tags in the PROG expression. Implicit jumps between blocks that are contiguous are added. Implicit PROGNS are made explicit. SETQs are removed by performing substitutions that effect the same variable updates. GO and RETURN statements are annotated to indicate the variable updates in effect when they are reached. The resulting blocks are rewritten as LISP functions. Iterative loops in PROG blocks become tail recursive functions. GOs are replaced by function calls with actual arguments determined by the variable update annotations. RETURNs are replaced by the value of the variable they return. Simplifications during the transformation remove dead code, unnecessary PROGNS, and variables not referred to.
For example, consider the following solution to the MAX task:

Write a function which finds the maximum of a list of natural numbers. Return 0 for an empty list.

(DEFUN MAX (L) (MAX1 L 0))

(DEFUN MAX1 (L SO-FAR)
     (PROG ()
       LOOP
       (IF (NULL L) (RETURN SO-FAR) NIL)
       (IF (GREATERP (CAR L) SO-FAR)
           (SETQ SO-FAR (CDR L)) ; Bug: should be CAR NIL)
       (SETQ L (CDR L))
       (GO LOOP)))

Talus will apply a PROG-REMOVAL transform to MAX1 to eliminate the PROG. First Talus breaks the PROG up into two blocks. The first block consists of all variable initializations and is empty since there are none here. That block starts at the beginning of the PROG and ends right before the label LOOP. The second block is the remainder of the PROG starting with LOOP.

The first step in removing a PROG is to break the PROG apart into blocks. Talus names the variable initialization block INIT and names the other blocks by the tags that begin them. Here are the two blocks in the example above:

INIT: [ ] ; empty.

LOOP: [ (IF (NULL L) (RETURN SO-FAR) NIL)

       (IF (GREATERP (CAR L) SO-FAR)
           (SETQ SO-FAR (CDR L))
           NIL)

       (SETQ L (CDR L))

       (GO LOOP)

       (RETURN NIL) :Added ]

Notice that Talus has added a (RETURN NIL) to the end of the LOOP block. This RETURN is implicitly present at the end of every PROG. Next Talus links the blocks together:

INIT: [ (GO LOOP) ]

LOOP: [ (IF (NULL L) (RETURN SO-FAR) NIL)

       (IF (GREATERP (CAR L) SO-FAR)
           (SETQ SO-FAR (CDR L))
           NIL) ]
(setq l (cdr l))

(go loop)

(return nil)

]

Then implicit PROGNs are made explicit:

init: [ (go loop) ]

loop: [ (if (null l)
   (progn (return so-far)
     (if (greaterp (car l) so-far)
       (progn (setq so-far (cdr l))
         (setq l (cdr l))
         (go loop)
         (return nil))
       (progn nil
         (setq l (cdr l))
         (go loop)
         (return nil)))]

(progn nil
  (if (greaterp (car l) so-far)
    (progn (setq so-far (cdr l))
      (setq l (cdr l))
      (go loop)
      (return nil))
    (progn nil
      (setq l (cdr l))
      (go loop)
      (return nil)))]

]

Then dead code is eliminated to simplify PROGNs:

init: [ (go loop) ]

loop: [ (if (null l)
   (return so-far)
   (if (greaterp (car l) so-far)
     (progn (setq so-far (cdr l))
       (setq l (cdr l))
       (go loop))
     (progn (setq l (cdr l))
       (go loop)))]

]

Then SETQs are eliminated by annotating the GO statements with the same variable updates that would be effected by the SETQs:

init: [ (go loop nil) ]
LOOP: [ (IF (NULL L)
  (RETURN SO-FAR)
  (IF (GREATERP (CAR L) SO-FAR)
    (GO LOOP ((L . (CDR L)) (SO-FAR . (CDR L))))
    (GO LOOP ((L . (CDR L)))))
  )]

Then each block is replaced by a function definition containing all possible formal variables. Function calls in the definitions replace GOs in the PROG blocks:

INIT: [ (DEFUN INIT (L SO-FAR)
  (LOOP (L . L) (SO-FAR . SO-FAR)))
  ]

LOOP: [ (DEFUN LOOP (L SO-FAR)
  (IF (NULL L)
    SO-FAR
    (IF (GREATERP (CAR L) SO-FAR)
      (LOOP (L . (CDR L)) (SO-FAR . (CDR L)))
      (LOOP (L . (CDR L)) (SO-FAR . SO-FAR)))
    )]

Next unnecessary formal variables are eliminated. No change occurs in this example. Then function calls are modified so that the variable being updated is represented implicitly by the position of each actual argument. The next step is to ensure that all function names are unique. Talus does this by adding one or more underscores in front of each function identifier until there are no conflicts with existing function identifiers:

_INIT: [ (DEFUN _INIT (L SO-FAR) (_LOOP L SO-FAR)) ]

_LOOP: [ (DEFUN _LOOP (L SO-FAR)
  (IF (NULL L)
    SO-FAR
    (IF (GREATERP (CAR L) SO-FAR)
      (_LOOP (CDR L) (CDR L))
      (_LOOP (CDR L) SO-FAR)))
    )]

The function body of _INIT replaces the PROG in MAX1 in the initial version. The final version of the program after the PROG removal transformation is:

(DEFUN MAX (L) (MAX1 L 0))

(DEFUN MAX1 (L SO-FAR) (_LOOP L SO-FAR))

(DEFUN _LOOP (L SO-FAR)
  (IF (NULL L)
    SO-FAR
    (IF (GREATERP (CAR L) SO-FAR)
      (_LOOP (CDR L) (CDR L))
      (_LOOP (CDR L) SO-FAR)))
    )

This code is debugged to become:

(DEFUN MAX (L) (MAX1 L 0))

(DEFUN MAX1 (L SO-FAR) (_LOOP L SO-FAR))
(DEFUN LOOP (L SO-FAR)
  (IF (NULL L)
      SO-FAR
      (IF (GREATERP (CAR L) SO-FAR)
          (_LOOP (CDR L) (CAR L)) ; Bug Fix: CAR replaces CDR
          (_LOOP (CDR L) SO-FAR)))))

The edits in the final program version are traced back to the initial program version:

(DEFUN MAX (L) (MAX1 L 0))

(DEFUN MAX1 (L SO-FAR)
  (PROG ()
    LOOP
    (IF (NULL L) (RETURN SO-FAR) NIL)
    (IF (GREATERP (CAR L) SO-FAR)
        (SETQ SO-FAR (CAR L)) ; Bug Fix: CAR replaces CDR
        NIL)
    (SETQ L (CDR L))
    (GO LOOP)))

3.3.2.2 Simplification of Mapping Functions

Mapping functions express common patterns of iteration over proper lists. Since PROG is more general, all the common mapping functions (MAP, MAPCAR, MAPC, MAPCAR, MAPCON, and MAPLIST) can be expressed with PROGs. Talus has stored templates for rewriting mapping functions to PROGs.

The MAP-FN-TO-PROG transform rewrites a mapping function to a PROG. The general form of the rewrite rule used is:

(MAP-FN (function fn) list) -> TEMPLATE
or
(MAP-FN (quote fn) list) -> TEMPLATE

where MAP-FN is one of the mapping functions and fn is a lambda expression or the name of a function. list is the list being mapped over. TEMPLATE is the resulting PROG template with the value of fn and list instantiated. The templates are:

For MAPCAR,

(PROG (_, AC NIL) (_, LEFT list))
  LOOP
    (IF (NULL LEFT) (RETURN AC))
    (SETQ AC (APPEND AC (LIST (fn (CAR LEFT)))))
    (SETQ LEFT (CDR LEFT))
    (GO LOOP))
For MAPLIST,

(PROG ((AC NIL) (LEFT list))
   LOOP
     (IF (NULL LEFT) (RETURN AC))
     (SETQ AC (APPEND AC (LIST (fn LEFT))))
     (SETQ LEFT (CDR LEFT))
     (GO LOOP))

For MAPC,

(PROG ((LEFT list))
   LOOP
     (IF (NULL LEFT) (RETURN list))
     (fn (CAR LEFT))
     (SETQ LEFT (CDR LEFT))
     (GO LOOP))

For MAP,

(PROG ((LEFT list))
   LOOP
     (IF (NULL LEFT) (RETURN list))
     (fn LEFT)
     (SETQ LEFT (CDR LEFT))
     (GO LOOP))

For MAPCAN,

(PROG ((AC NIL) (LEFT list))
   LOOP
     (IF (NULL LEFT) (RETURN AC))
     (SETQ AC (NCONC AC (fn (CAR LEFT))))
     (SETQ LEFT (CDR LEFT))
     (GO LOOP))

For MAPCON,

(PROG ((AC NIL) (LEFT list))
   LOOP
     (IF (NULL LEFT) (RETURN AC))
     (SETQ AC (NCONC AC (fn LEFT)))
     (SETQ LEFT (CDR LEFT))
     (GO LOOP))

8.3.2.3 LAMBDA Expansion

The LAMBDA-EXPANSION transform expands out LAMBDA expressions provided that no side effects can occur in evaluating the parameters to the LAMBDA expression. For example a LAMBDA-EXPANSION transform changes the following function definition:
(DEFUN REM (X L) ; Remove Occurrences of X from L
   ((LAMBDA (CAR-L CDR-L)
      (IF (NLISTP L)
         NIL
         (IF (EQUAL X CAR-L)
            (REM X CDR-L)
            (CONS CAR-L (REM X CDR-L))))))
   (CAR L) (CDR L)))

8.3.3 Transforms that Simplify Algorithm Recognition

The transforms in this section facilitate algorithm recognition by

- Removing Irrelevant Side-Effectors and PROGNs.

- Removing Irrelevant Function Definitions.

- Converting AND/OR to IF.

Function definitions, side-effectors, and PROGNs are considered irrelevant if they can easily be eliminated without changing the results returned by a solution. Algorithm recognition is improved by removing irrelevant constructs since then only essential differences remain.

8.3.3.1 Transforms that Eliminate Destructive Functions

The DEFUSE-DESTRUCTIVE-FNS transform removes irrelevant side-effectors. For example, NCONC can be replaced by APPEND if the arguments to the APPEND are fresh list structure. In general, destructive functions that alter list structure (RPLACA, RPLACD, and NCONC) are replaced by their nondestructive equivalents when their arguments are fresh list structure. The results returned will be the same, only the number of CONS cells used will differ. Consider the following solution to REVERSE,

(DEFUN REV (L)
   (IF (LISTP L)
      (NCONC (REV (CDR L)) (LIST (CAR L)))
      NIL))

Talus determines that REV always returns fresh list structure by examining the terminations and recursive calls. (LIST always returns fresh list structure, and NCONC always returns fresh list structure when its arguments are fresh list structure.) Then Talus replaces NCONC by APPEND in a DEFUSE-DESTRUCTIVE-FNS program simplification transformation. The transformed function is debugged and the edits are traced back to the original program. In this case there are no bugs.

Talus determines those functions that return fresh list structure by an iterative process. Certain primitives such as CONS and LIST always return fresh list structure. Auxiliary variables initialized to NIL and updated by CONS or APPEND are always fresh list structure. By examining function calls and the occurrence of side-effectors Talus can determine that pure LISP functions that only call other
functions that return fresh list structure must themselves return fresh list structure. Side-effectors such as NCONC can also be shown to return fresh list structure if their arguments are fresh list structure.

8.3.3.2 Handling PROGN

As explained in Section 8.2, special heuristics are needed to compare expressions containing PROGNs. These are not appropriate when the PROGN is unnecessary since the heuristics assume that some expressions in the PROGN are being performed for side effect only. The REMOVE-TRIVIAL-PROGNs transform removes PROGNs that are unnecessary so that the PROGN heuristics will only apply when comparing expressions that really depend on side effects. The rewrite rule applied is:

\[(\text{PROGN} \ exp) \quad \rightarrow \quad exp\]

On the other hand, PROGNs must be made explicit when they are needed so that the PROGN heuristics will apply. Many other rewrites assume that a function's body or the action part of a conditional is only one s-expression, so implicit PROGNs must be made explicit prior to applying these rewrites. The REVEAL-PROGNs-IN-DEFINITIONS transform rewrites function definitions with implicit PROGNs so that the PROGNs are made explicit:

\[(\text{DEFUN} \ fn \ vars \ exp_1 \ exp_2 \ \ldots \ exp_n)\]

\[\rightarrow\]

\[(\text{DEFUN} \ fn \ vars\n\quad (\text{PROGN} \ exp_1 \ exp_2 \ \ldots \ exp_n))\]

The REVEAL-PROGNs-IN-CONDS transform rewrites COND clauses with implicit PROGNs so that the PROGNs are made explicit:

\[(\text{COND} \ \ldots \n\quad (\text{test} \ exp_1 \ exp_2 \ \ldots \ exp_n) \n\quad \ldots)\]

\[\rightarrow\]

\[(\text{COND} \ \ldots \n\quad (\text{test} \ (\text{PROGN} \ exp_1 \ exp_2 \ \ldots \ exp_n)) \n\quad \ldots)\]

8.3.3.3 Eliminating Unnecessary Function Definitions

The MACROEXPANSION transform allows nonrecursive functions that only call other functions or primitives to be treated as macros and expanded inline wherever they are called. The parameters in function calls to the "macro" must not contain side-effectors or assignments for this transform to apply. For an example of this transform, consider this buggy solution to the MEMTREE task:

\[(\text{DEFUN} \ MEMTR \ (X \ TR) \ (IN \ X \ (SMASH \ TR)))\]

\[(\text{DEFUN} \ IN \ (X \ L) \n\quad (\text{IF} \ (\text{EQUAL} \ X \ (\text{CAR} \ L)) \ \text{T} \ (IN \ X \ (\text{CAR} \ L))))\]
(DEFUN SMASH (TREE) (SMASH* TREE NIL))

(DEFUN SMASH* (TR ANS)
  (IF (ATOM TR)
    ANS
    (SMASH* (CAR TR) (SMASH* (CDR TR) ANS)))))

The function SMASH serves no purpose other than to call SMASH*. Thus, the call to SMASH in MEMTR can be replaced by a call to SMASH*. The result of the MACROEXPANSION transform is:

(DEFUN MEMTR (X TR) (IN X (SMASH* TR NIL)))

(DEFUN IN (X L)
  (IF (EQUAL X (CAR L)) T (IN X (CAR L))))

(DEFUN SMASH* (TR ANS)
  (IF (ATOM TR)
    ANS
    (SMASH* (CAR TR) (SMASH* (CDR TR) ANS)))))

Note that the function SMASH has been removed from the solution. The MACROEXPANSION transform aids the algorithm recognition process by removing unnecessary "macro" functions such as SMASH. If these functions were not removed they would degrade the quality of the best function mapping since they would map to EXTRA and incur a penalty.

8.3.3.4 Implicit Conditionals in AND and OR

The REWRITE-IMPLIED-CONDITIONALS transform rewrites AND and OR expressions to be conditional tests when they occur outside of conditional tests. For example, this transform changes the following correct solution to MEMTREE:

(DEFUN MEMTR (X TR)
  (IF (ATOM TR)
    (EQUAL X TR)
    (OR (MEMTR X (CAR TR)) ;Implicit Conditional Expression
        (MEMTR X (CDR TR))))))

to

(DEFUN MEMTR (X TR)
  (IF (ATOM TR)
    (EQUAL X TR)
    (IF (MEMTR X (CAR TR)) ;Explicit Conditional Expression
        (MEMTR X (CAR TR))
        (MEMTR X (CDR TR))))))

This simplifies algorithm recognition by making all conditional expressions explicit, as they are in the reference functions.

The rule for AND expressions is:
(AND \(\text{exp}_1\) \(\text{exp}_2\) \ldots \(\text{exp}_{n-1}\) \(\text{exp}_n\))

---->

(IF \(\text{exp}_1\)
  (IF \(\text{exp}_2\)
    \ldots
    (IF \(\text{exp}_{n-1}\)
      \(\text{exp}_n\)
      NIL)
    NIL)
  NIL)
and the rule for OR expressions is:

(OR \(\text{exp}_1\) \(\text{exp}_2\) \ldots \(\text{exp}_{n-1}\) \(\text{exp}_n\))

---->

(IF \(\text{exp}_1\) \(\text{exp}_1\)
  (IF \(\text{exp}_2\) \(\text{exp}_2\)
    \ldots
    (IF \(\text{exp}_{n-1}\)
      \(\text{exp}_{n-1}\)
      \(\text{exp}_n\))))

All expressions \(\text{exp}\) in both rewrites above must be side effect free except for the very last expression \(\text{exp}_n\). Otherwise the transformed code will have side effects in conditional tests, which is not allowed.

The **REMOVE-ORS-IN-IF-TESTS** transform removes OR expressions from conditional tests by applying the following rewrite rule:

(IF (OR \(\text{disjunct}_1\) \(\text{disjunct}_2\)) \(\text{if-true}\) \(\text{if-false}\))

---->

(IF \(\text{disjunct}_1\) \(\text{if-true}\) (IF \(\text{disjunct}_2\) \(\text{if-true}\) \(\text{if-false}\)))

This simplifies conditional expressions by decomposing disjunctions into separate conditional expressions; this simplification aids algorithm recognition by rewriting student functions to be more similar to reference functions.

### 8.3.4 Ordering of Transform Applications

**Talus** applies each transform repeatedly to the student's solution until it no longer applies. The following steps occur in program simplification:

1. Mapping functions are expanded to PROGs.
2. PROGs are simplified to recursive functions.
3. PROGNs are made explicit.
4. CONDs are expanded to IFs.
5. Conjunctions and disjunctions in symbolic values are rewritten as IF-expressions.
6. Macros are expanded.
7. IF expressions embedded in function calls or conditional tests are rewritten.
8. Trivial PROGNs are removed.
9. Disjunctions in conditional tests are removed.
10. Functions are analyzed to see if they return fresh list structure.
11. The primitives RPLACA, RPLACD, and NCONC are replaced by pure LISP equivalents if their arguments are fresh list structure.

The exact order in which transforms are applied is:

1. REVEAL-PROGNS-IN-DEFINITIONS
2. MAP-FN-TO-PROG
3. REVEAL-PROGNS-IN-CONDS
4. COND-TO-IF
5. REWRITE-IMPLICIT-CONDITIONALS
6. PROG-REMOVAL
7. LAMBDA-EXPANSION
8. MACROEXPANSION
9. REMOVE-IFS-EMBEDDED-IN-FN-CALLS
10. REMOVE-IFS-EMBEDDED-IN-IF-TESTS
11. REMOVE-TRIVIAL-PROGNs
12. REMOVE-ORS-IN-IF-TESTS
13. DEFUSE-DESTRUCTIVE-FNS

The DEFUSE-DESTRUCTIVE-FNS transform cannot be applied until functions have been analyzed to see if they return fresh list structure. This analysis is simplified by the sequencing of transforms since no conditionals, macros, or LAMBDAAs can occur inside function calls when the analysis is performed.

8.4 Edit Inversion

All the program simplification transformations discussed in Section 8.3 are represented by graphs as explained in Section 8.1.2. In many cases edits can be inverted simply by following COPIED-FROM and TRANSFORMED-FROM links across program transformations. These links tie together s-expressions that have been copied or transformed during program simplification. This means of edit inversion is called simple edit inversion and was illustrated earlier in Figure 8-5. Section 8.4.1 discusses the applicability of simple edit inversion and when it can fail. Section 8.4.2 discusses special methods that override default edit inversion when it is inappropriate.

8.4.1 Default Edit Inversion

Edit inversion defaults to simple edit inversion, as described in Section 8.1.2, if there are no special case edit inversion methods set up for a transform. These edit inversion methods intercept certain kinds of edits for a particular transform, overriding the default method. The kinds of special cases that arise and how they are handled are discussed in Section 8.4.2.
Simple edit inversion is adequate for all transforms that are one-to-one transforms. A transformation from partition \( a \) to partition \( b \) is one-to-one if:

- Every conditional test in \( a \) other than \( T \) or \( NIL \) occurs exactly once in \( b \), and
- Every function termination or function recursion in \( a \) occurs exactly once in \( b \), but atomic s-expressions can be changed.
- Only special forms are added or deleted in the transformation.

Transformations that merely rearrange s-expressions are one-to-one. The \( \text{COND-TO-IF} \) transform is one-to-one since it merely replaces \( \text{COND} \) by \( \text{IF} \) and rearranges s-expressions accordingly. The \( \text{PROG-REMOVAL} \) transform is not one-to-one since auxiliary functions are introduced.

One-to-one transforms need not be equivalence preserving since the definition only refers to the simple invertibility of edits. The transforms listed previously are intended to be equivalence preserving, although this has not been formally proven and would be difficult to do so. Every node representing a side effect free function call, variable reference, or constant in a symbolic value in a partition will be tied by a single \( \text{COPIED-TO} \) or \( \text{TRANSFORMED-TO} \) link to a node in the next partition, if that partition results from the application of a one-to-one transform.

The following transforms are one-to-one:
- \( \text{REVEAL-PROGNS-IN-DEFINITIONS} \). Only adds \( \text{PROGN} \), a special form.
- \( \text{REVEAL-PROGNS-IN-COND} \). Also adds \( \text{PROGN} \).
- \( \text{COND-TO-IF} \). Replaces \( \text{COND} \) by \( \text{IF} \).
- \( \text{REMOVE-TRIVIAL-PROGNS} \). Removes \( \text{PROGN} \).
- \( \text{DEFUSE-DESTRUCTIVE-FNS} \). Performs atomic substitutions, e.g. \( \text{NCONC} \) is replaced by \( \text{APPEND} \). The original atoms are connected to their substitutions by \( \text{TRANSFORMED-TO} \) links.

The following transforms are not one-to-one:
- \( \text{REMOVE-IFS-EMBEDDED-IN-FN-CALLS} \). Nonatomic function arguments can be duplicated.
- \( \text{REMOVE-IFS-EMBEDDED-IN-IF-TESTS} \). Conditional tests are duplicated.
- \( \text{REWRITE-IMPLICIT-CONDITIONALS} \). Conditional tests can be duplicated.
- \( \text{MAP-FN-TO-PROG} \). Conditional tests are introduced.
- \( \text{PROG-REMOVAL} \). Auxiliary functions are introduced.
- \( \text{LAMBDA-EXPANSION} \). Actual arguments may be copied more than once.
- \( \text{MACROEXPANSION} \). The macro body may be copied more than once.
- \( \text{REMOVE-ORS-IN-IF-TESTS} \). Conditional tests are introduced.

The \( \text{REWRITE-IMPLICIT-CONDITIONALS} \) transform is one-to-one for the rewrite rule that applies to \( \text{AND} \) expressions that occur outside of conditionals:
(AND \(exp_1\)
    \(exp_2\)
    \(\ldots\)
    \(exp_{n-1}\)
    \(exp_n\))

\(\implies\)

(IF \(exp_1\)
    (IF \(exp_2\)
        \(\ldots\)
        (IF \(exp_{n-1}\)
            \(exp_n\)
            NIL)
        NIL)
    NIL)

but is not for the rule that applies to OR expressions that occur outside of conditionals:

(OR \(exp_1\)
    \(exp_2\)
    \(\ldots\)
    \(exp_{n-1}\)
    \(exp_n\))

\(\implies\)

(IF \(exp_1\) \(exp_1\)
    (IF \(exp_2\) \(exp_2\)
        \(\ldots\)
        (IF \(exp_{n-1}\)
            \(exp_{n-1}\)
            \(exp_n\)))

since the conditional expressions are copied more than once.

The PROG-REMOVAL transform is not one-to-one since implicit variable initializations and updates are made explicit in newly created data structures in the transformed program version. For example, the following code:

\[
\text{(DEFUN REV (L)}
\text{ (PROG (ANS)
  LOOP
    (IF (NULL L) (RETURN ANS) NIL)
    (SETQ ANS (CONS (CAR L) ANS))
    (GO LOOP)))
\]

implicitly initializes ANS to NIL, the default initialization, and implicitly updates L to L (no change) each time through the loop. L should of course be updated to (CDR L), so that is a bug in the program above. The PROG-REMOVAL transform changes the code above to:
(DEFUN REV (L) (_LOOP L NIL))

(DEFUN _LOOP (L ANS)
  (IF (NULL L) ANS (_LOOP L (CONS (CAR L) ANS))))

where the underlined expressions are newly created after the transformation, and their node representations have no TRANSFORMED-FROM or COPIED-FROM links back to the previous partition. All bug edits to the underlined s-expressions cannot be propagated to the original partition without the intervention of special case edit inversion methods.

Both LAMBDA-EXPANSION and MACROEXPANSION are not one-to-one since expressions can be copied during the transformations. For example:

(DEFUN REM (X L) ; REMOVE OCCURRENCES OF X FROM L
  ((LAMBDA (CAR-L CDR-L)
    (IF (NLISTP L)
      NIL
      (IF (EQUAL X CAR-L)
        (REM X CDR-L)
        (CONS CAR-L (REM X CDR-L)))))
   (CAR L) (CDR L)))

is transformed via LAMBDA-EXPANSION to:

(DEFUN REM (X L)
  (IF (NLISTP L)
      NIL
      (IF (EQUAL X (CAR L))
        (REM X (CDR L))
        (CONS (CAR L) (REM X (CDR L))))))

where the two underlined expressions in the first program version are each copied twice in the second program version. Edits to either occurrence of (CAR L) will affect the other when edits are traced back to the actual arguments of the LAMBDA expression in the original partition.

REMOVE-ORS-IN-IF-TESTS is not one-to-one since the if-true branch of an IF expression is copied in the rewrite:

(IF (OR disjunct\textsubscript{1} disjunct\textsubscript{2}) if-true if-false)

\[\Rightarrow\]

(IF disjunct\textsubscript{1} if-true (IF disjunct\textsubscript{2} if-true if-false))

8.4.2 Handling Special Cases

All edits are performed by simple edit inversion by default, unless there are special case edit inversion methods defined that will override it. Special case edit inversion methods account for the semantics of transforms when edits are regressed through particular program transformations. When Talus attempts to invert an edit through a transform t to a previous partition, it first calls any edit inversion methods defined for t to see if they can successfully handle the edit. If there are no edit inversion methods that successfully handle the edit, then default edit inversion is used.
When a special case edit inversion method applies it performs the bug edit in the current partition and then propagates the edit to the previous partition. The edit inversion method takes into account the semantics of the transform to determine the edit to perform. Edit propagation stops when the initial partition is reached or no further edit propagation is possible. For example, edit inversion methods for the MAP-FN-TO-PROG transform invert references to PROG accumulator variables (e.g. _AC) back to references to LAMBDA variables. Special case edit inversion methods for the PROG-REMOVAL transform insert missing SETQs into PROGs if variable updates or initializations are absent. The debugging of the following solution to the REVERSE task illustrates just such an edit. The task and solution are:

**REVERSE TASK**
Write a function to reverse a proper list.

```
(DEFUN REV (L)
  (PROG (ANS)
    LOOP
      (IF (NULL L) (RETURN ANS) NIL)
      (SETQ ANS (CONS (CAR L) ANS))
      (GO LOOP))))
```

The PROG-REMOVAL transform changes the code above to:

```
(DEFUN REV (L) (_LOOP L NIL))

(DEFUN _LOOP (L ANS)
  (IF (NULL L)
      ANS
      (_LOOP L ;Bug: L never changes.
               (CONS (CAR L) ANS))))
```

Talus debugs the code in the final version to:

```
(DEFUN REV (L) (_LOOP L NIL))

(DEFUN _LOOP (L ANS)
  (IF (NULL L)
      ANS
      (_LOOP (CDR L) ;Bug Fix: (CDR L) replaces L.
                  (CONS (CAR L) ANS))))
```

The bug edit in the final partition, replacing L by (CDR L), cannot be traced back to the initial partition since that does not contain an explicit update to L. Default edit inversion would apply if the initial program had been:

```
(DEFUN REV (L)
  (PROG (ANS)
    LOOP
      (IF (NULL L) (RETURN ANS) NIL)
      (SETQ ANS (CONS (CAR L) ANS))
      (SETQ L L) ;Explicit update to L.
      (GO LOOP))))
```

Instead a special case edit inversion method intercepts bug edits like the one in this example. It is triggered by an attempt to invert a bug edit through a PROG-REMOVAL transform where the buggy code
replaced is an update to a variable only updated implicitly in the original code. The actions of this method are to insert a SETQ that updates the variable according to the bug fix. Heuristics determine where the SETQ should be inserted. The final debugged code is:

```lisp
(defun rev (l)
  (prog (ans)
    (loop
      (if (null l) (return ans) nil)
      (setq ans (cons (car l) ans))
      (setq l (cdr l)) ; inserted SETQ.
    (go loop))
  ))
```

### 8.4.3 Inserting Missing Conditional Tests

Missing conditionals are detected as described in Chapter 6. The content and insertion points in the final partition are also the same as discussed in Section 7.3. However extensions to allow propagation of the insertion to prior partitions, propagation of future bug edits, and simplifications of conditional expressions inserted into CONDs must be provided for.

The default means of propagating the insertion of conditionals is as follows. The insertion point of the conditional in the current partition is traced back to a node in the prior partition. The if-true and if-false branches are also traced back to the prior partition, or copied if they cannot be. The insertion in the previous partition occurs at the new insertion point and with the new if-true and if-false branches.

As with ordinary bug edits, there are special case methods to handle the insertion of conditionals past partitions created by transforms that are not one-to-one. For example there is a special case method to allow the insertion of conditionals into PROGs.

Conditionals with copied branches that have been inserted into prior partitions have special PROPAGATED-TO pointers that can be traced just as COPIED-FROM pointers are traced during simple edit inversion. This allows bugs in copied branches to be traced back to the original partition.

### 8.4.4 Edit Inversion Failures

Edit inversion can fail to reach the initial partition if necessary edit inversion methods are absent. This can also happen if the student used a mapping function (e.g. MAPCAR) where tree recursion or early termination is necessary. In that case the edits will be inverted back to the PROG that the mapping function was reduced to, but not to the mapping function. Such edits cannot be inverted to the mapping functions since these functions must enumerate every element or sublist in a proper list; early termination or tree recursion is impossible.

Some edits are inverted to the initial partition but are erroneous due to the absence of edit inversion methods. For example no special methods are defined for the MACROEXPANSION transform. Consider the following solution to the REMOVE task:
Write a function which removes all top level occurrences of an atom in a proper list.

(DEFUN REM (X L)
  (IF (NILSTP L)
      NIL
      (IF (EQUAL X (HEAD L))
          (REM X (TAIL L))
          (CONS (TAIL L) (REM X (TAIL L)))))

(DEFUN HEAD (Y) (CAR Y))

(DEFUN TAIL (Y) (CDR Y))

The macro functions HEAD and TAIL are eliminated by a MACROEXPANSION transform, resulting in:

(DEFUN REM (X L)
  (IF (NILSTP L)
      NIL
      (IF (EQUAL X (CAR L))
          (REM X (CDR L))
          (CONS (CDR L) (REM X (CDR L)))))

Talus discovers a bug in the last clause and corrects it in the final version of the program:

(DEFUN REM (X L)
  (IF (NILSTP L)
      NIL
      (IF (EQUAL X (CAR L))
          (REM X (CDR L))
          (CONS (CAR L) (REM X (CDR L)))))

and then uses default edit propagation to invert the edit back to the initial partition:

(DEFUN REM (X L)
  (IF (NILSTP L)
      NIL
      (IF (EQUAL X (HEAD L))
          (REM X (TAIL L))
          (CONS (TAIL L) (REM X (TAIL L)))))

(DEFUN HEAD (Y) (CAR Y))

(DEFUN TAIL (Y) (CAR Y))

The edit is erroneous. What is needed is an edit inversion method for the MACROEXPANSION transform that detects when a "macro" function has been expanded inline more than once and that intercepts edits to the macro body in that case, instead directly altering just the particular macro call that is buggy and replacing it with the propagated edit.

8.5 Summary

The key points of this chapter are:

- Talus can debug solutions that use the extended dialect and that are not in IF-Normal Form.
- Program simplification transforms reduce solutions to the core dialect and IF-Normal Form.
• Talus applies heuristics to compare student and reference code containing side-effectors. These heuristics disallow some correct solutions.

• Simplified programs are debugged and then their bug fixes are inverted to the original programs. Edit inversion is facilitated by a graph representation of program transformations. In this representation, s-expressions that are copied and transformed during program transformations are linked together.

• Special case edit inversion methods take into account the semantics of program transformations. These methods invert edits when the default scheme - following links - fails.
Chapter Nine
Empirical Evaluation

Previous chapters have addressed the debugging methodology of Talus but not its actual performance. This chapter evaluates the performance of Talus in debugging actual student solutions to tasks. Empirical results indicate that Talus performs well in debugging widely varying solutions to nontrivial tasks. The first section in this chapter briefly summarizes performance results in algorithm recognition, bug detection, and bug correction. Later sections discuss the means of data collection and provide more refined measures of performance. Section 9.2 explains the role of student data in the initial development of Talus. Section 9.3.1 explains how data was collected and evaluated to obtain the empirical results of this chapter. Measures of debugging performance in algorithm recognition, bug detection, and bug correction are introduced. Section 9.4 discusses the wide variability of solutions collected. Section 9.5 interprets the results collected and analyzes performance limitations.

9.1 Overview of Performance Results
This section provides a concise overview of the data collected in evaluating the debugging performance of Talus. 106 student solutions were collected\(^{25}\) to five separate tasks. Table 9-1 summarizes the data collected. Performance in algorithm recognition is shown in Table 9-2. Algorithm recognition is considered completely correct when the student’s algorithm was correctly identified, the function mapping was correct, and the formal variable mappings were correct. Algorithm recognition was completely correct more than 90% of the time.

| Solutions | 106 |
| Functions | 176 |
| Formal Variables | 205 |

Table 9-1: Summary of Data Analyzed

| Completely Correct | 97 | 91.5 % |
| Not Completely Correct | 9 | 8.5 % |

Table 9-2: Summary of Algorithm Recognition

Table 9-3 summarizes performance in bug detection. The bugs considered are:

* Extra and missing function definitions.

\(^{25}\)This data is obtained from the second data set and the third data set, before reanalysis. The three data sets collected and the improvements made to Talus are discussed in Section 9.3.1.
- Extra\textsuperscript{26} and missing conditional expressions.
- All bugs in symbolic values. E.g. Wrong function calls.

Bug detection is close to 90\% and false alarms are less than 3\% of the bugs present. One reason that bugs are missed are missing examples - these are examples necessary to invalidate false conjectures that are not initially present in the stored examples. Six missing examples were discovered.

<table>
<thead>
<tr>
<th>Not Counting Extra Conditions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>130</td>
</tr>
<tr>
<td>Detected</td>
<td>120</td>
</tr>
<tr>
<td>Not Detected</td>
<td>10</td>
</tr>
<tr>
<td>False Alarms</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Counting Extra Conditions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>193</td>
</tr>
<tr>
<td>Detected</td>
<td>172</td>
</tr>
<tr>
<td>Not Detected</td>
<td>21</td>
</tr>
<tr>
<td>False Alarms</td>
<td>5</td>
</tr>
<tr>
<td>Missed Examples</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 9-3: Summary of Data Analyzed

Bug correction is summarized in Table 9-4. Two kinds of bug edits are attempted: the insertion of missing conditional expressions and the replacement of buggy s-expressions. Attempted edits succeed a high proportion of the time.

| Attempted Edits            | 120   |
| Succeeded                  | 118   | 98.3 \%|
| Failed/Incorrect           | 2     | 1.7 \% |

Table 9-4: Summary of Bug Correction

9.2 Tuning the Heuristics

Before discussing data collection and more refined measures of performance, the use of student data in the initial development of Talus will be addressed. An analysis of student data motivated many improvements and refinements to an earlier version of Talus. 37 solutions to the SINGLETONS task were collected from a take home assignment in an undergraduate programming languages survey course (CS 345). SINGLETONS was the hardest task in the assignment. Students debugged their solutions on computers before turning them in, consequently there were very few bugs in the solutions. However the

\textsuperscript{26}Initially, redundant conditional expressions were counted as extra conditions and there were no special provisions for their detection. In reanalyzed data, discussed in Section 9.3.1, these two kinds of stylistic bugs are distinguished since improvements to detect redundant conditional tests had been incorporated into Talus. The nondetection of redundant conditional tests and the occurrence of missing examples explain the relatively low performance on bug detection in Table 9-3 when extra conditions are considered.
wide variability in solutions were very useful in tuning the algorithm recognition heuristics. These heuristics were tuned to select the correct algorithm identifications, function mappings, and formal variable mappings in almost every one of the student solutions.

Improvements that reduced the search space required for algorithm recognition were motivated by initial difficulties in correctly identifying the algorithms in this student data. It was necessary to introduce plausibility constraints and to add global solution transforms to reduce the search space during algorithm recognition. Too many function mappings were generated without the plausibility constraints and too many algorithms had to be considered when all possible algorithm variants were stored.

Several E-frame slots were added so that plausibility constraints could be easily checked. The first plausibility constraint requires that the parents of paired functions must themselves be paired. Talus can easily enforce this requirement by checking the FUNCTIONS-CALLING, CONSTRUCTORS-CALLED, and PREDICATES-CALLED slots of any two functions’ E-frames prior to pairing them. The second plausibility constraint requires that paired functions have the same function role and function type. This constraint can be easily enforced by checking FUNCTION-ROLE and FUNCTION-TYPE E-frame slots. With the addition of these five E-frame slots and the enforcement of constraints many implausible function mappings were no longer generated.

Common algorithm variants occurred repeatedly in the student’s data. The following table shows the frequency with which specific solution transforms accounted for these variants among the 37 solutions:

<table>
<thead>
<tr>
<th>Transform</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate-Negation</td>
<td>4</td>
<td>10.8 %</td>
</tr>
<tr>
<td>Tail-Recursion-Introduction</td>
<td>17</td>
<td>45.9 %</td>
</tr>
<tr>
<td>Idempotent-Transform</td>
<td>6</td>
<td>16.2 %</td>
</tr>
</tbody>
</table>

Each transform can occur singly or in combination. There were solutions where two or all three transforms were necessary. Before solution transforms were implemented, common variants of algorithms were stored but it soon became difficult for Talus to distinguish between variants of the same algorithm. When solution transforms were implemented the number of algorithms that had to be stored was significantly reduced. Four algorithms were sufficient to account for 36 solutions, as shown in Table 9-5. The remaining solution used a buggy algorithm. The need for buggy algorithms did not become apparent until further data was collected.

<table>
<thead>
<tr>
<th>Transform</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singletons-W-Member</td>
<td>17</td>
<td>47.2 %</td>
</tr>
<tr>
<td>Singletons-W-Two-Ac-Vars</td>
<td>6</td>
<td>16.7 %</td>
</tr>
<tr>
<td>Singletons-Sliding-Window</td>
<td>12</td>
<td>33.3 %</td>
</tr>
<tr>
<td>Singletons-Collectsame</td>
<td>1</td>
<td>2.8 %</td>
</tr>
<tr>
<td>Total</td>
<td>36</td>
<td>100 %</td>
</tr>
</tbody>
</table>

Table 9-5: Algorithms in Initial SINGLETONS Data

Solution transforms allowed algorithms to be limited to a reasonable number that were sufficiently different that reliable algorithm identification was possible. Since each of the three transforms can apply to any of the four algorithms and in any combination, at least 8 times as many algorithms - 32 algorithms - would be necessary without solution transforms. Furthermore, when algorithm variants were stored it was very difficult to correctly select the right algorithm variant. It is much easier to first identify the
algorithm and then identify any variants.

The SINGLETONS data set also allowed refinement of the heuristics for algorithm recognition. These functions (match, discard, h.g, and vmatch) evaluate the quality of E-frame mappings and formal variable mappings. They are discussed in Chapter 5. Their parameters were manually tuned from the student data in an iterative process. Parameters were adjusted when an algorithm was incorrectly identified, or a function or formal variable mapping failed. When parameters could not be adjusted without invalidating previously correct solution mappings, either new E-frame slots or new measures between existing E-frame slots were added.

9.3 Data Collection and Evaluation

This section discusses the student data that was collected and the measures of debugging performance that were extracted. Three data sets were collected. A total of 337 functions were collected in 143 separate task solutions. Altogether five different tasks (LENGTH, PROG-LENGTH, FLATTEN, COPY-BUT-LAST, and SINGLETONS) were assigned in the data sets. Overall debugging performance on these tasks was measured in addition to performance in algorithm recognition, bug detection, and bug correction. Section 9.3.2 explains these measures. The next section, Section 9.3.1 discusses the three data sets collected; these are the basis for the empirical results of this chapter.

9.3.1 The Data Sets

The first data set consists of the 37 solutions to the SINGLETONS task, that were discussed previously. Talus was tuned to achieve a high level of debugging performance on this data set before the other data sets were considered. Only four algorithms and three solution transforms were required to correctly recognize the algorithms in 36 of the 37 solutions. Five algorithms total were stored - the fifth was never used in any solution. Table 9-6 briefly summarizes the data collected.

<table>
<thead>
<tr>
<th>Task</th>
<th>SINGLETONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0 ;Not yet introduced.</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>5</td>
</tr>
<tr>
<td>Solutions</td>
<td>37</td>
</tr>
<tr>
<td>Functions</td>
<td>161</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>260</td>
</tr>
</tbody>
</table>

Table 9-6: Summary of First Data Set

The second set of data consists of 17 solutions to the COPY-BUT-LAST task. The task description is:

**TASK:** Write a LISP function which copies an exp while replacing the rightmost deepest CONS in the exp with NIL. Atomic sexps are simply returned.

**E.g.** (COPY-BUT-LAST 'X) --> X  
(COPY-BUT-LAST 'A) --> A  
(COPY-BUT-LAST '(A B C)) --> (A B)  
(COPY-BUT-LAST '(A B (C))) --> (A B NIL)  
(COPY-BUT-LAST '(A B (C . D))) --> (A B NIL)  
(COPY-BUT-LAST '(A B (C (E) D))) --> (A B (C (E)))
Here is a correct solution:

```lisp
(DEFUN COPY-BUT-LAST (TR)
  (COND ((ATOM TR) TR)
         ((ATOM (CDR TR))
          (COND ((ATOM (CAR TR)) NIL)
                 (T (CONS (COPY-BUT-LAST (CAR TR))
                     (CDR TR)))))
         (T (CONS (CAR TR) (COPY-BUT-LAST (CDR TR))))))
```

Frequent bugs in solutions to this task are to replace ATOM by NULL in the conditional tests above, or to completely omit one or more conditional tests.

The COPY-BUT-LAST task was handed to students to perform as an in-class exercise in a graduate programming languages survey course (CS 386L). They were told to put their names on the assignments and did not know if they were going to be graded on the assignment (they were not). The students did not have access to a computer so they could not debug their solutions online. Table 9-7 summarizes this data.

<table>
<thead>
<tr>
<th>Task</th>
<th>COPY-BUT-LAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>Solutions</td>
<td>12</td>
</tr>
<tr>
<td>Functions</td>
<td>16</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 9-7: Summary of Second Data Set

The third set of data was a take home assignment, for extra credit, given to students in a graduate level AI class (CS 381K). The instructions specified the LISP dialect and requested that students not use a computer to debug their solutions, although they should attempt to solve the tasks as best they were able to. There are five tasks of varying complexity in the third dataset: LENGTH, PROG-LENGTH, FLATTEN, COPY-BUT-LAST, and SINGLETIONS. The complete handout and all task descriptions are in Appendix V. 20 solutions were collected for LENGTH, FLATTEN, and SINGLETIONS. Table 9-8, Table 9-10, and Table 9-12 summarize the data for these three tasks. Fewer solutions (17) were collected for PROG-LENGTH and COPY-BUT-LAST since some students either did not understand the task or did not know how to use PROG. These tasks are summarized in Table 9-9 and Table 9-11.

<table>
<thead>
<tr>
<th>Task</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>Solutions</td>
<td>20</td>
</tr>
<tr>
<td>Functions</td>
<td>21</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 9-8: Summary of Third Data Set, LENGTH Task

The tasks were intentionally at different degrees of difficulty. LENGTH, PROG-LENGTH, and FLATTEN were intended to be easy and most students did in fact solve them correctly. COPY-BUT-LAST was supposed to moderately difficult and SINGLETIONS was intended to be difficult. As expected there were few solutions that were completely bug-free. These five tasks would be easy for an
<table>
<thead>
<tr>
<th>Task</th>
<th>PROG-LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>Solutions</td>
<td>17</td>
</tr>
<tr>
<td>Functions</td>
<td>17</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 9-9: Summary of Third Data Set, PROG-LENGTH Task

<table>
<thead>
<tr>
<th>Task</th>
<th>FLATTEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>Solutions</td>
<td>20</td>
</tr>
<tr>
<td>Functions</td>
<td>22</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 9-10: Summary of Third Data Set, FLATTEN Task

<table>
<thead>
<tr>
<th>Task</th>
<th>COPY-BUT-LAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>Solutions</td>
<td>17</td>
</tr>
<tr>
<td>Functions</td>
<td>25</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 9-11: Summary of Third Data Set, COPY-BUT-LAST Task

<table>
<thead>
<tr>
<th>Task</th>
<th>SINGLETONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>0</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>6</td>
</tr>
<tr>
<td>Solutions</td>
<td>20</td>
</tr>
<tr>
<td>Functions</td>
<td>75</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>103</td>
</tr>
</tbody>
</table>

Table 9-12: Summary of Third Data Set, SINGLETONS Task

experienced LISP programmer with computer access, but these students had little experience in LISP programming and were asked not to use a computer.

Originally the handout had included more tasks that specifically required the use of side-effectors (e.g., RPLACD). The ability of Talus to analyze solutions with PROG was to be tested in smaller tasks similar to PROG-LENGTH. All of these tasks except PROG-LENGTH were removed so students would not view the assignment as too lengthy and be discouraged from completing it. However, the instructions were not changed so several students used PROG and property list side-effectors in solutions to more difficult tasks, such as SINGLETONS.

In order to analyze these latter solutions to SINGLETONS a sixth algorithm that used property lists was added to the task representation. The PLIST-MARKER-TRANSFORM was also added. The new
algorithm and transform were debugged to correctly identify the algorithms in the two student solutions that used property lists, and to correctly account for the property list indicators they had used. This was viewed as more desirable than discarding these solutions. The original intent was that all solutions to SINGLETONS would be in pure LISP so that direct comparison with the first data set would be possible.

The data for the SINGLETONS task was also reanalyzed after incorporating three improvements into Talus. The first improvement was the addition of buggy algorithms. In the first data set only one student used an incorrect algorithm since solutions could be completely debugged before being turned in. In this third data set however, eight students out of twenty misinterpreted the task and implemented buggy algorithms. The frequent occurrence of buggy algorithms prompted the addition of buggy algorithms to the task representation for the SINGLETONS task. One more correct algorithm was also added when it was discovered that a student had incorrectly implemented an algorithm that was not in the task representation of Talus.

Additional improvements increased the performance of Talus in correctly detecting anomalous conditional tests. The second improvement was to add checks for redundant conditional tests. Previously redundant conditional tests were not detected. The final improvement was to detect when Talus had unnecessarily inserted a conditional test. This can happen when there are logically overlapping predicates or when one conditional nest is nested inside another. If the reference function orders the nested tests differently than the student's function then Talus may erroneously believe that a conditional test needs to be inserted. With the final improvement Talus could usually detect when it had unnecessarily inserted a conditional test and suppress the erroneous bug report.

These two sets of data for SINGLETONS will be distinguished when discussing performance. Data collected after improvements to Talus have been incorporated will be referred to as "reanalyzed". The reanalyzed SINGLETONS data in the third data set is shown in Table 9-13.

<table>
<thead>
<tr>
<th>Task</th>
<th>SINGLETONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buggy-Algorithms</td>
<td>3</td>
</tr>
<tr>
<td>Correct-Algorithms</td>
<td>7</td>
</tr>
<tr>
<td>Solutions</td>
<td>20</td>
</tr>
<tr>
<td>Functions</td>
<td>75</td>
</tr>
<tr>
<td>Formal Variables</td>
<td>103</td>
</tr>
</tbody>
</table>

Table 9-13: Summary of Third Data Set, SINGLETONS Task, After Reanalysis

9.3.2 Measures of Performance

Talus debugging performance can be measured in three areas: algorithm recognition, bug detection, and bug correction. Performance in edit inversion is measured as part of bug correction. Each phase depends on the previous phase and assumes the previous phase is correct. In this way errors in debugging performance are assessed in only one of the three areas. For example, performance in bug detection is evaluated assuming that Talus has correctly identified the student's solution. If not, this is an error in algorithm recognition, not bug detection. In measuring the performance of bug correction only those bugs that have been successfully detected are considered. If Talus does not correct an undetected bug that is not considered an error in bug correction, instead it is an error in bug detection. This division of results into three areas allows greater isolation of the strengths and weaknesses of the current implementation.
9.3.2.1 Algorithm Recognition

One measure of performance is the percentage of solutions in which the algorithm is correctly identified. It can be difficult to decide on what it means to correctly identify an algorithm in some cases. This situation arises when Talus correctly chooses the stored algorithm that most closely matches the student’s algorithm, but the student’s algorithm is not one of those stored. To resolve the ambiguity, this situation is counted as an incorrect algorithm identification.

Initially, Talus only stored correct algorithms. During the analysis of the SINGLETONS task in the third data set it was discovered that many students misunderstood the task and attempted to implement algorithms based on their misunderstandings. Before buggy algorithms were added to the task representation, Talus was forced to choose one of the correct algorithms as matching the student’s algorithm even when the student used a buggy algorithm. The data was reanalyzed after the introduction of buggy algorithms. Performance in both cases will be studied in Section 9.5.1.1. Unless otherwise stated, it should be assumed that all data presented has not been reanalyzed and that better or equivalent performance was measured after reanalysis.

In addition to algorithm identification, performance in function mapping and formal variable mapping is also considered. The number of correct function mappings are measured, assuming that algorithm identification is correct. Then the number of correct formal variable mappings is measured, assuming that both algorithm identification and function mapping are correct.

Bugs that Talus can detect at this stage are

- Buggy Algorithms.
- Inefficient Algorithms.
- Extra Functions.
- Missing Functions.

Bugs that Talus itself can fail to detect are:

- Incorrect Algorithm Identification.
- Wrong Function Mappings.
- Wrong Formal Variable Mappings.
- Inability to Map Formal Variables.

Talus cannot map formal variables between paired functions if they have differing numbers of formal variables that cannot be accounted for by a TAIL-RECURSION-INTRODUCTION transform. This is referred to as a function analysis failure since the resulting function cannot be debugged. These failures are infrequent, as will be discussed in Section 9.5.1.1.

9.3.2.2 Bug Detection

Key measures of performance in bug detection are the number of bugs present that are detected, and the number of false alarms raised. A false alarm is a bug report generated where no actual bug exists. Actual bugs are bugs that are present, whether or not they are detected. Detected bugs are bug reports generated for actual bugs, i.e. false alarms are not counted. Missed bugs are undetected actual bugs.

In all three data sets only the conjecture disprover was used to evaluate conjectures. As mentioned in Section 6.3, invalid conjectures may be believed since there is no stored example to invalidate the conjecture. An example that must be added to the stored examples for correct evaluation of a conjecture
is called a missing example. Missing examples tend not to be a major problem, as discussed in Section 9.5.2. When a missing example was discovered it was added to the set of stored examples for a task before continuing.

Bugs that Talus can detect at this stage are:

- **Extra Conditions.** These are extra or redundant conditional tests present in student functions.
- **Missing Conditions.** Both missing conditions and missing guards are counted in the same category.
- **Wrong Terminations or Recursions.** Bugs in symbolic values.

Bugs that Talus itself can fall prey to are:

- **Missing Examples.**
- **Undetected Bugs or False Alarms in:**
  - **Extra Conditions.**
  - **Missing Conditions.**
  - **Wrong Terminations or Recursions.**

### 9.3.2.3 Bug Correction

The only measure of performance in bug correction is the number of bug edits attempted in the final partition and successfully propagated to the initial partition. It is assumed that algorithm recognition and bug detection are correct. Two kinds of bug edits are possible:

- **The Insertion of a Missing Conditional Expression.**
- **The Replacement of One S-expression in a Symbolic Value by Another.**

Edits can fail in one of the following ways:

- **Failure to Reach Initial Partition.** The edit was not propagated through an intermediate partition due to an unimplemented edit inversion method.
- **Erroneous Edit to Initial Partition.** The edit was propagated by default edit inversion, but the edit in the initial partition is incorrect. It may, for instance, refer to internal variables or auxiliary functions. Again, the failure indicates that an edit inversion method is missing.

### 9.4 Solution Variability

Table 9-14 shows some of the variability present in the initial data set of 37 solutions to the SINGLETONS task. Solutions had from three to six functions, each function having from one to four formal variables. One solution had one extra function and several had one or two missing functions. Global solution transforms were necessary to alter stored solutions to more closely match student solutions. The majority of student solutions required at least one global solution transform, and in one case three were required. Program simplification transforms were used in all solutions to simplify student function definitions for analysis. The most common transform applied was COND-TO-IF. In all solutions at least three program simplification transformations occurred and in one solution seven simplifications were required.

Initially 4 algorithms were sufficient to cover 36 solutions to the SINGLETONS task as was shown earlier in Table 9-5. Allowing side effects required the addition of one more algorithm. Allowing buggy algorithms required the addition of three more algorithms. An additional correct algorithm was also discovered in the solutions to SINGLETONS in the third data set. The following algorithms were used in
Table 9-14: Variability in First Data Set

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Fns in Student Solution:</td>
<td>3</td>
<td>6</td>
<td>161</td>
</tr>
<tr>
<td>Number of Formal Vars per Function:</td>
<td>1</td>
<td>4</td>
<td>260</td>
</tr>
<tr>
<td>Number of Extra Student Functions:</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of Missing Student Functions:</td>
<td>0</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Global Solution Transforms Applied:</td>
<td>0</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>Simplification Transforms Applied:</td>
<td>3</td>
<td>7</td>
<td>155</td>
</tr>
</tbody>
</table>

the 20 solutions to SINGLETONS in the third data set:

<table>
<thead>
<tr>
<th>Correct Algorithms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Singletons-W-Member</td>
<td>5</td>
</tr>
<tr>
<td>Singletons-Sliding-Window</td>
<td>2</td>
</tr>
<tr>
<td>Singletons-W-Single-Store</td>
<td>1</td>
</tr>
<tr>
<td>Singletons-W-Plist-Markings</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buggy Algorithms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unify-Flatten-W-Seen</td>
<td>3</td>
</tr>
<tr>
<td>Unify-Flatten</td>
<td>4</td>
</tr>
<tr>
<td>Unify-Remove</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unknown Algorithms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>2</td>
</tr>
</tbody>
</table>

9.5 Interpretation of Results

This section summarizes and interprets the performance results obtained for algorithm recognition, bug detection, and bug correction. All three data sets are considered. The first area to be studied is performance in algorithm recognition.

9.5.1 Algorithm Recognition

9.5.1.1 Performance

Table 9-15 summarizes performance in algorithm identification over all three tasks. High performance for Data Set 1 is unsurprising since Talus was tuned to achieve this performance. Talus was very successful in identifying algorithms for small tasks such as those assigned in the second data set and the bulk of the third data set.

The lowest scores were achieved in the SINGLETONS task in the third data set before reanalysis. Before an ability to recognize buggy algorithms was introduced Talus could only recognize implementations of correct algorithms. This was always sufficient for tasks of moderate difficulty, such
<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Id</td>
<td>36</td>
<td>12</td>
<td>83</td>
</tr>
<tr>
<td>Missed or Unknown</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>37</td>
<td>12</td>
<td>93</td>
</tr>
<tr>
<td>% Correct Id</td>
<td>97.3 %</td>
<td>100 %</td>
<td>89.2 %</td>
</tr>
</tbody>
</table>

**Table 9-15: Performance in Algorithm Identification**

as COPY-BUT-LAST, or for more difficult tasks such as SINGLETONS when students had computer access. The need for buggy algorithms only became apparent when a moderately difficult task was assigned and online debugging was discouraged. Table 9-16 summarizes the results of algorithm identification in the SINGLETONS task, both before and after reanalysis. Eight students used buggy algorithms and two used unrecognizable variants of either buggy or correct algorithms. When the solutions were reanalyzed with buggy algorithms added, performance in algorithm identification rose from 50 to 90 %. Section 9.5.1.2 analyzes the algorithm variants that Talus could not recognize either before or after reanalysis.

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Id</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Missed or Unknown</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>% Correct Id</td>
<td>50 %</td>
<td>90 %</td>
</tr>
</tbody>
</table>

**Table 9-16: Performance in Algorithm Recognition, SINGLETONS Task, Third Data Set**

Table 9-17 summarizes performance in mapping functions. These figures assume that algorithm identification is correct. Functions were correctly mapped a high percentage of the time in all data sets and all tasks. A function mapping was considered incorrect if:

- A student and reference function were erroneously paired,
- A student function was erroneously mapped to EXTRA, or
- A reference function was erroneously mapped to EXTRA.

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Fn Mappings</td>
<td>161</td>
<td>16</td>
<td>155</td>
</tr>
<tr>
<td>Incorrect</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>161</td>
<td>16</td>
<td>160</td>
</tr>
<tr>
<td>% Correct</td>
<td>100 %</td>
<td>100 %</td>
<td>96.9 %</td>
</tr>
</tbody>
</table>

**Table 9-17: Performance in Mapping Functions**

Table 9-18 summarizes performance in mapping formal variables, assuming algorithm identification and function mappings are correct. When formal variables could be mapped they were always correctly mapped. However three functions in the first data set had too many variables and the extra variables could not be accounted for by a TAIL-RECURSION-INTRODUCTION transform. These functions
could not be analyzed since their formal variables could not be mapped.

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vars Correctly Mapped</td>
<td>250</td>
<td>16</td>
<td>103</td>
</tr>
<tr>
<td>Vars Incorrectly Mapped</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vars Not Mapped</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>260</td>
<td>16</td>
<td>103</td>
</tr>
<tr>
<td>% Correctly Mapped</td>
<td>96.2 %</td>
<td>100 %</td>
<td>100 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fns Analyzed</td>
<td>158</td>
<td>16</td>
<td>160</td>
</tr>
<tr>
<td>Fn Analysis Failures</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>161</td>
<td>16</td>
<td>160</td>
</tr>
<tr>
<td>% Analyzed</td>
<td>98.1 %</td>
<td>100 %</td>
<td>100 %</td>
</tr>
</tbody>
</table>

Table 9-18: Performance in Mapping Formal Variables

In summary, Talus correctly completes all aspects of algorithm recognition a high percentage of the time. After an initial training period where algorithms and missing examples are collected, it seems reasonable to expect Talus to perform algorithm recognition correctly more than 90% of the time for tasks the size of SINGLETONS, if side effects and PROGs are disallowed. Recall that upon final analysis, 10 different algorithms, most having four or five functions were stored in the SINGLETONS task representation. When the initial data set and the SINGLETONS solutions in the third data set were reanalyzed, Talus successfully selected the correct algorithm out of the 10 possible 54 times out of 57 (94.7% correct). Higher performance figures for smaller tasks with fewer algorithms are to be expected. Function analysis failures proved not to be a problem, occurring less than 3% of the time.

9.5.1.2 Analysis of Performance Failures

It is worth examining some of the algorithms that Talus has difficulty identifying. The first solution that will be considered was correctly identified as an implementation of a buggy algorithm, but it illustrates the kind of algorithm variants that can be misidentified. Even if Talus was allowed to debug this implementation using the identified buggy algorithm, instead of the closest matching correct algorithm, it would have coerced the student’s solution into using a different, simpler scheme of recursion than the student intended. Note the unusual recursive calls in SINGLETONWORK below:
(DEFUN SINGLETONS (X)
   (SINGLETONWORK (LIST (CAR (FLAT X))) NIL (CDR (FLAT X))))

(DEFUN SINGLETONWORK (GOOD TEMP ORIG)
   (COND ((EQUAL ORIG NIL)
         (COND ((EQUAL TEMP NIL)
                 (FLAT GOOD))
               (T (SINGLETONWORK (CONS (CAR (FLAT TEMP)) GOOD)
                   NIL
                   (CDR (FLAT TEMP))))))
       (COND ((EQUAL (CAR GOOD) (CAR ORIG))
              (SINGLETONWORK GOOD TEMP (CDR ORIG)))
             (T (SINGLETONWORK GOOD
                 (LIST TEMP (CAR ORIG))
                 (CDR ORIG))))))

(DEFUN FLAT (TREE)
   (COND ((EQUAL TREE NIL) 'NIL)
         ((ATOM TREE) 'Bug: Should be (LIST TREE)
           (T (APPEND (FLAT (CAR TREE))
                 (FLAT (CDR TREE))))))

The algorithm is intended to uniquify the atoms in the flattened input X. It does this by accumulating unique results in GOOD. ORIG is stepped through and each atom is compared to the first atom in GOOD. If it is not equal then it is retained in TEMP otherwise it is discarded. In this way the first atom in GOOD is stripped from the remainder of ORIG. When ORIG is NIL then the first atom in TEMP is added to GOOD, the remainder of TEMP replaces ORIG, and testing continues.

Since this algorithm returns unique occurrences of all atoms, rather than those atoms that occur uniquely, it is a buggy algorithm. The bug in FLAT makes it a buggy implementation of a buggy algorithm. However, Talus correctly identifies this as an instance of the buggy algorithm UNIQUIFY-REMOVE. In the student's solution SINGLETONWORK performs dual roles. First, it steps through a list to accumulate unique atoms. Second, it removes duplicates from the remainder of that list. In the reference functions for the buggy algorithm UNIQUIFY-REMOVE these roles are performed by two separate functions, SINGLETONS1 and REMOVE:
(DEFUN SINGLETONS (TR)
  (SINGLETONS1 (FLAT TR) NIL))

(DEFUN SINGLETONS1 (BAG)
  (COND ((NULL BAG) NIL)
     (T (CONS (CAR BAG)
               (REMOVE (CAR BAG) (CDR BAG))))))

(DEFUN FLAT (TR)
  (COND ((NULL TR) NIL)
     ((ATOM TR) (LIST TR))
     (T (APPEND (FLAT (CAR TR)) (FLAT (CDR TR))))))

(DEFUN REMOVE (X L)
  (COND ((NULL L) NIL)
     ((EQUAL X (CAR L))
      (REMOVE X (CDR L)))
     (T (CONS (CAR L) (REMOVE X (CDR L))))))

Usually, Talus does not recognize unusual variants of algorithms such as the solution with SINGLETONWORK. Another solution transform could account for variations like this one. It would be triggered by recursions that depend on more than one variable where these variables must all be NULL for the function to terminate. For example, in SINGLETONWORK both ORIG and TEMP must be NULL for termination. Talus does not presently have such a solution transform.

One of the two solutions that was incorrectly identified used a PROG:
(DEFUN SINGLETON (L)
  (PROG (RHS LHS COMP S FIND)
    (SETQ RHS (FLATTEN L))
    (SETQ COMP (CAR RHS))
    (SETQ LHS NIL))
  
START
  (SETQ LHS (CONS COMP LHS))
  (SETQ COMP (CAR RHS))
  (COND ((NULL COMP) (RETURN FIND)))
  (SETQ RHS (CDR RHS))
  (SETQ S (APPEND LHS RHS))

LOOP
  (COND ((EQUAL COMP (CAR S)) (GO START)))
  (SETQ S (CDR S))
  (COND ((NULL S)
    (SETQ FIND (CONS COMP FIND))
      (GO START))
    (T (GO LOOP)))))

(DEFUN FLATTEN (L)
  (COND ((NULL L) L)
    ((ATOM L) (LIST L))
    ((ATOM (CAR L))
      (CONS (CAR L) (FLATTEN (CDR L))))
    (T (APPEND (FLATTEN (CAR L))
         (FLATTEN (CDR L)))))))

The PROG-REMOVAL transform expanded SINGLETON into three functions. The two functions derived from the PROG blocks START and LOOP were mutually recursive and each had five formal variables. There are no algorithms for SINGLETONS with mutually recursive functions, and none have more than three formal variables, so Talus could not correctly identify this algorithm.

The previous solution suggests that Talus can more easily be led astray with solutions containing PROG since there can be:

* Many unnecessary local variables. Since variable roles characterize algorithms, the extraneous variables will degrade performance in algorithm recognition.

* Multiple program blocks and complicated flow of control. All algorithms in Talus assume simple recursion strategies, not complicated strategies like that of SINGLETONWORK or the mutually recursive functions that result from SINGLETON.

The second solution that was incorrectly identified did not combine two different functions, instead it combined algorithms in an unusual way:
(DEFUN SINGLETONS (X)
 (SO (FLATTEN X)))

(DEFUN S0 (X)
 (COND ((NULL X) NIL)
 ((NOT (MEMBER (CAR X) (CDR X)))
  (CONS (CAR X) (S0 (CDR X))))
 (T (S1 (CDR X) (CAR X))))

(DEFUN S1 (X Y)
 (COND ((AND (ATOM X)
  (NOT (MEMBER (CAR X) Y))) X)

 ((AND
  (NOT (MEMBER (CAR X) Y))
  (NOT (MEMBER (CAR X) (CDR X))))
  (CONS (CAR X) (S1 (CDR X) (CONS (CAR X) Y))))
 (T (S1 (CDR X) Y))))

(DEFUN MEMBER (X Y)
 (COND ((NULL Y) NIL)
 ((EQUAL X Y) T)
 (T (MEMBER X (CDR Y))))

(DEFUN FLATTEN (L)
 (COND ((ATOM L) (CONS L NIL))
 ((ATOM (CAR L)) (CONS (CAR L) (FLATTEN (CDR L))))
 (T (APPEND (FLATTEN (CAR L)) (FLATTEN (CDR L))))))

This solution appears to combine two correct algorithms that are part of the SINGLETONS task representation. The first algorithm, SINGLETONS-W-MEMBER, checks the CDR of a list to see if the CAR is present. If it is then it is removed from the CDR and not collected. If it is not present then it is collected. This algorithm is partially implemented by the function S0 above, but there is no function REMOVE to strip out duplicate atoms. The second algorithm, SINGLETONS-W-SLIDING-WINDOW, keeps track of the nonunique atoms that have been seen so far and checks the CAR of the list left to see if it is in this list or the remaining CDR. That algorithm is partially implemented by the function S1, although the termination test is incorrect. Essentially this solution follows the first algorithm until a nonunique atom is encountered then switches to the next algorithm.

Program simplification transforms are intended to facilitate algorithm recognition, but in some cases these transforms can introduce errors. Functions were not mapped correctly when program simplification transforms removed them from the student’s solution. For example one student solution had the following definition of FLATTEN:

(DEFUN FLATTEN (S)
 (COND ((NULL S) (LIST NIL))
  ((ATOM S) (LIST S))
  (T (APPEND (CAR S) (CDR S))))

This function is not recursive and was removed from the simplified program version by a MACROEXPANSION-TRANSFORM. Consequently the function FLATTEN was not mapped to its reference function counterpart and was never analyzed. Instead it was considered missing from the student’s solution.
Functions can also fail to be analyzed if their formal variables cannot be mapped. Formal variable mappings fail when the number of formal variables differ between paired student and stored functions, and the difference cannot be accounted for a TAIL-RECURSION-INTRODUCTION transform. An example of a student function whose formal variables cannot be mapped is shown below:

```
(DEFUN SINGLETONS2 (CARX CDRX COLLECT1 COLLECT2)
  (COND ((NULL CDRX))
    ((NOTINSET CARX COLLECT2) (CONS CARX COLLECT1))
    (T COLLECT1)))
((AND (NOTINSET CARX COLLECT2) (NOTINSET CARX CDRX))
 (SINGLETONS2 (CAR CDRX) (CDR CDRX))
 (CONS CARX COLLECT1) COLLECT2))
(T (SINGLETONS2 (CAR CDRX) (CDR CDRX) COLLECT1
 (CONS CARX COLLECT2)))))
```

A solution transform that could account for destructured variables such as CARX and CDRX could accommodate such a solution, but this has not been implemented.

### 9.5.2 Bug Detection

It is worth reemphasizing that all data presented in this section assumes that the algorithm recognition and solution mapping is completely correct. Bugs missed due to incorrect algorithm identification or solution mapping are counted in the algorithm recognition statistics, not here.

At the algorithm level the only bugs Talus detects are buggy and inefficient algorithms. No statistics were kept on inefficient algorithms but Table 9-19 summarizes detection of buggy algorithms after data reanalysis. Only data for the SINGLETONS task is considered since there are no buggy algorithms for any other tasks. There was only one solution in the first data set that used a buggy algorithm. When Data Set 1 was reanalyzed it was correctly identified but another solution was erroneously identified as using a buggy algorithm. When Data Set 3 was reanalyzed all 8 solutions based on buggy algorithms were correctly identified and there were no false alarms.

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Id</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Missed</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>False Alarms</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Solutions</td>
<td>37</td>
<td>20</td>
<td>57</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Table 9-19</strong>: Performance in Detecting Buggy Algorithms in Reanalyzed Data</td>
<td></td>
</tr>
</tbody>
</table>

At the function level the only bugs to consider are missing and extra function definitions. There was only one extra function out of 337 in all the data before reanalysis, and three after reanalysis\(^{27}\). All were detected and there were no false alarms. There were 22 missing functions before reanalysis and 23 missing functions after reanalysis. Talus correctly detected them all. One false alarm occurred: Talus reported one function missing that was not really missing. The false alarm was due to a solution where

\(^{27}\) Function mappings can change after reanalysis since the introduction of additional algorithms will result in new function mappings to these algorithms.
the recursive calls in a function that should be recursive were omitted:

\[
\text{(DEFUN FLATTEN (S)} \\
\quad \text{(COND \ ((NULL S) (LIST NIL))} \\
\quad \quad \text{((ATOM S) (LIST S))} \\
\quad \quad \text{(T (APPEND (CAR S) (CDR S))))})
\]

As mentioned previously, the MACROEXPANSION transform expanded calls to FLATTEN inline and removed FLATTEN during program simplification. The reference function FLATTEN could not map to the student's definition in the simplified solution so FLATTEN was considered missing.

Moving to the implementation level, the first measure of bug performance that will be considered is detection of anomalous conditional tests. Table 9-20 summarizes performance in detecting extra and redundant conditional tests in the SINGLETONS task. There were 89 extra or redundant conditional tests total when all three data sets are combined. Of these 89, 72 were correctly detected (81%). Upon reanalysis, when provisions for detecting redundant conditional tests were improved, 81 were correctly detected (91%). On the other hand, there were 8 false alarms before reanalysis and 13 afterwards.

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly Detected</td>
<td>20 (76.9 %)</td>
<td>3 (42.9 %)</td>
<td>49 (87.5 %)</td>
</tr>
<tr>
<td>Not Detected</td>
<td>6</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>False Alarms</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Actually Present</td>
<td>26</td>
<td>7</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 9-20: Performance in Detecting Extra Conditional Tests

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly Detected</td>
<td>11 (100 %)</td>
<td>17 (94.4 %)</td>
<td>43 (95.6 %)</td>
</tr>
<tr>
<td>Not Detected</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>False Alarms</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Actually Present</td>
<td>11</td>
<td>18</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 9-21: Performance in Detecting Missing Conditional Tests

Table 9-21 summarizes performance results in detecting missing conditional tests. There were 74 missing conditional tests total in all the data. Of these, 71 were identified (96%) and there were eight false alarms. Checks for the unnecessary insertion of conditional tests were added to Talus to reduce false alarms. Consequently, all false alarms were eliminated in reanalyzed data. There were 72 missing conditional tests in the reanalyzed data of which 69 were identified (95.8%). Missing conditional tests that went undetected were missed because of missing examples.

In addition to anomalous conditional tests, bugs at the implementation level include bugs in function terminations or recursions. Table 9-22 summarizes performance in detecting these bugs. There were 72 actual bugs in terminations/recursions in all the data. Of these 70 (97%) were detected using the original

---

\[^{28}\text{See Section 7.3.1.3 for further details.}\]
Table 9-22: Performance in Detecting Bugs in Terminations and Recursions

examples. Undetected bugs were due to missing examples entirely. All bugs were detected when missing examples were added. There were no false alarms. Reanalysis did not change these figures.

Missing examples are the major cause of undetected bugs at the implementation level. There were six missing examples. Missing examples can cause anomalous conditional tests or bugs in symbolic values to be undetected. Once an example was determined to be missing it was added to the task representation, but only after analyzing and recording the statistics for a solution where the missing example was first noticed.

Most false alarms in detecting anomalous conditional tests are due to differing orders of embedded conditional tests in paired student and reference functions. The application of simplification transforms may alter a branch under an unnecessarily inserted conditional; this alteration will prevent Talus from detecting the unnecessary insertion.

Another cause of false alarms for missing conditional tests are student function definitions that omit guards for function terminations that can never be reached given implicit data restrictions. For example a correct definition of a function to remove occurrences of an atom X from a list L known to contain exactly one occurrence of X is:

```lisp
(DEFUN REM1 (X L)
  (IF (EQUAL X (CAR L))
      (CDR L)
      (CONS (CAR L) (REM1 X (CDR L))))
```

However, the reference function will be defined as:

```lisp
(DEFUN REMOVE (ITEM BAG)
  (IF (NULL BAG)
      NIL
      (IF (EQUAL ITEM (CAR BAG))
          (REMOVE ITEM (CDR BAG))
          (CONS (CAR ITEM)
                (REMOVE ITEM (CDR BAG))))))
```

The conditional test (NULL BAG) and function termination NIL are included in the reference function so that it will be stylistically correct and admissible to the theorem prover. As a result Talus will claim that REM1 has a missing conditional test (NULL L). This was considered a false alarm since REM1 correctly performs its role in solving the SINGLTONS task, but it can be argued that the NULL test should be there for stylistic reasons.
9.5.3 Bug Correction

<table>
<thead>
<tr>
<th></th>
<th>Data Set 1</th>
<th>Data Set 2</th>
<th>Data Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful Edits</td>
<td>21 (100%)</td>
<td>25 (100%)</td>
<td>93 (97.9%)</td>
</tr>
<tr>
<td>Failed Edits</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Total Attempted</td>
<td>21</td>
<td>25</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 9-23: Performance in Bug Correction

The summary of bug correction in Table 9-23 assumes correct algorithm recognition and correct bug detection. Bug edits are attempted both to insert missing conditional tests and to replace buggy s-expressions in symbolic values with correct s-expressions. Talus attempted 141 edits before reanalysis and succeeded in all but two cases (98.6%). After reanalysis, Talus succeeded in 135 attempts or 95.7% of its edit attempts. Most edit failures occur when edits are erroneously inverted through PROG-REMOVAL transforms.

9.6 Summary

The key points of this chapter are:

- Three sets of data were analyzed to measure performance in algorithm recognition, bug detection, and bug correction.
- The final version of Talus achieves high performance in all three areas.
- For one task, SINGLETONS, algorithm recognition was significantly improved when buggy algorithms were added.
- Limitations of the algorithm recognition heuristics and the conjecture evaluator were not serious:
  - Missing examples were infrequent.
  - Failed formal variable mappings occurred rarely.
Chapter Ten
Conclusion

This chapter summarizes this research and its contributions, then considers future directions for extending this work. Section 10.1.1 reviews the various approaches to automated debugging and the trade-offs incurred in selecting one approach over another. Although each approach has its own particular assets and liabilities all approaches have one common limitation. Their ability to cope with solution variability is dependent on their ability to reason about computational semantics. When reasoning capabilities are restricted, potential solution variability and debugging capabilities must also be reduced.

One particular approach to automated debugging that incorporates an ability to explicitly reason about computational semantics has been presented. This is the debugger Talus. Talus debugs recursive programs by repairing induction proofs. Section 10.1.2 reviews this approach. Two particular advantages of this approach are an ability to distinguish unexpected implementation variants from buggy code and an ability to correct buggy code with code from exemplar solutions. Section 10.2 considers several unimplemented extensions to Talus. Integration of Talus into a complete intelligent tutoring system is explored along with extensions to handle other programming languages.

10.1 Summary
This section summarizes this research. Section 10.1.1 reviews alternate approaches to program debugging. The importance of reasoning about computational semantics to allow increased solution variability is reviewed. Section 10.1.2 reviews the debugging approach of Talus. Particular emphasis is given to the direct incorporation of an explicit capability to reason about computational semantics during the debugging process. The heuristic extensions that allow this approach to be practically applied to debugging recursive programs are also reviewed.

10.1.1 Approaches to Automated Program Debugging
There are many alternate strategies to choose from when designing an automated debugger for an intelligent tutoring system. Knowledge of the relative advantages of one approach over another provides a framework for choosing a strategy along with realistic expectations of what can be expected. Figure 10-1, first presented in Chapter 2, provides a taxonomic classification of the approaches available. The major split is between approaches that rely on program execution on specific examples (dynamic analysis) and those that do not (static analysis).

Dynamic analysis allows the rapid localization of bugs with minimal effort in program analysis. Unfortunately, bugs can be missed if the wrong examples are tried or if the program trace is incomplete. Some stylistic bugs can be especially difficult or impossible to detect. Extending this approach to handle side effects requires incorporation of program state information into execution traces and program tests.

Static analysis requires more thorough program analysis but allows advantages that dynamic analysis cannot offer. Stylistic bugs are more amenable to detection. Unexpected program implementations can be proven correct. Misconceptions can be more readily identified from program context and by understanding code teleology - the purpose of code statements. No special provisions for
side effects are necessary. Buggy code can be repaired and code can by synthesized to extend partial solutions. These advantages come with a price. More detailed program analysis is required; errors in analysis can prevent or degrade program debugging.

There are several variations to the static analysis approach. Analysis by synthesis offers the advantage that partial solutions can be extended or bugs can be fixed by program synthesis. But program synthesis is difficult in most domains. Heuristic plan recognition or plan parsing can be applied when program synthesis is unavailable. Unfortunately, unexpected implementations can be unanalyzed or erroneously interpreted as bugs. All three approaches depend on correctly anticipating acceptable approaches to tasks and likely bugs that will arise.

Program verification approaches attempt to prove that student programs satisfy task specifications. When the proof fails the student’s program is altered to restore the validity of the proof. With this approach, unanticipated implementations can be proven correct. However, imperative programs are

---

29 Although special provisions for side effects are necessary for program verification approaches such as Talus, they are not necessary for approaches based on heuristic plan recognition (e.g. PROUST [Johnson 85]) or analysis by synthesis (e.g. GREATERP [Reiser 85]).
difficult to debug if tasks are specified with loop invariants. Synthesizing loop invariants and altering program statements to derive necessary invariants is difficult. Talus does not rely on loop invariants but does require the presence of model solutions. Its approach incorporates direct reasoning about computational semantics with bug correction from model solutions to provide a practical means of debugging recursive programs.

Each approach has its own advantages and disadvantages; there is no single "best" approach. However all approaches must address a common problem: there must be some means of coping with the potential for enormous variability in student solutions. Variability arises in choice of algorithm, procedural decomposition, role of variables, control flow, data flow, identifiers, and particular programming constructs chosen. All systems impose limitations to manage variability.

This dissertation draws a relationship between allowed variability and reasoning capabilities in automated debuggers. Their potential to correctly debug widely varying novice programs is proportional to their capabilities to reason about computational semantics. Debugging capabilities and solution variability can be expanded by increasing capabilities to reason about computational semantics. If, on the other hand, these reasoning capabilities are limited then corresponding limitations must be imposed on solution variability and debugging capabilities.

Chapter 2 supported this contention by pointing out limitations in existing debuggers due to restricted reasoning capabilities. Various debugging approaches were considered. In each case restrictions in debugging capabilities and allowed solution variability resulted from limited program reasoning capabilities. Chapters 3 to 8 presented an approach to program debugging that explicitly reasons about computational semantics, resulting in additional debugging capabilities. Facilities for algorithm recognition, bug detection, and bug correction that are not typically present in other debugging approaches result from this increased ability to reason about programs. This demonstration of increased debugging power that can be attributed to greater program reasoning capacity further supports this dissertation's thesis.

10.1.2 Reasoning about Computational Semantics in Talus

This section reviews the debugging approach of Talus, with particular emphasis on its capabilities to reason about programs and programming language semantics. In this debugging approach, recursive programs are compared to model programs. These exemplar solutions act both as specifications and sources of correction. An inductive proof of equivalence is constructed to compare student and reference programs. Where the proof would fail the student program is altered with code from reference programs.

To extend this approach to multiple functions, heuristics are necessary to pair reference functions with student functions. Formal variables must also be paired heuristically before verification conditions can be generated. These verification conditions are based on the Principle of Induction in the Boyer-Moore Logic and can be evaluated with the Boyer-Moore Theorem Prover. Additional heuristic means of conjecture evaluation allow this approach to be practically applied to larger programs than would otherwise be possible.

Programs are simplified prior to algorithm recognition and the generation of verification conditions. Equivalence preserving transformations reduce programs to a standard form and a simpler dialect. Heuristic procedures are used to trace edits from simplified programs back to unsimplified programs.

Intertwined in this debugging approach are three kinds of program reasoning:
• **Heuristic Reasoning.** The most plausible algorithm identification, function mappings, and formal variable mappings are all determined heuristically.

• **Conjecture-based Reasoning.** Symbolic evaluation and bug detection depend on the evaluation of conjectures. Conjectures express logical implication, functional equivalence, and termination of program fragments.

• **Procedural Reasoning.** Knowledge of LISP semantics is encoded in Talus procedures to parse programs into frames, and to invert program edits.

The four steps in Talus debugging (program simplification, algorithm recognition, bug detection, and bug correction) will be reviewed, emphasizing the role of these kinds of reasoning.

### 10.1.2.1 Program Simplification

Programs are simplified to IF-Normal Form and the core dialect through the application of equivalence preserving transformations. Program simplification facilitates algorithm recognition and bug detection by reducing the complexity and superficial variability of programs.

Procedural reasoning supports program simplification. Procedures determine when transforms are applicable and their results. For instance, procedures determine if function definitions return fresh list structure. This analysis allows the replacement of destructive functions (e.g., NCONC) by nondestructive equivalents (e.g., APPEND) when arguments are fresh list structure. Another procedure reexpresses an imperative program (i.e., containing PROG) as an equivalent set of recursive functions.

### 10.1.2.2 Algorithm Recognition

Algorithms are identified heuristically. The algorithm recognition process of Talus allows increased solution variability through free choice of algorithm, functional decomposition, and role of formal variables. An implementation of the A* algorithm simultaneously determines the best mapping of reference functions to student functions and the best algorithm identification. Formal variables are also mapped heuristically. The heuristic functions \( g, h, \text{match}, \text{discard}, \) and \( \text{vmatch} \) are a vital part of the algorithm recognition process; these were discussed in detail in Chapter 5.

Heuristics must be used when determining the "best" algorithm identification, function mapping, or formal variable mapping. Formal definitions of "best" and "most similar" are impractical. Robust identification must be supported while allowing for program bugs to introduce noise in the process. Partial matching is forced.

### 10.1.2.3 Bug Detection

Bugs are detected by generating and evaluating verification conditions. These verification conditions are required for the inductive proofs that establish the functional equivalence of each student function to its paired reference function. Evaluation of these conjectures can be implemented by a mechanical theorem prover such as the Boyer-Moore Theorem Prover, or through heuristic means such as the conjecture disprover.

Conjecture-based reasoning allows explicit reasoning about logical implications, functional equivalence, and program termination. Logical implications arise in symbolic evaluation, which is used both to detect missing conditional tests and to decompose program debugging into disjoint cases. Functional equivalence verification conditions allow unexpected but correct implementations of programs. Consequently, functional equivalence verification conditions replace the plan-difference rules that are present in other debugging systems (e.g., PROUST [Johnson 85]) to account for implementation
variants in student programs. Talus relies on the second kind of verification condition, termination verification conditions, to detect potential program nontermination.

10.1.2.4 Bug Correction

Bugs are corrected by replacing buggy student code with normalized reference code. Normalized code replaces stored identifiers by student identifiers. This replacement can be justified by the instantiation of inductive hypotheses when proving the equivalence of paired student and reference functions.

Procedural reasoning is required to trace edits from simplified programs back to the original student code. A graph representation of program transformations facilitates this edit inversion process. S-expressions copied or transformed during program transformations are linked together. A default procedure for tracing bug edits along these links is augmented with specific edit inversion methods associated with particular program transforms. These special case methods procedurally encode knowledge about the semantics of program transformations.

10.2 Future Directions

This dissertation concludes by briefly considering future directions for extending this work. Integration of Talus into a complete intelligent tutoring system to teach LISP is already planned. It is possible that additional information obtained from other components of an ITS can further improve the performance of Talus. For example, information about student capabilities derived from the student model could bias the algorithm recognition process towards certain identifications. In informal experiments it was observed that proficient LISP programmers considered algorithms that beginners would not. Natural language capabilities could provide additional information about the intended role of functions or formal variables. In turn, Talus could be further developed to search for additional stylistic errors. Poor use of identifiers and additional violations of implicit programming conventions could be detected. Specific bug rules to allow (even) greater specificity in bug isolation, correction, and hint generation could be incorporated.

Program debuggers for other programming languages than LISP can be supported with Talus. Since Talus is specifically designed for debugging recursive programs, programming languages such as LOGO seem especially amenable to this approach. Extensions to debug LOGO programs should be straightforward due to the strong similarity of LOGO and LISP. Imperative programming languages appear less well suited for the debugging approach of Talus. Programs in procedural programming languages would need to be converted to recursive equivalents, debugged, and then the edits inverted to the original programs. But the equivalent recursive formulations may be far removed from the procedural style of the original programs and it may be difficult to invert edits. At this time it is unclear how applicable Talus is to procedural programming languages.

In addition to functional programming languages, Talus can be applied to debugging deterministic PROLOG programs. Deterministic PROLOG programs can be mechanically translated to equivalent LISP programs, debugged, and then the edits inverted back to the original PROLOG programs. For example consider the PROLOG task and program below:
MEMTREE Task

Write a predicate that takes two arguments, an atom and a list, and that succeeds only when the atom is present in the list, at any level of embedding.

E.g. \textit{memtree}(\texttt{x, [8, x, 9]}) succeeds.
\textit{memtree}(\texttt{x, [8, [j, k, x], 9]}) succeeds.
\textit{memtree}(\texttt{x, [8, [j, k, y], 9]}) fails.

% The student's solution is shown below.

\texttt{memtr(X,Y) :- flat([],Y,Z), in(X,Z).}

\texttt{flat(I, [X|Y], Z) :- flat(I,Y,V), flat(V,X,Z).}
\texttt{flat(I, X, I).}

\texttt{in(X, []).}
\texttt{in(X, [X|Y]).}
\texttt{in(X, [Y|Z]) :- in(X, Y).}

Talus would represent this program internally as:

\begin{verbatim}
( (MEMTR X Y) <- (FLAT [] Y Z) (IN X Z) )

( (FLAT I [X|Y] Z) <- (FLAT I Y V) (FLAT V X Z) )
( (FLAT I X I) )

( (IN X [X]) )
( (IN X [X|Y]) )
( (IN X [Y|Z]) <- (IN X Y) )
\end{verbatim}

A program simplification transform would convert this program to LISP code similar to that in the MEMTREE case study:

\begin{verbatim}
(DEFUN MEMTR (X Y)
 (IN X (FLAT NIL Y)))

(DEFUN FLAT (I L)
 (IF (LISTP L)
      (FLAT (FLAT I (CDR L))
           (CAR L))
      I))

(DEFUN IN (X L)
 (IF (LISTP L)
      (IF (EQUAL L (LIST X))
       T
       (IF (EQUAL (CAR L) X)
        T
        (IN X (CAR L))))
      NIL))
\end{verbatim}

After algorithm recognition, bug detection, and bug correction the debugged LISP code would be:
(DEFUN MEMTR (X Y)  
  (IN X (FLAT NIL Y)))

(DEFUN FLAT (I L)  
  (IF (LISTP L)  
      (FLAT (FLAT I (CDR L))  
            (CAR L))  
      (CONS L I)))

(DEFUN IN (X L)  
  (IF (LISTP L)  
      (IF (EQUAL L (LIST X))  
          T  
          (IF (EQUAL (CAR L) X)  
              T  
              (IN X (CDR L))))  
      NIL))

Special edit inversion methods for PROLOG would determine the corresponding corrections in the original code:

memtr(X,Y) :- flat([],Y,Z),in(X,Z).

flat(I,[X|Y],Z) :- flat(I,Y,V),flat(V,X,Z).
flat(I,X,[X|I]).
in(X,[X]).
in(X,[X|Y]).
in(X,[Y|Z]) :- in(X,Z).

A combination of debugging approaches may be best for the fully automated debugging of nondeterministic PROLOG programs. Shapiro’s bug detection routines for PROLOG [Shapiro 83] could be combined with the algorithm recognition, bug isolation, and bug correction routines of Talus. The algorithm recognition and program frame representations of Talus would have to be modified for PROLOG. Student programs would be parsed into frame representations. Algorithm recognition would pair student predicates and formal variables with reference predicates and formal variables. Stored examples would test the student’s predicates. Oracle queries about program traces would be automatically answered by determining the correct trace from the execution of reference predicates. Oracle queries about program divergence (i.e. nontermination) can be answered by applying a measure and well-founded relation, obtained from the task representation, to the arguments of successive procedure calls. Shapiro’s routines only isolate bugs to either uncovered goals or incorrect predicate clauses. Uncovered goals could be satisfied by adding normalized reference clauses that cover the goals. Bugs in incorrect predicate clauses could be isolated by comparison with the corresponding correct predicate clauses from the normalized reference program. Talus would augment Shapiro’s routines by adding

• Automatic algorithm recognition.
• Bug isolation within clauses.
• Uniform bug correction that always succeeds.
• Limited handling of side effects.

No user query would be required. In return, Shapiro’s bug detection routines handle nondeterminism and
finite failure, two programming issues that arise in PROLOG but not in LISP.
I. Talus in Classical Mythology

There are two myths about Talus. In the first myth Talus is a bronze robot that guards Crete. Here is the myth as described in [Oswalt 69]:

Talus a) Talus, like the Melic nymphs (Meliades), was sprung from the ash trees in the early ages of the world. Others say that he was one of the brazen race, or that he was made by Hephaestus. Talus was all of bronze, and had a single vein running from his head to his feet, which was closed up by a bronze nail, or was covered by skin. He was invulnerable all over his body, except for the point where the vein ended. Talus had been given to Europa, or to Minos by Zeus, to be a guardian of Crete. He ran round the island three times a day, and pelted any ship that approached with rocks. When the Argonauts wished to land in Crete, they too were threatened with rocks; but Medea sang incantations, so that Talus fell and grazed the spot where his single vein ended, and died. Others say that Calais and Zetes, or the Dioscuri killed him.

In [Tripp 70] this myth is also described:

Talus. A brazen giant who guarded Crete. Talus was said by some writers to have been the last survivor of the bronze race (see RACES OF MAN). Talus was given by Zeus to Europa to guard her island of Crete. According to others, Talus was one of the ingenious constructions of the divine artisan Hephaestus, who gave the robot to King Minos. Still others claimed that Talus was only a bronze bull. In the two stories that survive about him, Talus was a giant who marched around Crete three times a day and kept off intruders by pelting their ships with huge rocks. His bronze body was kept alive by an ichor contained in a single vein. This vein, closed at its end in one ankle with either a bronze nail or a thin layer of skin, was the giant’s one vulnerable spot.

The ARGONAUTS [S] encountered Talus on their way home from Libya. Medea overcame him by one of three means: she maddened him with drugs; or, having his confidence with a promise of immortality, she drew out the nail in his ankle; or she worked on him with the evil eye and the power of suggestion until he accidentally grazed his ankle against a rock and himself opened his vein. According to an entirely different version of the story, Talus was killed by the Argonaut Poes, who shot him in the ankle with an arrow. [Apollonius Rhodius 4.1639-1693; Apollodorus 1.9.26]

The other myth describing Talus as an apprentice of Daedalus, is taken from [Oswalt 69] under the entry of Daedalus:

Daedalus 1. Daedalus the architect, of the family of Erechtheus of Athens, was the son of Metion or of Eupalamus and Alcippie. He was the most talented artist of his day, and made statues so life-like, that the onlooker might think that the statue was the person portrayed. His sister gave him her son Talus, who is also called Calus or Perdix, to be his apprentice, and Talus soon outdid his master; he invented the saw by copying the jawbone of a snake with its teeth, or the backbone of a fish. He also invented compasses for drawing circles, and the potter’s wheel. Thus Daedalus became jealous of his pupil and pushed him from a roof-top or from the Acropolis, and then he buried the body, and while doing this was discovered. Or else, before Talus touched the ground, Athena, who loves clever craftsmen, transformed him into a partridge (perdix). In any case, Daedalus was tried for this murder before the Areopagus, was found guilty and exiled. 2. He went to Crete where Minos was king. When Pasiphae, the wife of Minos, conceived her mad passion for the bull of Poseidon, Daedalus became her accomplice by making a cow and setting Pasiphae inside it. When the bull saw the cow he mated with it. Thus Pasiphae brought into the world the terrible Minotaur, who had a human body with a bull’s head. Then Daedalus designed the Labyrinth, which was constructed in such a way that once in it, one could not find the exit again. The Minotaur was kept there and devoured the Athenian maidens and youths who were sent as tributes (see Androgueus). 3. When Theseus of Athens came to slay the Minotaur, and Ariadne, daughter of Minos and Pasiphae, fell in love with him, Daedalus gave her the ball of thread, by means of which it was possible to find the way back to the entrance of the labyrinth. So Theseus accomplished his task and fled with Ariadne. Learning this, Minos imprisoned Daedalus in the labyrinth with Icarus, the son whom a slave girl of Minos had borne to him. But Daedalus constructed wings out of wax and feathers for himself and Icarus, and they flew away from Crete. Icarus, however, contrary to the warnings of Daedalus, flew too high, and the sun melted the wax of the wings, so that Icarus fell into the sea and drowned. The sea was called Icarian after him. Daedalus buried him on the island Icaria, named after Icarus, and as he did so, the partridge that had been Talus maliciously looked on.
II. An Annotated PDS6 Scenario

This appendix presents a trace of PDS6 debugging a QUICKSORT program. This trace largely duplicates the scenario in the book Algorithmic Program Debugging [Shapiro 83] on page 147. Explanatory comments, in italics, have been added to specifically emphasize the role of user query and the dependence of PDS6 on it. The user’s replies to queries are underlined. This scenario illustrates:

- The power of the system to isolate bugs since it can reason about program execution traces.
- The reliance on the user to provide information about code teleology, and to make decisions that require this knowledge.
- Debugging limitations due to restrictions in the ability to reason about computational semantics.

Here is the quicksort program to be debugged:

```prolog
qsort([X|L],L0) :-
partition(L,X,L1,L2),
qsort(L1,L3),qsort(L2,L4),
app(L3,[X|L4],L0).

partition([X|L],Y,L1,[X|L2]) :- partition(L,Y,L1,L2),X<Y.
partition([X|L],Y,[X|L1],L2) :- partition(L,Y,L1,L2),X =< Y.
partition([],X,[]).

app([X|L1],L2,L3) :- app([X|L1],L2,[X|L3]).
app([],L1,L).
```

The user informs PDS6 about properties of QSORT:

```prolog
:- declare(qsort(+[x],-[x]),[determinate,total]).
:- declare_called(qsort(X,Y),
                  [qsort(_,_),partition(_,_,_,_),app(_,_,_)]).
```

This declaration means that the first argument to QSORT is always a list and considered to be program input and the second argument is also a list and is considered to be program output. QSORT is determinate, i.e. it never succeeds more than once on a goal. QSORT is total, i.e. it always succeeds at least once on a goal. QSORT only calls itself, PARTITION, and APP. This information considerably limits the search space for bug corrections. Similar declarations for PARTITION and APP are also provided:

```prolog
:- declare(partition(+[x],+x,-[x],-[x]),[determinate,total]).
:- declare_called(partition(X,Y,Z,U),
                 [partition(_,_,_,_),<_,_=<]).

:- declare(app(+[x],+[x],-[x]),[determinate,total]).
:- declare_called(app(X,Y,Z),[app(_,_,_)])
```

The user chooses an example to test QSORT on:

```prolog
@qsort([2,1,3],X).
```
Solving qsort([2, 1, 3], X)...
Error trapped: no solution to partition([3], 2, X, Y)
Error: missing solution partition([3], 2, X, Y). diagnosing...

An uncovered goal has been detected. User query will be necessary

to determine the expected trace so that the bug can be further
isolated.

Query: partition([3], 2, X, Y)? y.
Which (X, Y)? [1, [3]].

Query: partition([1, 2, [1, [1]]]? y.
Error diagnosed: partition([3], 2, [], [3]) is uncovered.
add (y) or (m)odify a clause? m.

Since PDS6 does not know the purpose of the clauses that are
already present, it must rely on user query to inform it whether an
existing clause should have covered the goal, and if so which one.

which?
partition([X|L], Y, L1, [X|L2]) :- partition(Z, Y, L1, L2), X<Y.

Found clause: (partition([X|Y], Z, U, [X|W]) :- partition(Y, Z, U, W))
after searching 1 clauses.

ok? n.

The user must approve proposed changes. The proposed clause is
consistent with the facts provided to PDS6 so far, but the user knows
that this clause should have an arithmetic test, so the clause is
rejected. The user's knowledge about the intended purpose of this
clause and whether or not the proposed change achieves it compensates
for PDS6's ignorance of the clause's teleology.

Found clause:
(partition([X|Y], Z, U, [X|W]) :- partition(Y, Z, U, W), Z<X)
after searching 5 clauses.

ok? y.

Listing of partition(X, Y, Z, U):
(partition([X|Y], Z, [X|W], W) :- partition(Y, Z, V, W), X=<Z).
(partition([X|Y], Z, [X|W]), [1]: true).
(partition([X|Y], Z, U, [X|W]) :- partition(Y, Z, U, W), Z<X).

The original goal is retried automatically:

Solving qsort([2, 1, 3], X)...
Error trapped: no solution to qsort([], X)
Error: missing solution qsort([], X). diagnosing...

Query: qsort([], X)? y.
Which X? [].

Error diagnosed: qsort([],[]) is uncovered.

Again an uncovered goal is found. This time the user manually corrects the bug by adding a missing clause:

add (y) or (m)odify a clause? y.

which? qsort([],[]).

Listing of qsort(X,Y):
(qsort([X|Y],Z):-partition(Y,X,W,X1),
 qsort(W,Z1),
 qsort(X1,Y1),
 app(Z1,[X|Y1],Z)).
(qsort([],[]):-true).

Again the original goal is retried:

Solving qsort([2,1,3],X)... Error: stack overflow on qsort([2,1,3],X). diagnosing...
app([1],[2,3],X) is looping.
Error diagnosed:
(app([1],[2,3],X):-app([1],[2,3],[1|X])).
is diverging.

retract (y), (m)odify, or (r)eplace it? m.

What is a reason for (app([X|Y],Z,U):-app([X|Y],Z,[X|U]))?
ap([1,2],[3,4],[1,2,3,4]).

The instance supplied by the user represents the teleology of the buggy clause since the new clause must also cover it. Again user query compensates for an inability to reason about code teleology.

Refining: (app([X|Y],Z,U):-true)

The questions below are used to prune out incorrect clauses in the search for a bug correction:

Query: app([2],[3,4],[1,2,3,4])? n.
Query: app([3,4],[2],[1,2,3,4])? n.
Query: app([3,4],[1],[1,2,3,4])? n.
Query: app([1],[3,4],[1,2,3,4])? n.
Query: app([2],[4],[1,2,3,4])? n.
Query: app([4],[2],[1,2,3,4])? n.
Query: app([2], [3, 4], [2, 3, 4])? v.

Found clause: (app([X|Y], Z, [X|V]) :- app(Y, Z, V))
after searching 289 clauses.

ok? v.

Listing of app(X, Y, Z):
(app([1], X, X) :- true).
(app([X|Y], Z, [X|V]) :- app(Y, Z, V)).

Solving qsort([2,1,3],X)...
solution: qsort([2,1,3],[1,2,3]); ok? v.
no (more) solutions. ok? v.

The program is correct at this point. Suppose we wish to modify
the existing program to disallow duplicates in the sorted list
returned as output by QSORT. Duplicates will be removed by PARTITION.
Debugging continues with this more stringent specification.

@qsort([2,1,2,3],X).
Solving qsort([2,1,2,3],X)...
solution: qsort([2,1,2,3],[1,2,2,3]) ok? n.
Error: wrong solution qsort([2,1,2,3],[1,2,2,3]).
diagnosing...

The next questions are needed to check whether the output from the
procedures called in QSORT is correct so that the bug can be isolated
to a particular clause in one of the procedures.

Query: qsort([1,2],[1,2])? v.
Query: qsort([3],[3])? v.
Query: app([1,2],[2,3],[1,2,2,3])? v.
Query: partition([1,2,3],2,[1,2],[3])? n.
Query: partition([2,3],2,[2],[3])? n.

Error diagnosed:
(partition([2,3],2,[2],[3]) :- partition([3],2,[],[3]),2=<2)
is false.

retract (y), (m)odify, or (r)eplace it? m.

What is a reason for
(partition([X|Y],Z,[X|V],W) :- partition(Y,Z,V,W),X=<Z)?
partition([1],2,[1],[1]).

Found clause:
(partition([X|Y],Z,[X|V],W) :- partition(Y,Z,V,W),X=<Z,X<Z)
after searching 3 clauses.
ok? y.

Listing of partition(X, Y, Z, U):

(partition([[], X, [ ], [ ]]), true).
(partition([X|Y], Z, U, [X|W])) = partition(Y, Z, U, W), Z<X).
(partition([X|Y], Z, [X|V], W)) = partition(Y, Z, V, W), X=<Z, X<Z).

Note the two tests X=<Z and X<Z in the last clause are redundant.
PD56 does not infer that the test X=<Z is redundant once the test X<Z has been added since it has no knowledge of the computational semantics of =< and <.

Solving qsort([2, 1, 2, 3], X).
Error trapped: no solution to partition([2, 3], 2, X, Y)
Error: missing solution partition([2, 3], 2, X, Y). diagnosing...

Query: partition([2, 3], 2, X, Y)? v.
Which (X, Y)? [1, [3]].
Error diagnosed: partition([2, 3], 2, [1], [3]) is uncovered.
add (y) or (m)odify a clause? m.
which? any.

Found clause:
(partition([X|Y], Z, U, V)) = partition(Y, Z, U, V)
after searching 99 clauses.

ok? v.

Listing of partition(X, Y, Z, U):

(partition([[], X, [ ], [ ]]), true).
(partition([X|Y], Z, U, [X|W])) = partition(Y, Z, U, W), Z<X).
(partition([X|Y], Z, [X|V], W)) = partition(Y, Z, V, W), X=<Z, X<Z).
(partition([X|Y], Z, U, V)) = partition(Y, Z, U, V).

Error: wrong solution partition([3], 2, [1], [1]). diagnosing...
Error diagnosed:
(partition([3], 2, [1], [1])) = false.
retract (y), (m)odify, or (r)eplace it? m.

What is a reason for
(partition([X|Y], Z, U, V)) = partition(Y, Z, U, V)?
partition([2], 2, [1], [1]).

Found clause:
(partition([X|Y], Z, U, V)) = partition(Y, Z, U, V)
after searching 1 clauses.

ok? m.

Found clause:
(partition([X|Y], Z, U, V)) = partition(Y, Z, U, V), X=<Z)
after searching 5 clauses.

ok? y.

Listing of partition(X,Y,Z,U):
(partition([],X,[[],[]]):-true).
(partition([X|Y],Z,U,[X|W]):-partition(Y,Z,U,W),Z<X).
(partition([X|Y],Z,[X|V],W):-partition(Y,Z,V,W),X=<Z,X<Z).
(partition([X|Y],Z,U,V):-partition(Y,Z,U,V),X=<Z).

Solving qsort([2,1,2,3],X)...
Error trapped: too many solutions to partition([1,2,3],2,X,Y)

Query: partition([1,2,3],2,[[],[]])? n.

Error: wrong solution partition([1,2,3],2,[[],[]]).
    diagnosing...
Error diagnosed:
    (partition([1,2,3],2,[[],[]]):-partition([2,3],2,[[],[]]),1=<2)
is false.

retract (y), (m)odify, or (r)eplace it? m.

What is a reason for
    (partition([X|Y],Z,U,V):-partition(Y,Z,U,V),X=<Z)?
partition([2],2,[[],[]]).

Found clause:
    (partition([X|Y],Z,U,V):-partition(Y,Z,U,V),X=<Z,Z=<X)
after searching 3 clauses.

ok? y.

Listing of partition(X,Y,Z,U):
(partition([],X,[[],[]]):-true).
(partition([X|Y],Z,U,[X|W]):-partition(Y,Z,U,W),Z<X).
(partition([X|Y],Z,[X|V],W):-partition(Y,Z,V,W),X=<Z,X<Z).

Solving qsort([2,1,2,3],X)...
   solution: qsort([2,1,2,3],[1,2,3]) ok? y.
   no (more) solutions. ok? y.
III. The Task Library

These are the tasks that are presently stored in the task library of Talus: REVERSE, FLATTEN, SUBSTITUTE, REMOVE, LENGTH, MEMBER, MAX, FACTORIAL, UNION, INTERSECTION, SET-DIFFERENCE, MEMTREE, EXP, SINGLETONS, PALINDROMES, UPDATE-ALIST, BUBBLE-SORT, and COPY-BUT-LAST. Most require definition of simple list and set manipulation functions not in the pure dialect in terms of primitives that are in the pure dialect.

The more complicated tasks are COPY-BUT-LAST, SINGLETONS, and PALINDROMES. SINGLETONS has the most algorithms stored for any task: 3 buggy algorithms and 7 correct algorithms. Typical solutions have four functions since auxiliary functions such as MEMBER, REMOVE, and FLATTEN must be defined along with the main function.

The UPDATE-ALIST and BUBBLE-SORT tasks require the use of side effects. Talus also recognizes algorithms that use property list markers for tasks such as INTERSECTION and SINGLETONS. REVERSE can be written as a destructive reverse (NREVERSE) using RPLACD.

Here are the task assignments:
1. REVERSE - "Write a function to reverse a proper list."
2. FLATTEN - "Write a function to return a proper list containing all the atoms in a tree."
3. SUBSTITUTE - "Write a function that takes three inputs TREE, OLD, and NEW. The function should replace all occurrences of OLD that occur anywhere in TREE by NEW. Thus if the first argument is NEW, the second OLD, and the third TREE, (SUBST '(^ X 2) '(* X X) '(+ Y (* 3 (* X X)))) returns (+ Y (* 3 (^ X 2))). You can call the variables and function anything you want."
4. REMOVE - "Write a function that accepts as input any item and a proper list and returns the list with all occurrences of the item removed from (just) the top level and the remaining items in their original order."
5. LENGTH - "Write a program that determines the length of a proper list."
6. MEMBER - "Write a function that determines whether an item is in a proper list. You need only examine the list at the top level."
7. MEMTREE - "Write a function that determines whether an atom is one of the leaves of a tree."
8. MAX - "Write a function that returns the maximum of a list of natural numbers. Zero should be returned if the list is empty."
9. FACTORIAL - "Write a function to compute the factorial of a natural number."
10. UNION - "Write a function that takes two sets, represented as lists, and returns their union."
11. INTERSECTION - "Write a function that takes two sets, represented as lists, and returns their intersection."
12. SET-DIFFERENCE - "Write a function that takes two sets, represented as lists, and returns their set difference: those elements in the first set not present in the second set."
13. EXP - "Write a function to raise a number X to a power Y. [Hint: Note that if Y is even, then X raised to the Y power is equal to X squared raised to Y divided by two.]"
14. SINGLETONS - "Design a function (SINGLETONS X) which takes as argument an s-expression X and produces as a result a list of all atoms other than NIL which occur
exactly once in \( x \). For example,

\[
\begin{align*}
\text{SINGLETONS of NIL} & \quad \text{is NIL} \\
\text{SINGLETONS of (A)} & \quad \text{is (A)} \\
\text{SINGLETONS of (A (B (E) B (C D)) E) C} & \quad \text{is (A D)} \\
& \quad \text{or (D A)}
\end{align*}
\]

Use a function \( \text{FLATTEN \( X \)} \), where the argument \( X \) is an s-expression with subexpressions nested to any depth, such that the result of \( \text{FLATTEN \( X \)} \) is just a list of atoms with the property that all atoms other than NIL appearing in \( X \) also appear in \( \text{FLATTEN \( X \)} \).

15. PALINDROMES - "Write a function that takes a list of elements and returns, in order, those elements which are lists that are unchanged when reversed. For example,

\[
\begin{align*}
\text{PALINDROMES of } ((A B A) (A D) (E) () ((1 2) (1 2)) 9) \\
\text{is} & \ldots \\
((A B A) (E) ((1 2) (1 2)))
\end{align*}
\]

NOTE: all atoms that are elements in the input list are discarded. Thus,

\[
\text{PALINDROMES of } (X Y Z) \text{ is NIL.}
\]

Assume the input list and any sublists of that list are proper lists."

16. UPDATE-ALIST - "Write a function that will update an alist so that the new value of the pair (KEY . VALUE) is NEW. If no pair with the key KEY exists in the alist then the pair (KEY . NEW) should be added to the end of the alist. If such a pair exists in alist it is unique. The actual alist should be modified. Each alist begins with a dummy header node NIL.

E.g. (update 'size 'large (NIL (color . red)
\[
\begin{align*}
& \quad \text{(size . small)} \\
& \quad \text{(price . cheap))}
\end{align*}
\]

\[
\rightarrow ((color . red) \\
\quad \text{(size . large)} \\
\quad \text{(price . cheap))}
\]

\[
\begin{align*}
\text{(update 'size 'large \( (NIL (color . red) \)
\end{align*}
\]

\[
\begin{align*}
\quad \text{(price . cheap))}
\end{align*}
\]

\[
\rightarrow ((color . red) \\
\quad \text{(price . cheap)} \\
\quad \text{(size . large))}"
\]

17. BUBBLE-SORT - "Write functions to bubble sort an array into increasing order. The input to the calling function consists of the array and the length of the array. The first element of the array has index 1."

18. COPY-BUT-LAST - "Write a LISP function which copies an sexp while replacing the rightmost deepest CONS in the sexp with NIL. Atomic sexps are simply returned.

Examples:

\[
\begin{align*}
\text{(COPY-BUT-LAST 'X)} & \quad \rightarrow X \\
\text{(COPY-BUT-LAST 'A)} & \quad \rightarrow A \\
\text{(COPY-BUT-LAST '(A B C))} & \quad \rightarrow (A B)
\end{align*}
\]
(COPY-BUT-LAST ' (A B (C))) --> (A B NIL)
(COPY-BUT-LAST ' (A B (C . D))) --> (A B NIL)
(COPY-BUT-LAST ' (A B (C (E) D))) --> (A B (C (E)))
IV. Algorithm Representations for MEMTREE

This appendix presents the algorithm representations of the TREE-WALK and MEMTREE-FLATTEN algorithms discussed in Section 4.4. These two algorithms solve the MEMTREE task. The complete E-frames of the algorithms' reference functions are shown here; previously only the reference function definitions were provided. The algorithm representation of the TREE-WALK algorithm is:

FUNCTION-NAME: MEMTREE
FORMALS: (ITEM TREE)
DEFINITION: (DEFUN MEMTREE (ITEM TREE)
   (IF (ATOM TREE)
      (EQUAL ITEM TREE)
      (IF (MEMTREE ITEM (CAR TREE))
         T
         (IF (MEMTREE ITEM (CDR TREE))
            T F))))
NORMALIZED-CODE: (IF (ATOM TREE)
      (EQUAL ITEM TREE)
      (IF (MEMTREE ITEM (CAR TREE))
         T
         (IF (MEMTREE ITEM (CDR TREE))
            T F)))
TERMINATION-FORMALS: (ITEM TREE)
OUTPUT-FORMALS: (ITEM TREE)
VARIABLE-DATA-TYPES: ((ITEM)
   (TREE CONS))
OUTPUT-DATA-TYPE: (BOOLEAN)
CONDITIONS: ((NOT (ATOM TREE))
   (MEMTREE) (ITEM TREE))
   (MEMTREE) (ITEM TREE))
TERMINATIONS: ((NOT (ATOM TREE))
   (NOT (MEMTREE ITEM (CAR TREE)))
   (NOT (MEMTREE ITEM (CDR TREE)))
   F)
   (NOT (ATOM TREE))
   (NOT (MEMTREE ITEM (CAR TREE)))
   (MEMTREE ITEM (CDR TREE)))
   T)
   (NOT (ATOM TREE))
   (MEMTREE ITEM (CAR TREE)))
   T)
   (ATOM TREE))
   (EQUAL ITEM TREE)))
RECURSIONS: ((NOT (ATOM TREE))
   (NOT (MEMTREE ITEM (CAR TREE)))
   (MEMTREE ITEM (CDR TREE)))
   (NOT (ATOM TREE))
   (MEMTREE ITEM (CAR TREE)))
CONSTRUCTIONS: ((NOT (ATOM TREE))
   (NOT (MEMTREE ITEM (CAR TREE)))
   (MEMTREE ITEM (CDR TREE)))
   ((NOT (ATOM TREE)))
   (MEMTREE ITEM (CAR TREE)))
VARIABLE-UPDATES: ((ITEM

The three E-frames for the MEMTREE-FLATTEN algorithm represent the reference functions MEMTREE, MEMBER, and FLATTEN. These E-frames are:

1. For the reference function MEMTREE:

   FUNCTION-NAME: MEMTREE
   FORMALS: (ITEM TREE)
   DEFINITION: (DEFUN MEMTREE (ITEM TREE)
                (MEMBER ITEM (FLATTEN TREE)))
   NORMALIZED-CODE: (MEMBER ITEM (FLATTEN TREE))
   TERMINATION-FORMALS: NIL
   OUTPUT-FORMALS: (ITEM TREE)
   VARIABLE-DATA-TYPES: ((ITEM)
                          (TREE))
   OUTPUT-DATA-TYPE: NIL
   CONDITIONS: NIL
   TERMINATIONS: ((NIL
                    (MEMBER ITEM (FLATTEN TREE))))
   RECURSIONS: NIL
   CONSTRUCTIONS: NIL
   VARIABLE-UPDATES: ((ITEM)
                       (TREE))
   RECURSION-TYPE: (NOT-RECURSIVE)
   FUNCTION-TYPE: CALLING
   FUNCTION-ROLE: TOP
   CONSTRUCTORS-CALLED: (MEMBER FLATTEN)
   PREDICATES-CALLED: NIL
   FUNCTIONS-CALLING: NIL
   SIDE-EFFECTORS: NIL
   DB-FETCH-FNS: NIL
   PROGNS: 0

2. For the reference function MEMBER:
FUNCTION-NAME:  MEMBER
FORMALS:  (ITEM BAG)
DEFINITION:  (DEFUN MEMBER (ITEM BAG)
  (IF (NLISTP BAG)
    F
    (IF (EQUAL ITEM (CAR BAG))
      T
      (MEMBER ITEM (CDR BAG))
    )))
NORMALIZED-CODE:  (IF (NLISTP BAG)
  F
  (IF (EQUAL ITEM (CAR BAG))
    T
    (MEMBER ITEM (CDR BAG))
  ))
TERMINATION-FORMALS:  (ITEM BAG)
OUTPUT-FORMALS:  NIL
VARIABLE-DATA-TYPES:  ((ITEM)
  (BAG CONS))
OUTPUT-DATA-TYPE:  (BOOLEAN)
CONDITIONS:  (((NLISTP) (BAG))
  ((EQUAL) (ITEM BAG)))
TERMINATIONS:  (((NOT (NLISTP BAG))
    (EQUAL ITEM (CAR BAG)))
  ((NLISTP BAG) F))
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-CALLING:  (MEMTREE)
SIDE-EFFECTORS:  NIL
DB-FETCH-FNS:  NIL
PROGNS:  0

3. For the reference function FLATTEN:

FUNCTION-NAME:  FLATTEN
FORMALS:

DEFINITION:

(NORMALIZED-CODE:

TERMINATION-FORMALS:

OUTPUT-FORMALS:

VARIABLE-DATA-TYPES:

OUTPUT-DATA-TYPE:

CONDITIONS:

TERMINATIONS:

RECURSIONS:

CONSTRUCTIONS:

VARIABLE-UPDATES:

RECURSION-TYPE:

FUNCTION-TYPE:

FUNCTION-ROLE:

CONSTRUCTORS-CALLED:

PREDICATES-CALLED:

FUNCTIONS-CALLING:

SIDE-EFFECTORS:

DB-FETCH-FNS:

PROGNs:

(TREE)

(DEFUN FLATTEN (TREE)
 (IF (ATOM TREE)
  (LIST TREE)
  (APPEND
   (FLATTEN (CAR TREE))
   (FLATTEN (CDR TREE))))
)

(IF (ATOM TREE)
  (LIST TREE)
  (APPEND
   (FLATTEN (CAR TREE))
   (FLATTEN (CDR TREE))))

(TREE)

(TREE)

((TREE CONS))

(PROPER-LIST)

(((ATOM) (TREE)))

(((ATOM TREE))
  (LIST TREE)))

(((NOT (ATOM TREE)))
  (FLATTEN (CAR TREE))
  (FLATTEN (CDR TREE))))

(((NOT (ATOM TREE)))
  (APPEND (FLATTEN (CAR TREE))
   (FLATTEN (CDR TREE)))
)

(((NOT (ATOM TREE)))
  (CAR TREE))

(((NOT (ATOM TREE)))
  (CDR TREE)))

(TREE-RECURSION)

RECURSIVE

SUPPORTING-CONSTRUCTOR

NIL

NIL

(MEMTREE)

NIL

NIL

0
V. Handout for CS 381K

*General Instructions:* Please write solutions to the tasks below. Assume a vanilla LISP with these primitives CONS, CAR, CDR, APPEND, LIST, ADD1, SUB1, ZEROP, LESSP, GREATERP, ATOM, LISTP [same as CONSP], NLISTP, NOT, AND, OR, EQUAL, RPLACA, RPLACD, NCONC, PUTPROP, and REMPROP. QUOTE, IF, COND, and DEFUN are available as special forms.

E.g. If APPEND were a task and not a primitive, a solution would be:

```lisp
(defun app (a b)
  (if (null a) b
      (cons (car a) (app (cdr a) b))))
```

The following are not available as primitives: MEMBER, REVERSE, LAST, LENGTH, FLATTEN, and MAX. If you need them they should be defined in the solution. Note that your solutions may require more than one function when auxiliary functions and predicates need to be defined.

*Please do not debug your solutions on a computer!* An answer sheet will be provided if you wish to see answers to these problems.

*Purpose:* The collection of these solutions will allow the performance of Talus, a debugging system capable of debugging solutions to these problems, to be evaluated. Without empirical data of this sort answering performance questions about the system is very difficult.

Don’t feel embarassed if you are rusty in LISP and think some of your solutions may be buggy - I need *real* bugs to evaluate Talus! [On the other hand please don’t intentionally insert bugs in any programs].
LENGTH TASK: Write a program that determines the length of a proper list.
LENGTH TASK: Write a program that determines the length of a proper list. Use a PROG rather than a recursive function.
FLATTEN TASK: Write a function to return a proper list containing all the atoms in a tree.

E.g. \( \text{FLAT '(A (B . D) C)) \rightarrow (A B D C NIL) \)
**COPY-BUT-LAST TASK:** Write a LISP function which copies an sexp while replacing the rightmost deepest CONS in the sexp with NIL. Atomic sexps are simply returned.

E.g.  
(COPY-BUT-LAST 'X) --> X  
(COPY-BUT-LAST 'A) --> A  
(COPY-BUT-LAST '(A B C)) --> (A B)  
(COPY-BUT-LAST '(A B (C))) --> (A B NIL)  
(COPY-BUT-LAST '(A B (C . D))) --> (A B NIL)  
(COPY-BUT-LAST '(A B (C (E) D))) --> (A B (C (E)))
**SINGETONS TASK:** Design a function \( \text{SINGETONS X} \) which takes as argument an s-expression \( X \) and produces as a result a list of all atoms other than NIL (either in order or in reverse order) which occur exactly once in \( X \).

E.g.,

\[
\begin{align*}
\text{SINGETONS of NIL} & \quad \text{is NIL} \\
\text{SINGETONS of (A)} & \quad \text{is (A)} \\
\text{SINGETONS of (A (B (E) B (C D)) E C)} & \quad \text{is either (A D) or (D A)}
\end{align*}
\]

Use a function \( \text{FLATTEN X} \), where the argument \( X \) is an s-expression with subexpressions nested to any depth, such that the result of \( \text{FLATTEN X} \) is just a list of atoms with the property that all atoms other than NIL appearing in \( X \) also appear in \( \text{FLATTEN X} \) and in the same order.
Glossary

Active List - The list of nodes in a best first search that have not yet been expanded.

Actual Bugs - Bugs that are present, whether or not they are detected.

Algorithm - An algorithm is a particular way of solving a problem that specifies a strategy for the problem's solution but leaves out details of the implementation. In Talus algorithms are represented as collections of E-frames (see Algorithm Representation).

Algorithm Level - Design decisions made in choosing an algorithm.

Algorithm Representation - A collection of E-frames that represent an algorithm.

Analysis by Synthesis - A debugging approach where plans are synthesized down to the code level.

Believed Conjecture - A conjecture that the conjecture disprover can find no counterexample to when all stored examples are considered.

Binary Tree Representation - A representation of a function body that is in IF-Normal Form as a binary tree. Nonterminal nodes represent conditional tests and terminal nodes represent symbolic values.

Bug Edit - The replacement of a buggy s-expression by a correct s-expression, or the insertion of a missing conditional test.

Bug Equivalence Classes - A bug equivalence class is a set of code fragments that are perturbations of a buggy code fragment; the perturbations introduced for each bug equivalence class reflect common errors. In the PROLOG debugger PDS6 [Shapiro 83] the code fragments considered are PROLOG clauses. One bug equivalence class consists of variable misspellings. Another is incorrect use of arithmetic tests.

Bug Experts - Demons that detect and generate bug reports for specific bugs.

Bug Fix Nodes - Nodes representing bug edits in the program graph representation.

Bug Rules - Rules that account for discrepancies between stored plans and student code in terms of bugs.

Buggy Algorithm - An algorithm that will not solve the assigned task, even when correctly implemented. The algorithm is "buggy" with regards to the assigned task, but correctly solves some other task that is usually a misreading of the one assigned.

Case - The governing terms of a terminal node in the binary tree representation of a function definition.

Clock - A counter used to ensure termination of extra student functions in evaluation of conjectures.

Commentary Templates - Templates contained in a task representation for the generation of program commentary. Commentary is generated by substituting student identifiers in place of stored identifiers in the template associated with the identified algorithm.

Computational Equivalence - The equivalence of two function definitions with respect to values returned for identical inputs. The two functions defined must be EQUAL for all inputs.
Computational Semantics - The ability to reason about programs, program execution, programming language semantics, and the data structures that are created and processed by programs.

Conjecture-Based Reasoning - Reasoning where decisions are made on the basis of the evaluation of conjectures. The conjectures are well-formed formulas in the Boyer-Moore Logic and their truth value is well defined by formal rules.

Conjecture Disprover - A heuristic means of evaluating conjectures by testing conjectures with stored examples. If a counterexample is found then the conjecture being tested is false, otherwise it is "believed". The conjecture disprover can be the sole means of evaluating conjectures or it can act as a filter to the Boyer-Moore Theorem Prover.

Core Dialect - The restricted LISP dialect of Talus that includes the pure LISP dialect and also primitives that perform side effects on shared list structure, property lists, and arrays.

Counterexample Generator - A heuristic means of evaluating conjectures that attempts to construct counterexamples for conjectures rather than rely solely on stored examples. Talus can use the EGS system [Kim 85] for this purpose.

Counterexamples - Stored examples.

Detected Bugs - Actual bugs for which bug reports are correctly issued. False alarms are not counted.

Dynamic Analysis - Program analysis that requires program execution on specific examples.

Edit Inversion - Back tracing bug edits through program transformations to determine bug edits for the initial version of the student's solution.

Edit Inversion Method - A demon that can override the default means of edit inversion to accommodate program transformations that require special handling.

E-Frame - A frame representation of a LISP function. E-frame slots represent abstract properties of recursive functions and the data structures they enumerate. E-frames are discussed extensively in Chapter 4.

Enforcing a Verification Condition - Altering the student's program so that a particular verification condition will be true in the modified program.

Equivalence Preserving Transform - A transform whose application to a program always results in a program computationally equivalent to the program transformed.

error - A distinguished constant returned to signal a runtime error. For example, (CAR 'A) returns error in the Talus LISP dialects.

Extended Dialect - The restricted LISP dialect of Talus that extends the core dialect by adding special forms for mapping functions and imperative programming.

Extra Conditional Test - Conditional tests that are not strictly necessary to correctly implement a function.
Extra Function - A student function that is unnecessary to the implementation of the identified algorithm.

False Alarms - Bug reports that claim bugs are present in correct code.

Final Partition - The program graph representation of the final version of the student solution.

Final Version - The simplified student's solution, after all program simplification transforms have been applied.

Flow Graph - A graph representation of programs that represents abstract data flow explicitly and is used in the algorithm recognition component [Brotsky 84] of the Programmer's Apprentice.

Function - A LISP function definition, not a mathematical function.

Functional Equivalence - The equivalence of two function definitions with respect to the correct implementation of an algorithm. Two predicates need only be logically equivalent (satisfy the relation IFF) to be functionally equivalent. Two set constructors need only return the same sets (satisfy the relation PERMUTATIONP).

Functional Equivalence Verification Conditions - Verification conditions that compare two symbolic values for functional equivalence.

Function Analysis Failure - A function that cannot be analyzed since its formal variables cannot be mapped to the formal variables of the reference function that it has been paired with. In practice these failures are infrequent.

Function Level - Design decisions regarding the functional decomposition of an algorithm after a particular algorithm has been selected.

Function Recursion - A symbolic value of a function definition that contains a recursive call to the function defined.

Function Role - A function's role in an algorithm's implementation. One of MAIN, TOP, EXTRA, SUPPORTING-CONSTRUCTOR, or SUPPORTING-PREDICATE. Discussed fully in Chapter 4.

Function Termination - A base case of a function definition. Equivalently, a symbolic value containing no recursive calls.

Function Type - One of RECURSIVE, SIMPLE, or CALLING. A RECURSIVE function contains recursive function calls. A SIMPLE function calls only LISP primitives. A CALLING function calls other nonprimitive functions.

Global Solution Transforms - A transform that rewrites the reference functions of an algorithm representation to computationally equivalent functions that more closely match the student's functions. These transforms account for common algorithm variants. For example, if a reference function is not tail recursive and it has been paired with a student function that is, then that reference function will be transformed to an equivalent tail recursive implementation.

Governing Terms - The conditional tests that must be true or false to reach an expression. If a
conditional test \( p \) must be true to reach an expression \( e \) then the governing terms of \( e \) include \( p \). If a conditional test \( p \) must be false to reach \( e \) then the governing terms of \( e \) include (NOT \( p \)).

**Graph** - See Flow Graph or Graph Representation.

**Graph Representation of a Program** - A graph representation of LISP function definitions. Each CONS cell in a LISP function definition corresponds to a node in a program graph. Each node has both CAR and CDR pointers that duplicate the role of the CAR and CDR pointers of the CONS cell represented. In addition, there are PARENT pointers that point to a node’s parent and an SEXP property that caches the node’s list representation.

**Graph Representation of a Program Transformation** - Two program graph representations linked together with pointers indicating s-expressions copied and transformed during the program simplification transformation represented. The first program graph represents the program before the transform was applied; the second program graph represents the program after the transform was applied.

**Heuristic Reasoning** - Reasoning where plausible inferences are derived from the application of heuristics rather than formal rules of inference. The semantics of heuristic reasoning is difficult to state formally, since such inferences are of the nature "\( x \) appears most similar to \( y \)."

**Heuristic Plan Recognition** - A debugging approach where plans are stored in a plan library and partially matched to student code. Bugs are detected by match discrepancies between plans and student code.

**Identified Algorithm** - The stored algorithm that most closely matches the student solution, ignoring buggy algorithms.

**Idempotent** - A function \( f \) is idempotent if \( f(f(x)) = f(x) \) is always true.

**IF-Normal Form** - A form for function bodies in which the only conditional expression is IF and no IF expression occurs as part of the test of another IF expression or inside a function call.

**Implementation** - A collection of LISP function definitions that are syntactically correct.

**Implementation Critic** - The part of an automated debugger that compares task specifications and code abstractions for discrepancies that it interprets as bugs.

**Implementation Level** - The remaining design decisions that must be made after an algorithm and functional decomposition have been selected. These design decisions include choice of identifiers, conditional tests, ordering of conditional tests, variable roles, and specific programming constructs used.

**Initial Partition** - The program graph representation of the initial version of the student solution.

**Initial Version** - The original student solution before program simplification transforms were applied.

**Intention Based Program Diagnosis** - Bug detection based on an understanding of the intended purpose of the student’s code. Johnson demonstrates the contribution of intention based program diagnosis to automated debugging with PROUST [Johnson 85]. This dissertation does not deny the importance of intention based program diagnosis but rather stresses that reasoning about computational semantics is an equally important part of automated debugging.
Intelligent Tutoring System - An intelligent tutoring system is a computer based system intended to provide effective, appropriate, and flexible instruction to students. Artificial intelligence techniques are usually required for domain expertise and student modelling.

ITS - Intelligent Tutoring System.

List - Any CONS.

Logically Overlapping Predicates - Two predicates that are not logically equivalent but where the first predicate (or its negation) implies the second predicate (or its negation).

Misconception - A misunderstanding that can lead to a program bug.

Missed Bugs - Undetected actual bugs.

Missing Conditional Test - Conditional tests that are necessary to a function's correct implementation. These are nonstylistic bugs since either nontermination, errors, or termination with incorrect values result.

Missing Condition - A missing conditional test that is not a missing guard.

Missing Example - An example necessary to invalidate a false conjecture that will be believed otherwise.

Missing Function - Functions essential to the implementation of the identified algorithm that are not present in the student's solution.

Missing Guard - A missing conditional test in a student function whose absence results in symbolic evaluation to error when a particular reference case is assumed.

Model Tracing - A debugging approach where a student's program design is analyzed incrementally as the solution is coded step by step. A model of correct and incorrect program design is used to trace the student's design. The LISP tutor GREATERP [Reiser 85] exemplifies this approach.

Nonprimitive - A function defined by the student.

Nonstylistic Bug - An error in program design that causes a program to return incorrect results, to incur a runtime error, or to loop until system resources are exhausted. For example, a missing conditional test is a nonstylistic bug.

Normalization - The replacement of identifiers in reference function definitions by identifiers in student function definitions.

One-to-One Transforms - Program simplifications transforms for which simple edit inversion is adequate.

Oracle - A source of information outside of a debugging system. This source is assumed to answer all questions and to answer them correctly. Frequently the oracle is the user. An oracle is required for PDS6 [Shapiro 83] but not for Talus.

Partition - A program graph representation of a particular program version.

Picture Model - The task specification of a simple picture to be drawn by a LOGO program in
MYCROFT [Goldstein 74].

Plan - A mapping from a task goal to either code or an abstract representation of code that achieves that goal. For example, in PROUST plans relate task goals to parameterized code templates.

Plan-Difference Rules - Rules that account for discrepancies between stored plans and student code in terms of implementation variants (plan transformation rules) or bugs (bug rules).

Plan Parsing - A debugging approach where grammars represent plans and plan difference rules, and where student programs are parsed in terms of the grammar.

Plan-Based Program Analysis - A debugging approach that relies on plans to determine a mapping between task specifications and student code.

Plan Transformation Rules - Rules that account for discrepancies between stored plans and student code in terms of implementation variants.

Plausibility Constraints - Constraints on acceptable function mappings. These reduce the search space for the best function mapping from reference functions to student functions.

Principle of Definition - The series of conditions that must be true for a function definition to be admitted to the Boyer-Moore Logic. These conditions ensure that the function defined is total and will not render the logic inconsistent.

Principle of Induction - The series of conditions that must be true for a conjecture to be proved by induction in the Boyer-Moore Logic.

Primitive - A built in function present in the extended dialect of Talus, not one defined by the student.

Procedure - Refers to either a subroutine or a function. A subroutine is called for side effect only; a function returns one or more values. In the current implementation of Talus the only procedures are functions that return single values.

PROG Block - A part of a PROG expression, delimited either by tags in the PROG expression or the boundaries of the PROG expression.

Program Simplification Transformation - The application of a program simplification transform to a program version to produce a new program version.

Program Simplification Transforms - Transforms that rewrite programs to facilitate program analysis while not altering the values returned by the programs.

Program Verification Approaches - A debugging approach based on an attempt to prove that the program being debugged meets certain formal specifications. When the proof fails the program is altered to repair the proof.

Program Version - A program that results from applying program transformations to the student's solution.

Proper List - NIL, or a CONS ending in NIL.
Pure Dialect - The restricted LISP dialect of Talus that contains only pure LISP. The pure dialect contains the basic pure LISP functions for constructing pairs, lists, and numbers. Accessors and recognizers for these data objects are also provided, along with special forms required to define recursive functions that operate on these data structures.

Rational Form Violations [Goldstein 74] - Program bugs or violations of rules of programming discourse that can be recognized without knowledge of the assigned task. For example, unreachable code is a rational form violation.

Recursion - A symbolic value containing recursive calls.

Redundant Conditional Test - A predicate that is the test part of an IF expression and which will always be true or false when reached.

Reference Case - A case in the reference function currently being discussed.

Reference Function - A function definition that is both stylistically and computationally correct.

Repair Heuristic - An approach to bug correction where heuristics suggest plausible bug edits for specific bug manifestations.

Shell Principle - A means by which new data types can be added to the Boyer-Moore Logic.

Side-Effector - A LISP primitive that can cause a side effect (e.g. PUTPROP) or whose value depends on the previous occurrence of side effects (e.g. GET).

Significant Variability - The large variability in student solutions due to different choices among algorithms, functional decompositions, identifiers, role of formal variables, data flow, control flow, values returned by functions, and programming constructs.

Simple Edit Inversion - A default means of regressing edits through program transformations from the final version to the initial version of the student’s program. This method relies on following pointers connecting s-expressions that were copied or transformed in program transformations.

Sniffers - Bug experts in SNIFFER [Shapiro 81].

Static Analysis - Program analysis that does not involve program execution on specific examples. No program interpreter or compiler is required for static analysis.

Stored Algorithm - One of the algorithms represented in the task solution.

Stored Function - A reference function.

Stored Solution - The reference functions associated with the identified algorithm in the task representation.

Strict - A function is strict if it returns error when any of its arguments evaluates to error.

Student Case - A case in the student function currently being discussed.

Student Function - A function definition in the student’s solution.
Student Solution - One or more function definitions intended to solve an assigned task. The student solution is not necessarily correct.

Stylistic Bug - An error in programming style that does not necessarily cause a program to return incorrect results, but does suggest student misconceptions. For example, unnecessary conditional tests are stylistic bugs.

Symbolic Evaluation - The simplification of a function body to symbolic values when a hypothesis is assumed. Discussed in Section 6.1.

Symbolic Value - The s-expression represented by a terminal node in the binary tree representation of a function in IF-Normal Form.

Task - An assignment intended to exercise or test the student’s knowledge of the programming language and programming concepts being taught.

Task Library - A database of task representations.

Task Representation - A frame representing a task. Slots specify the task’s textual description, known algorithms and their representations, and examples for conjectures involving the algorithm representations’ reference functions.

Task Specifications - Precise criteria for successful completion of a task.

Teleology - The intended purpose of program code. Often expressed as the achievement of some part of the assigned task.

Termination - 1. The completion of a program’s execution. 2. A symbolic value containing no recursive calls.

Termination Verification Conditions - Verification conditions that ensure termination in debugged student functions.

Top-Level Function in a Solution - A function that has no callers.

Top-Level Function in an S-Expression - In a function call \( f \text{arg}_1...\text{arg}_n \), \( f \) is the top-level function.

Top-Level S-expression - An s-expression \( s \) that occurs in code is top-level if the only s-expression containing \( s \) is code.

Verification Conditions - Conjectures that must be true to establish the correctness of a program with respect to some specification. In Talus these specifications are reference function definitions and verification conditions establish the functional equivalence of a student function to a reference function.

Well-Founded Relation - A relationship that does not permit an infinite sequence of objects, each smaller than its predecessor according to the relation. For example, LESSP is a well founded relation over the natural numbers \( (0, 1, 2...) \).

Wrong Termination Bugs - Bugs in function terminations.
Wrong Recursion Bugs - Bugs in function recursions.
References

[Adam 80] A. Adam and J. Laurent.
LAURA, A System to Debug Student Programs.

[Barr 76] A. Barr, M. Beard, and R. Atkinson.
The Computer as a Tutorial Laboratory.

*A Computational Logic.*

[Brotsky 84] D. C. Brotsky.
*An Algorithm for Parsing Flow Graphs.*

[Clancey 79] W. Clancey.
Tutoring Rules for guiding a case method dialogue.

*Understanding Simple Picture Programs.*

A Formal Basis for the Heuristic Determination of Minimum Cost Paths.

Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'.

Personal communication.
May, 1986
Discussion about the origin of the distinction between debuggers performing static
analysis and those performing dynamic analysis.

[Johnson 84] W. L. Johnson and E. Soloway.
Intention-based Diagnosis of Programming Errors.
American Association for Artificial Intelligence, August, 1984.

[Johnson 85] W. L. Johnson.
*Intention-Based Diagnosis of Errors in Novice Programs.*


Logical Analysis of Programs.
EGS: A Transformational Approach to Automatic Example Generation.

Domain Specific Debugging Aids for Novice Programmers.

Understanding and Debugging Programs.

Structured Planning and Debugging.

Planning and Debugging in Elementary Computer Programming.

A Structured Planning and Debugging Environment for Elementary Programming.

Intelligent Tutoring for Programming Tasks.

Personal communication.
April, 1986
Informal description of Computer*Thought's Ada Tutor and previous discussions on automated program debugging.

Principles of Artificial Intelligence.
Tioga, Palo Alto, California, 1980.

Concise Encyclopedia of Greek and Roman Mythology.
Entries under Talus and Daedalus.

Dynamic Student Modelling in an Intelligent Tutor for Lisp Programming.

Inspection Methods in Programming.
Artificial Intelligence.

Personal communication.
April, 1986
Characterization of plan-based program analysis approaches to debugging.

Analysis of Algorithm Implementations.

Dictionary of Pronunciation.

Scripts, Plans, Goals, and Understanding.

[Shapiro 81] D. Shapiro.
Sniffer: A System that Understands Bugs.
MIT A.I. Memo 638.
1981

Algorithmic Program Debugging.
MIT Press, MIT, 1983.

[Shrobe 79] H. E. Shrobe.
Dependency Directed Reasoning for Complex Program Understandings.

Intelligent Tutoring Systems.

MENO-II: An AI-Based Programming Tutor.

[Tripp 70] E. Tripp.
Crowell's Handbook of Classical Mythology.
Index

A* algorithm 101, 230
Active list 95, 257
Actual bugs 214, 257
Algorithm 85, 257
Algorithm level 41, 58, 145, 257
Algorithm recognition 17, 85, 93, 230
Algorithm representation 86, 89, 257
Algorithm variants 49, 108
Analysis by synthesis 26, 35, 228, 257
Analysis of loop invariants 22, 34
Anomalous conditional tests 147

Believed 55, 139, 257
Binary tree representation 47, 50, 82, 119, 257
BIP 2, 22, 36
Bookkeeping E-frame slots 78
Bottom objects 75
Boyer-Moore Logic 5, 14, 51, 70, 229
Boyer-Moore Theorem Prover 5, 6, 51, 55, 140, 229
BUBBLE-SORT task 244
Bug edits 257
Bug equivalence classes 27, 257
Bug experts 257
Bug fix nodes 163, 257
Bug isolation 156
Bug rules 26, 30, 35, 231, 257
Buggy algorithm 145
Buggy algorithms 42, 57, 88, 209, 257
Bugs versus misconceptions 3

Case 50, 124, 257
Case splitting 16, 23, 51, 60, 121
Clock 140, 257
Code abstractions 21
Code teleology 4, 23
Commentary templates 59, 88, 117, 146, 257
Computational equivalence 257
Computational semantics 1, 4, 229, 257
cond-diff 103, 105
COND-TO-IF program simplification transform 64, 187
CONDITIONS E-frame slot 80, 103
Conjecture disprover 55, 138, 230, 258
Conjecture evaluation 6
Conjecture-based reasoning 71, 230, 258
CONSTRUCTIONS E-frame slot 78, 82
CONSTRUCTORS-CALLED E-frame slot 79, 84, 103, 209
COPY-BUT-LAST task 210, 244
Core dialect 39, 174, 258
correct 155
Corresponding recursions 86
Counterexample generator 6, 139, 258
Counterexamples 22, 91, 139, 258
Cross-fertilization 137

DB-FETCH-FNS E-frame slot 81, 103

Debugging approaches
  analysis by synthesis 26, 35, 228, 257
  analysis of loop invariants 22, 34
  debugging recursive programs by repairing induction proofs 6, 34, 229
  dynamic analysis 22, 34, 227, 258
  heuristic plan recognition 26, 35, 228, 260
  plan parsing 26, 35, 228
  plan-based program analysis 24, 262
  program verification debugging approach 24, 35, 228, 262
  static analysis 34, 22, 227, 263

See also BIP, GREATERP, LAURA, MENO-II, MYCROFT, PDS6, PROUST, SNIFER, TURTLE

Debugging recursive programs by repairing induction proofs 6, 34, 229

DEFUSE-DESTRUCTIVE-FNS program simplification transform 194

Detected bugs 214, 258
discard 103, 105, 210
discard1 105
discard2 105

Dynamic analysis 227, 22, 34, 258

E-frame slots
  CONDITIONS 103
  CONSTRUCTIONS 78, 82
  CONSTRUCTORS-CALLED 84, 103, 209
  DB-FETCH-FNS 103
  FORMALS 103
  FUNCTION-NAME 103
  FUNCTION-ROLE 209
  FUNCTION-TYPE 209
  FUNCTIONS-CALLING 84, 209
  OUTPUT-DATA-TYPE 78, 103
  OUTPUT-FORMALS 113
  PREDICATES-CALLED 84, 103, 209
  PROGS 103
  RECURSION-TYPE 103
  RECURSIONS 78, 82
  SIDE-EFFECTORS 103
  TERMINATION-FORMALS 113
  TERMINATIONS 78, 103
  VARIABLE-DATA-TYPES 78, 113
  VARIABLE-UPDATES 78, 113, 82

E-frames 48, 75, 83, 247, 258
Edit inversion 19, 174, 258
Edit inversion methods 198, 201, 258
Enforcing verification conditions 258, 12, 52
Equivalence preserving 199, 258
error 129, 258
Error handling 140, 258
EXP task 243
Extended dialect 39, 174, 258
Extra conditional tests 42, 51, 124, 128, 147, 258
EXTRA function role 80, 84, 105
Extra functions 42, 146, 258
EXTRA, mapping a function to 48, 96, 105
EXTRA, mapping a PROGN expression to 184

FACTORIAL task 64, 141, 243
False alarms 4, 6, 22, 26, 30, 138, 214, 259
Final partition 175, 259
Final version 175, 259
FLATTEN task 119, 243
Flow graphs 93, 259
Forced assumptions 127
FORMALS E-frame slot 103
Fresh list structure 68, 179
Function 259
Function analysis failures 214, 259
Function level 41, 145, 259
Function recursion type 76
Function recursions 42, 50, 82, 259
Function role 48, 100, 105, 107, 259
Function terminations 42, 48, 50, 77, 82, 259
Function type 100, 105, 107, 259
FUNCTION-NAME E-frame slot 103
FUNCTION-ROLE E-frame slot 80, 209
FUNCTION-TYPE E-frame slot 80, 209
Functional equivalence 133, 259
Functional equivalence verification conditions 131, 51, 56, 230, 259
FUNCTIONS-CALLING E-frame slot 79, 84, 209

g 107, 210
Global solution transforms 49, 54, 259
Governing terms 50, 259
Graph representation 64, 162, 231, 260
GREATERP 2, 3, 22, 27, 32, 35, 36, 169
GUIDON 2

h 107, 210
Heuristic plan recognition 26, 35, 228, 260
Heuristic reasoning 17, 71, 229, 260
Hint generation 165
Hint templates 165

Idempotent 111, 260
IDEMPOTENT-TRANSFORM global solution transform 111
Identified algorithm 260
IF-Normal Form 47, 81, 260
Imperative programming languages 231
Imperative programs 35, 39, 228, 230
Implementation 87, 260
Implementation Critic 21, 260
Implementation level 41, 145, 260
Implicit data restrictions 41, 145, 225
Inefficient algorithm 145
Initial partition 175, 260
Initial version 175, 260
Intelligent tutoring system 1, 2, 260
Intention based program diagnosis 4, 260
  See also code teleology
INTERSECTION task 243
isolate 155
isolate* 158
ITS 1, 231, 261
  See also intelligent tutoring system

LAMBDA-EXPANSION program simplification transform 193
LAURA 22, 26, 91
LENGTH task 243
List 261
  See also proper list
Logically overlapping predicates 151, 261

MACROEXPANSION program simplification transform 195, 203
MAIN function role 80, 100
Map to EXTRA 48
MAP-FN-TO-PROG program simplification transform 192
Mapping, leaves of one tree to another 125
Mark a function 84
march 102, 104, 210
MAX task 152, 188, 243
Measure 16, 51, 90, 135
MEMBER task 15, 60, 129, 243
MEMTREE task 16, 44, 53, 117, 139, 141, 146, 195, 231, 243, 247
MEMTREE-FLATTEN algorithm 53, 86, 248
MENO-II 22, 26, 36, 92
Misconceptions 3, 35, 165, 261
Missed bugs 214, 261
Missing conditional tests 15, 42, 51, 60, 124, 147, 261
Missing conditions 148, 261
Missing examples 214, 261
Missing functions 42, 59, 146, 261
Missing guards 148, 261
Missing PROGN expressions 183
Model tracing 32, 261
MYCROFT 22, 23, 168

NCONC 18, 68
Node, search space 96
Nonprimitive 261
Nonstylistic bugs 261, 3, 42, 146
Nontermination 231
  See also measure, well-founded relation, termination verification conditions
Nontrivial PROGNS 81, 103
Normalization 13, 50, 54, 60, 115, 184, 261
Normalization and instantiation of induction hypotheses 13, 50, 137, 231
  See also cross-fertilization

One-to-one transforms 198, 261
Oracle 5, 28, 261
TAIL-RECURSION-INTRODUCTION global solution transform 109, 113
Talus 1, 235
Task
  BUBBLE-SORT 244
  COPY-BUT-LAST 210, 244
  EXP 243
  FACTORIAL 64, 141, 243
  FLATTEN 119, 243
  INTERSECTION 133, 243
  LENGTH 243
  MAX 152, 188, 243
  MEMBER 15, 60, 129, 243
  MEMTREE 16, 44, 53, 117, 139, 141, 146, 231, 243, 247
  PALINDROMES 18, 64, 244
  REMOVE 203, 243
  REVERSE 69, 141, 202, 243
  SET-DIFFERENCE 243
  SINGLETONS 57, 139, 210, 243
  SUBSTITUTE 243
  UNION 243
  UPDATE-ALIST 244
Task library 40, 264
Task representation 41, 88, 133, 264
Task specifications 21, 264
Tasks 264
Teleology 264
Termination 264
  See also function terminations, nontermination
Termination verification conditions 51, 63, 131, 135, 231, 264
TERMINATION-FORMALS E-frame slot 80, 113
Terminations 264
TERMINATIONS E-frame slot 78, 82, 103
theorem 138
TOP function role 80, 100
Top-level function in a solution 80, 264
Top-level function in an s-expression 264
Top-level s-expressions 156, 264
TREE-WALK algorithm 53, 86, 247
TURTLE 22, 36, 169

Unanticipated implementations 4, 228
Undecidability of Boyer-Moore Logic 6, 52
Undecidability of predicate calculus 6
Undecidability of program equivalence 6
UNION task 243
Unnecessary PROGN expressions 183
UPDATE-ALIST task 244
updates 135

Variable mappings 113
Variable role 113
Variable updates 48, 78, 82
VARIABLE-DATA-TYPES E-frame slot 78, 113
VARIABLE-UPDATES E-frame slot 78, 82, 113
Verification conditions  6, 264
\textit{vmatch}  113, 210

Well-founded relation  16, 51, 135, 264
Wrong recursion bugs  42, 264
Wrong termination bugs  42, 264