MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper No. 321 June 1989

Determining the Limits of Automated Program Recognition

Linda M. Wills

Abstract

Program recognition is a program understanding technique in which stereotypic computational structures are identified in a program. From this identification and the known relationships between the structures, a hierarchical description of the program's design is recovered. The feasibility of this technique for small programs has been shown by several researchers. However, it seems unlikely that the existing program recognition systems will scale up to realistic, full-sized programs without some guidance (e.g., from a person using the recognition system as an assistant). One reason is that there are limits to what can be recovered by a purely code-driven approach. Some of the information about the program that is useful to know for common software engineering tasks, particularly maintenance, is missing from the code. Another reason guidance must be provided is to reduce the cost of recognition. To determine what guidance is appropriate, therefore, we must know what information is recoverable from the code and where the complexity of program recognition lies. I propose to study the limits of program recognition, both empirically and analytically. First, I will build an experimental system that performs recognition on realistic programs on the order of thousands of lines. This will allow me to characterize the information that can be recovered by recognizing the code and will provide insights into what cannot be recovered by this code-driven technique. Second, I will formally analyze the complexity of the recognition process. This will help determine how guidance can be applied most profitably to improve the efficiency of program recognition.

Contents

1 Introduction

The essential information that needs to be understood about a program is that which is used in performing common software engineering tasks, such as maintaining, enhancing, verifying, and debugging the program. A wish list of the types of information commonly perceived as being necessary for these activities is given in Figure 1. Their usefulness is evidenced by psychological experiments (such as [33, 30, 59, 64]) that study programmers engaged in various software engineering activities and by the widespread interest in capturing this knowledge during the development process (for example, [9, 40, 46, 68]).

```
- design described on various levels of abstraction
- implementation relationships between the components of the design
- purpose of each component
- dependencies among design components
- dependencies among design decisions
- alternative choices in design decisions
- design rationale (implementation guidelines)
- how goals were reformulated and refined during development
- tradeoffs made and preferences that guided making them
- optimizations performed
- location and localization of cliched computational structures
- input and output conditions on the program and its components
- bug manifestations
- location of known or potential bugs
- near-misses of stereotypic computational structures
- behavior of code
- intended behavior and purpose of code
```

Figure 1: Useful Information to Understand About a Program

Of this desired knowledge, the amount that is available for a given program ranges along the spectrum shown in Figure 2. The left extreme is a complete record of the program's development and needs for improvement. For example, it is the knowledge possessed by the program's author(s) just after the code was developed. A complete record like this rarely exists; for huge programs, often no one possesses full knowledge.

At the other end of the spectrum, only the code is available, with no documentation, comments, or any clues as to the purpose of the code. Understanding the program must

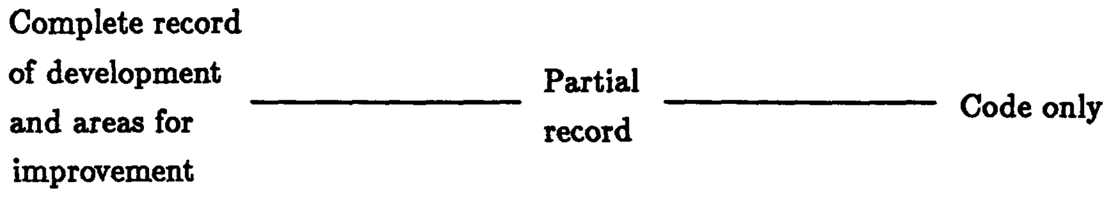


Figure 2: Range of Available Knowledge about Program

be an entirely code-driven process. This means that not all of the information that needs to be understood about the program will be recovered, since some of it is not contained in the code. For example, the rationale behind design decisions may not be clear from the code alone. In addition, having no knowledge of the purpose of the code means no guidance can be given to reduce the cost of this code-driven analysis.

Fortunately, a more typical situation occurs midway between these two extremes. The code usually has some associated documentation (e.g., comments and mnemonic variable and procedure names) which gives incomplete and possibly inaccurate information about the code and its purpose. From this, expectations about how the program was designed are derived. These are then confirmed, amended, or completed by checking them against the code. In this situation, two complementary processes are co-operating, one purpose-driven and the other code-driven. The purpose-driven process uses the partial record to provide information not recoverable from the code and to guide the code-driven process by generating expectations about the design. Meanwhile, the code-driven process fills in the gaps in the partial design information and verifies or rejects the expectations.

In designing such a hybrid program understanding system, the appropriate division of labor between the two processes depends on the relative strengths, limitations, and computational expense of each process. Neither process has been developed fully or studied formally to the extent that these things are known. I propose to move toward the goal of combining the two processes by studying the limits and complexity of a code-driven process.

The particular code-driven technique I will focus on is *program recognition*. This is a process in which occurrences of stereotypic computational structures are identified in a program. A hierarchical description of the program's design is generated based on the structures found and their known relationships.

Studying program recognition involves two main research efforts. First, it entails seeing how much can be recovered by recognition of the code, given unlimited time and space. To do this, I will build an experimental system which performs program recognition on realistic programs. Current program recognition systems deal with only a very restricted class of programs containing a subset of programming language features. The programs in this class are also very small (on the order of tens of lines). The experimental system will extend this class by handling more language features and program sizes on the order of thousands of lines.

Any code-driven approach can only recover the subset of information about the program that is contained in the program's code. The experimental system will help identify the types of information that are in this subset, namely, those types that it can extract. It cannot completely characterize the recoverable subset for two reasons. One is that in the future, there may be other code-driven techniques that can extract additional types of information. The other reason is that it is not possible to prove that some type of information is not in this subset on the basis of the capabilities of one particular system. The goal of building the experimental system is to determine what kinds of information the code-driven portion of a hybrid program understanding system can be responsible for recovering and to identify which types of information are likely to be difficult or impossible to recover by a code-driven technique. Since it is impossible to characterize all of the possible types of information that can be known about a program, the best I can do is to focus on those types that are known to be useful for a maintainer, programming tutor, user, or anyone who works with the code. I will take the information types listed in Figure 1 as my target set. To recover some of these types of knowledge about a program (e.g., design rationale), it is very likely that more is needed than just the recognition of the code. However, for others (e.g., dependencies among design decisions), this is less clear.

My second main research effort in studying program recognition involves analyzing the complexity of the recognition process used by the experimental system. This will reveal the inherent bottlenecks in the technique and will provide insights into how purpose-driven guidance can best be applied to reduce the cost.

Ultimately, the limits of the purpose-driven approach must also be delineated. As more of the information needed for future software engineering activities is recorded during development (see [9, 40, 46, 68]), the available knowledge about the code moves leftward on the spectrum in Figure 2, becoming more accurate and complete. In the extreme, all information needed can be recorded ahead of time, making it unnecessary to ever do a code-driven analysis. However, it is unlikely that this extreme will be reached. It may be that some of the information can be more efficiently extracted from the code rather than being redundantly stored in the design record. Finding the limits of the code-driven recognition technique will be relevant in making this tradeoff. This research will point out what should be recorded during development because it is difficult or costly to recover from the code.

This proposal is organized as follows. Section 2 discusses the process of program recognition and describes a prototype showing its feasibility. Section 3 describes the proposed full-scale program recognition system. Section 4 discusses how the complexity analysis will be useful. A comparative summary of other program recognition work is given in Section 5. Section 6 points to work in areas outside of program recognition which uses graph and attribute grammar technology similar to that used by the prototype recognition system.

2 Program Recognition

Program recognition is the process of identifying occurrences of stereotypical programming structures, called *clichés* [47, 51], in a program. From this identification, a hierarchical description of a design of the program is generated, based on the known behaviors of the clichés and the relationships between them.

In order to develop an intuition for the way program recognition can contribute to understanding a program, consider the program in Figure 3. A salient feature of this program is the clichéd use of `CAR`, `CDR`, and `NULL` in a loop to enumerate the list `BUCKET`. The comparisons made between `ELEMENT` and `ENTRY` (which lead to terminating the loop) indicate that the loop is a membership test for `ELEMENT` in the list `BUCKET` and that the list `BUCKET` is ordered. Finally, the way `BUCKET` is computed from `STRUCTURE` and `ELEMENT` using `HASH` indicates that `STRUCTURE` is a hash table and, therefore, that the program as a whole is testing for membership of `ELEMENT` in this hash table.

Several experiments [20, 30, 56, 67] give empirical data supporting the psychological reality of clichés and their role in understanding programs. Programmer's rely heavily on their accumulated programming experience when analyzing programs, rather than reasoning from first principles. In automating program recognition, the goal is not to mimic the human *process* of recognition, but to mimic only the use of experiential knowledge in the form of clichés to achieve a similar result.

```
(DEFUN HT-MEMBER (STRUCTURE ELEMENT)
  (LET ((BUCKET (AREF STRUCTURE (HASH ELEMENT)))
        (ENTRY NIL))
    (LOOP
      (IF (NULL BUCKET) (RETURN NIL))
      (SETQ ENTRY (CAR BUCKET))
      (COND ((STRING> ENTRY ELEMENT) (RETURN NIL))
            ((EQUAL ENTRY ELEMENT) (RETURN T)))
      (SETQ BUCKET (CDR BUCKET)))))
```

Figure 3: Undocumented Common Lisp Code

2.1 Difficulties in Automating Program Recognition

Although it seems effortless for people to recognize common patterns in code, program recognition is difficult to automate. The following are the main problems that have been focused on thus far in program recognition research:

- **Syntactic variation** — There are typically many different ways to achieve the same net flow of data and control (e.g., the program in Figure 3 could have had more temporary variables, or used `DO` instead of `LOOP`).

- **Implementation variation** — There are typically many different concrete algorithms that can be used to implement a given abstraction (e.g., the hash table in Figure 3 could have been implemented using a double hashing scheme rather than bucket lists).

- **Overlapping implementations** — Optimizations made to a program often merge the implementations of two or more distinct abstract structures. It is essential to allow a program fragment to be described as playing a role in more than one cliché.

- **Unrecognizable code** — Not all programs are completely constructed out of clichés. Recognition must be able to ignore an unforeseeable amount of unrecognizable structure.

- **Diffuse structures** — Pieces of a cliché are sometimes widely scattered throughout the text of a program, rather than being contiguous. (As shown in [33], this can be a significant source of difficulty for human program recognition as well.)

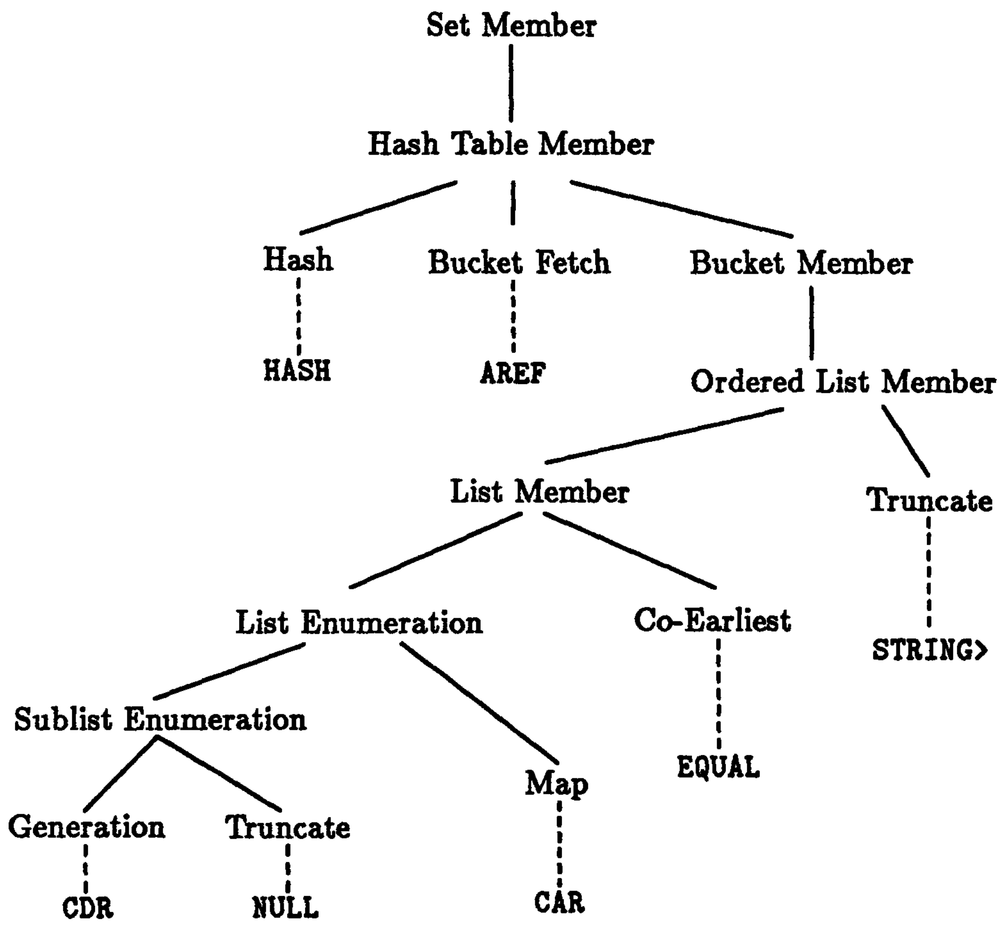


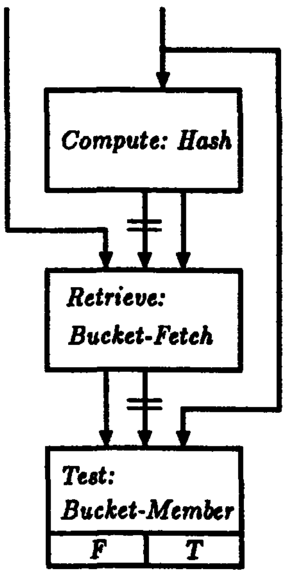
Figure 4: The Recognizer's Hierarchical Description of HT-MEMBER

2.2 A Feasibility Demonstration

The Recognizer [69, 70] is a prototype program recognition system, which deals with these problems. (Other program recognition systems, which deal with a subset of these problems, will be described in Section 5.) Given a program and a library of clichés, the Recognizer finds all occurrences of the clichés in the program and builds a hierarchical description of the program in terms of the clichés found and the relationships between them. (In general, there may be several such descriptions.)

Figure 4 is a simplified version of the design of `HT-MEMBER` as recognized by this system. The internal nodes of the tree indicate the clichés found in the program and how they relate to each other. The leaves of the tree point to the location of a particular cliché in the code (for example, the call to `AREF` is described as a bucket fetch operation).

The Recognizer is able to overcome the obstacles to automated program recognition



Hash-Table-Member

Figure 5: A sample plan diagram representing a test for hash table membership. Solid arcs denote data flow; cross-hatched arcs denote control flow. A branch in control flow is represented by a *test specification* (e.g., the node labeled "Test").

listed above by using an abstract representation of programs and an exhaustive recognition technique. Rather than dealing with a program in its textual form, the Recognizer uses the Plan Calculus representation of Rich, Shrobe, and Waters [47, 48, 49, 51, 55, 65]. In the Plan Calculus, the operations and data structures of a program and the data and control flow constraints between them are represented in a *plan.* Plans are depicted as directed, acyclic graphs, called *plan diagrams*, in which nodes represent the operations or parts of a data structure, and arcs denote the data and control flow constraints between them. For example, Figure 5 shows a plan diagram for one way to test for membership of an element in a hash table. This representation allows the Recognizer to deal with syntactic variation among programs by abstracting away from the syntactic features of the code and exposing its algorithmic structure. Using this representation also addresses the problem of diffuse structure, since many clichés are much more localized in the connectivity of the data and control flow graphs than in the program text.

The library of clichés represented in the Plan Calculus explicitly captures the implementation and specialization relationships between the clichés in the form of *overlays.* Each overlay consists of two plans and a set of correspondences between them. It formalizes the notion that one plan may be viewed as an instance of another. This enables

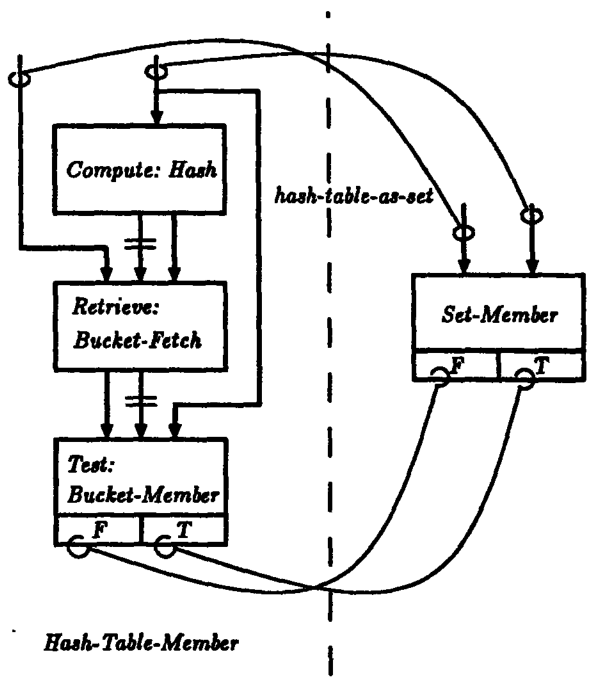


Figure 6: An Overlay for an Implementation of Set Member.

the Recognizer to describe program fragments as clichés on multiple levels of abstraction, thus dealing with implementation variation. For example, the overlay depicted in Figure 6 specifies that one way of implementing the cliché Set Member is by using the plan Hash-Table-Member. The correspondences between the two plans in the overlay are shown as hooked lines between the two sides. One of these correspondences is labeled with the name of another overlay called "hash-table-as-set". This is a data overlay specifying the data abstraction that a hash table can be viewed as a set. The other correspondences in the overlay are simple equality correspondences that do not involve any data abstractions.

Aside from the Recognizer's representation of programs and programming knowledge, the second key aspect of the system is that in searching for patterns in the code, it employs an exhaustive algorithmic technique, based on graph parsing. This approach is based on an analogy between the implementation of high-level programming operations in terms of lower-level operations and the derivation of a graph by systematically replacing given nodes with specified subgraphs. By representing a program as a graph and capturing allowable implementation steps between clichés in graph grammar rules, the Recognizer is able to parse the program's graph to obtain a hierarchical description of the program's design.

Figure 7 shows the Recognizer's architecture. Given a program, the system performs

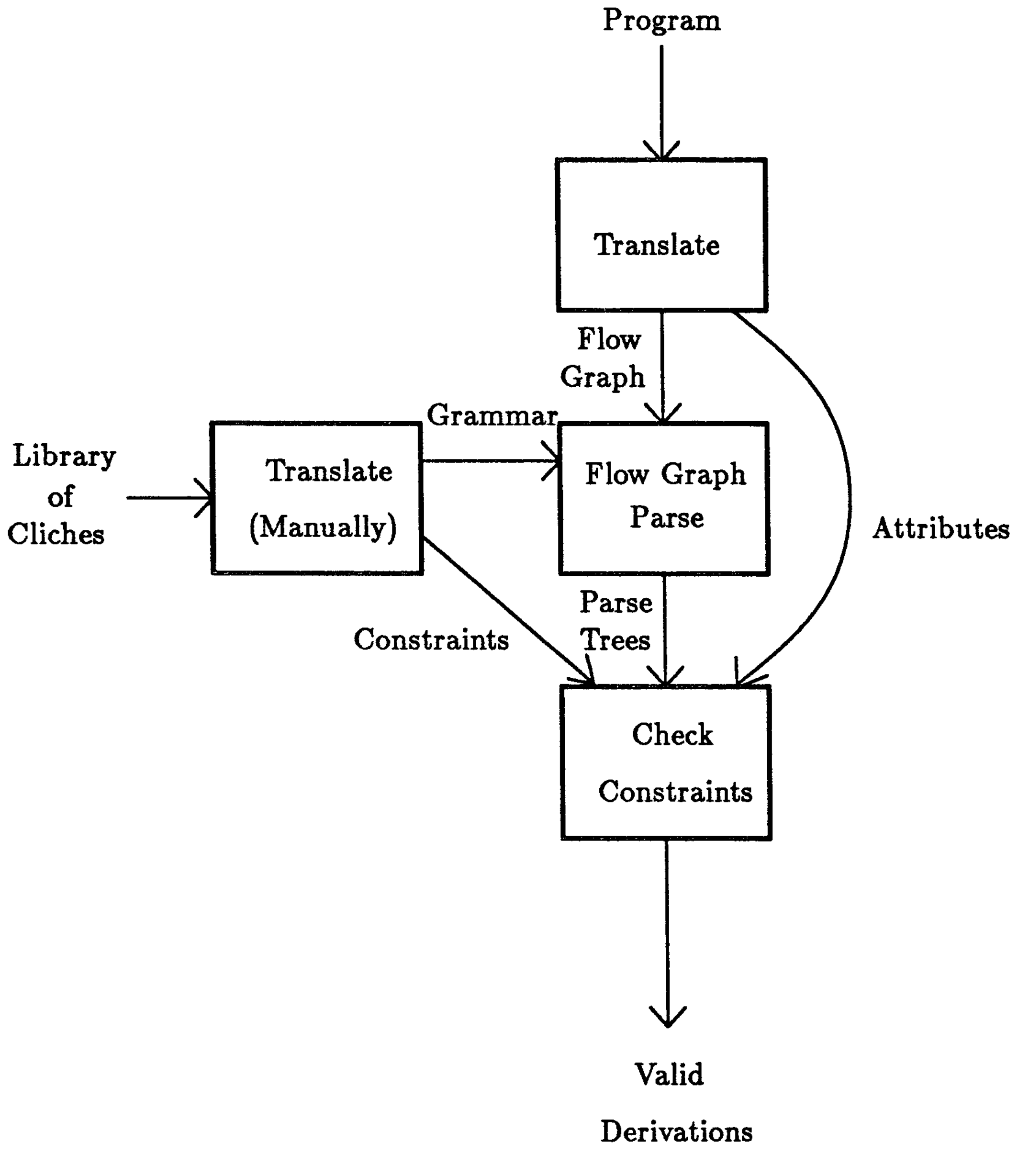


Figure 7: The Structure of the Recognizer

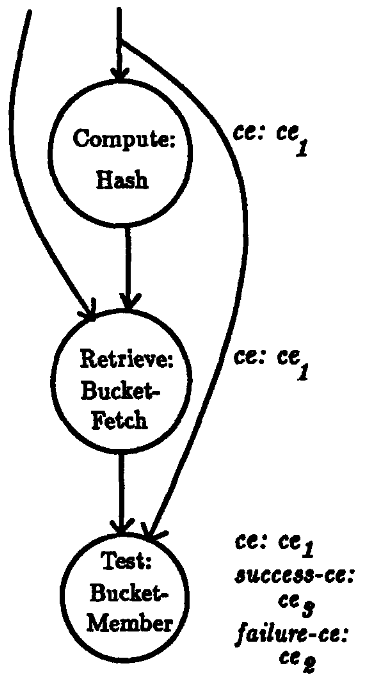


Figure 8: Attributed flow graph corresponding to the Hash-Table-Member plan diagram of Figure 5. (All nodes are annotated with control environment attributes in the form of subscripted "*ce*"s.)

a data and control flow analysis to translate the program into a plan, which is encoded as an attributed *flow graph* [4, 69]. (A flow graph is a labeled, directed, acyclic graph whose nodes have disjoint sets of input and output ports.) For example, the plan diagram of Figure 5 is represented as the attributed flow graph of Figure 8. Each node in the plan diagram becomes a node with distinct input and output ports for each argument and result (respectively) of the node's operation. Each data flow arc becomes an edge in the flow graph. Control flow information is represented by *control environment* attributes on the nodes. A control environment groups together operations that are performed under the same conditions. Control environments form a partial order; a control environment, ce_i, is less than or equal to ($\sqsubseteq$) another control environment, ce_j, iff the conditions under which the operations in ce_j are performed subsume the conditions under which those in ce_i are performed. (In other words, if the operations in ce_i are performed, then so are those in ce_j, but not vice versa.) (For more details and descriptions of other attributes placed on the flow graph, refer to [69, 70].)

The cliché library is likewise translated[1] into an attribute graph grammar as follows:

[1]Translation is currently done by hand (once for the entire library). There is no inherent reason for

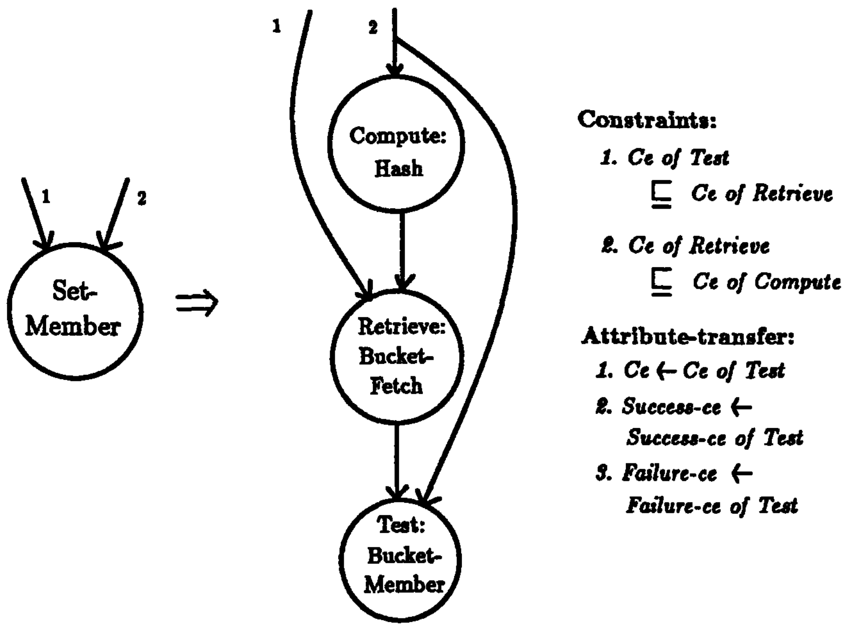

1
2
Set-Member
⇒
1
2
Compute: Hash
Retrieve: Bucket-Fetch
Test: Bucket-Member
Constraints:
1. Ce of Test ⊑ Ce of Retrieve
2. Ce of Retrieve ⊑ Ce of Compute
Attribute-transfer:
1. Ce ← Ce of Test
2. Success-ce ← Success-ce of Test
3. Failure-ce ← Failure-ce of Test

Figure 9: Rule corresponding to the implementation of Set Member using the Hash-Table-Member cliché (specified by the overlay of Figure 6).

A nonterminal is created for each cliché. Overlays which capture implementation relationships are translated into rules that specify how to replace the nonterminal representing some abstract cliché with another graph representing its implementation. For example, the overlay of Figure 6, which specifies that Hash-Table-Member can be viewed as an implementation of Set Member, is translated into the rule shown in Figure 9. When this rule is used in a parse, it uncovers a design decision to implement the abstract set data structure as a hash table, and to implement the set membership operation as a hash table membership. Each grammar rule places constraints on the attributes of any flow graph that matches its right-hand side. The constraint checking phase filters out the parses that do not satisfy the constraints. Although shown in Figure 7 as a separate phase (for clarity), constraint checking is actually folded into the parsing process. That is, constraints are checked each time a nonterminal is reduced, so invalid parses may be cut off early. *Attribute-transfer specifications* are associated with each rule, which compute attribute values for the left-hand side nonterminal based on the rule's right-hand side. This allows attributes to be propagated up through the derivation tree. For details of the meaning of the constraints and attribute-transfer specifications used by the rule in Figure 9, see [69, 70].

this and in the future, this step will be automated.

The flow graph representation of the program is parsed in accordance with these graph grammar rules, using a parsing algorithm developed by Brotsky [4]. This algorithm generalizes Earley's string parsing algorithm to flow graphs. This generates a hierarchical description of the program's design in the form of a derivation (e.g., Figure 4). An important feature of the parsing algorithm is that it is able to generate derivations that are not strictly hierarchical. This allows the Recognizer to recover the design of a program even when parts of the implementation of two distinct abstract operations overlap. That is, it is able to "undo" certain types of optimizations.

Clichés must be recognized even when they are in the midst of unfamiliar code. In order to perform this partial recognition of programs, the parsing algorithm has been extended to "skip over" unrecognizable sections of the input graph. It does this by systematically ignoring each part of the graph in turn and parsing the rest. Although Brotsky's basic parsing algorithm is polynomial in the size of the input, in the worst case, partial recognition using the extended parser takes exponential time. This is not surprising, since determining whether a flow graph is contained in another flow graph is a special case of the NP-complete subgraph isomorphism problem. However, our experience so far has been that the computational expense is greatly reduced because constraints on attributes prune the search performed by the parser. Dynamic programming is also used to decrease the cost. A major goal of my proposed research is to explore the computational limits of this process, both empirically and theoretically.

3 Pushing the Limits of Program Recognition

Of the existing program recognition systems (i.e., the Recognizer and those described in Section 5), I am choosing to extend the Recognizer because it deals with all of the problems listed in Section 2 and it uses an entirely algorithmic and exhaustive process. It does not use any knowledge of the purpose of the program or any heuristics which could cause a cliché to be missed. This makes it ideal for studying purely code-driven program recognition.

My goal is to scale the Recognizer up to perform completely automatic recognition of programs of realistic size (on the order of thousands of lines) and complexity. The Recognizer will be extended to handle more types of programming language features, in particular, recursion, aggregate data structures, and side effects to mutable data structures. The library of clichés will likewise be expanded. The Recognizer will also be extended to deal with incomplete programs (i.e., designs that are still under development) and rapidly evolving programs (which will require incremental analysis techniques to avoid recomputation).

A number of the extensions will require that the Recognizer have logical inferencing capabilities. These will be provided by a hybrid reasoning system, called CAKE [15, 16, 50]. This section will first briefly describe CAKE and how it is expected to be used by the system. It will then describe the extensions that will be made to the Recognizer and to the cliché library.

3.1 CAKE

CAKE is a hybrid automated reasoning and knowledge representation system which supports both efficient special-purpose techniques and limited general-purpose logical reasoning techniques. Among the special-purpose techniques in CAKE is a facility for reasoning about equality. This will be useful in modeling side-effects to mutable data structures and aliasing (i.e., sharing of structure). (See Section 3.2.) In addition, CAKE supports truth-maintenance. This will be convenient, among other things, in performing incremental analysis of evolving programs. (See Section 3.5.)

CAKE will play a major role in the constraint checking phase of the recognition process. Currently, the Recognizer checks constraints by applying procedural predicates to the attributes of graphs which have been successfully parsed. Constraint checking will be made more powerful by using CAKE. Attributes will be treated as propositions in a first order propositional logic. Constraints will be treated as theorems which need to be proved, given these propositions. Constraint checking will then be performed by logical inferencing.

3.2 Types of Program Features

The Recognizer is currently able to recognize clichés in programs that contain nested expressions, conditionals, iteration (expressed as tail-recursion), and some data structures. It must be extended to handle the following features.

- **Recursion** – Recursive calls cause a parsing problem because they typically do not have the same arity or the same node label as the corresponding recursive call in a cliché. This problem was handled for tail-recursive programs by removing the tail-recursive call from the recursive program's graph and capturing the information it contained in attributes. This works because no computations occur after a tail-recursive call; the removal of the tail-recursive call from the structure of the graph leaves just the body of the recursive program, which can be parsed normally. The same solution cannot be employed for general recursive programs because computations are performed after the recursive call, using the results of the recursion. This means that removing the recursive call leaves a "hole" in the program's graph, making it no longer a flow graph and thus unparsable by the flow graph parser.

- **Data Abstraction** – Aggregate data structures are represented in the Plan Calculus using *data plans*. These are not currently used by the Recognizer. Associated with each data plan is a set of input/output specifications which represent the constructors, selectors, and alterants of the data structure. When these standard *accessors* are recognized in a program, the Recognizer should recognize the use of the data structure.

- **Side Effects to Mutable Data Structures** – Modeling side effects involves representing mutable data structures and the modifications made to them. In the Plan Calculus, the standard alterants associated with a mutable structure's data plan destructively modify the structure. A significant portion of the information specifying the modifications is embodied in logical annotations on the plan diagram (e.g., preconditions and postconditions on the input/output specifications). Thus, the Recognizer will need to make use of CAKE's reasoning capabilities in recognizing these data plans. A major difficulty in dealing with mutable data structures is computing the net data flow involved, particularly when aliasing (i.e., sharing of structure) occurs. CAKE's equality reasoning is expected to be useful here.

3.3 Cliché Library

Part of what makes automated program recognition interesting as an Artificial Intelligence problem is that it is an ideal task for studying how programming knowledge and experience can be represented and used. An initial library of programming clichés has been collected and formalized by Rich [48, 51]. The Recognizer currently uses a subset of these clichés. Rich's entire library will be used once the Recognizer is able to handle the programming language features described above. This library will also be expanded to include the common algorithms and data structures described in introductory computer science texts, such as [2, 25, 54].

3.4 Incomplete Programs

Recognition is to be applied not only to code that is an end-product of the software development process, but also to incomplete programs (i.e., designs that are still under development). It should be able to handle programs that contain specifications (expressed in the Plan Calculus) which are not yet implemented. In terms of parsing, these programs may be seen as partially derived in that they contain specifications for unimplemented abstract clichés.

3.5 Rapidly Evolving Programs

The Recognizer needs to be able to deal with incremental changes to a program without recomputing its entire analysis each time. This involves keeping track of dependencies of parses (even those that fail) on how the graph is structured, as well as dependencies between the attributes on the graph and the constraints on them. When the graph or the attributes change, the dependencies indicate what needs to be updated. CAKE's truth-maintenance system will be very useful in recording these dependencies and making the appropriate retractions.

3.6 Choice of Target Programs

In extending the Recognizer to handle realistic programs, it is helpful to have a target application program to drive the extensions. For this purpose, I will be concentrating on an existing architectural simulator for a parallel message-passing machine [7] which contains the programming language features discussed above and is on the order of a few thousand lines. The benefit of using this program is that it was developed independent of this research and is not contrived for the proposed study.

Ideally, in order to study the limits of the technique (including its extensions), it would be best to choose a collection of example programs that range along some spectrum of difficulty of understanding. However, this type of classification of programs is not available at this point. Rather, it will be a product of this thesis. A key part of this research will be the search for programs which stretch the limits of the recognition technique. Of particular interest will be those which exceed the technique's capabilities. These will be studied to characterize what makes them difficult or costly to understand using recognition.

4 Complexity Analysis

In determining the limits of program recognition, it is important not only to find out how much can be recovered from the code alone, but also to understand the underlying complexity of this recovery. Toward this goal, I will perform both an empirical analysis (by metering the code) and a formal complexity analysis of the recognition algorithm. From this, I hope to find out what is dominating the process and what are the typical sizes of the factors involved.

The results of this analysis will suggest how higher-level guidance can be applied to improve the efficiency. For example, if the complexity of the recognition process is dominated by the size of the grammar, then future research should focus on finding ways to narrow down the set of clichés for which to search at any particular time. The complexity analysis may also suggest opportunities for parallel processing. (A small investigation has already started in this direction [52].)

Another result of this analysis may be to show how to control the amount of computational effort the Recognizer is allowed to spend. It may be possible for a control mechanism (or a person) using the Recognizer to incrementally invest an increasing amount of effort in the recognition process, stopping when "enough" information has been extracted. For instance, some constraints on a cliché may be very expensive to check. The controlling mechanism may opt to check only the less expensive constraints when a shallow understanding is satisfactory.

This notion of trying to control the Recognizer's expense is analogous to Hall's [19] notion of "parameterizing" a propositional reasoning system. The idea is to design a limited reasoner with a built-in "power dial" that controls the computational expense allotted to the reasoner. (An example is a resolution theorem prover that is limited to resolutions involving at least one clause of length less than or equal to a parameter CLEN.) As in parameterized reasoners, the potential for controlling the expense depends on the ability to obtain useful intermediate results from the recognition process when it uses up its allowance. All of the criteria Hall defines for usefulness of a parameter also apply to a controlling parameter of the Recognizer. That is, it must be possible to finely control the expense, the power of the system must monotonically increase as more expense is invested in it, and the power must be as large as possible relative to the expense, especially for smaller investments of expense (since, otherwise, it wouldn't be worth limiting the reasoning). Hall measures the power of a parameterized reasoner in terms of the number of clauses deduced by the prover. The power of a recognition system is the depth of understanding it achieves. This is much harder to measure. As a result of the complexity analysis, I hope to find an appropriate parameter and to define a notion of depth of understanding which together satisfy the criteria above.

If such a "power dial" were designed, it could be adjusted based on who is interacting with the recognition system and the task application. For example, a real-time response may be required of the system if a person were using it interactively as an assistant in maintaining code. In this situation, obtaining quick, shallow results may be more desirable than waiting a long time for a deeper understanding. Alternatively, there may be less stringent time constraints. For example, an automated debugging system may be using the system and may not require real-time interaction. The recognition process may be allotted much more expense to gain a much deeper understanding of the code.

5 Related Program Recognition Work

The feasibility of automated program recognition has been shown by a number of other researchers. Their prototypes are distinguished from the Recognizer by the representations of programs and the recognition techniques used. Both affect how well these systems can deal with the fundamental obstacles to program recognition listed in Section 2. Another difference between these feasibility studies and the proposed research is that no analysis of the complexity of the existing systems has been attempted.

5.1 Representation

Johnson's PROUST [22, 21], Ruth's system [53], Lukey's PUDSY [36], and Looi's APROPOS2 [35] operate directly on the program text. This limits the variability and complexity of the structures that can be recognized, because these systems must wrestle directly with syntactic variations, performing source-to-source transformations to twist the code into a recognizable form. Most of these systems' effort is expended trying to canonicalize the syntax of the program, rather than concentrating on its semantic content. In addition, diffuse clichés pose a serious problem.

Adam and Laurent's LAURA [1] represents programs as graphs, thereby allowing some syntactic variability. However, the graph representation differs from the Plan Calculus in that data flow is represented implicitly in the graph structure. That is, nodes represent assignments, tests, and input/output statements, rather than simply operations, and arcs represent only control flow. Because of this, LAURA must rely on the use of program transformations to "standardize" the data flow.

The system proposed by Fickas and Brooks [17] uses a plan-like notation, called *program building blocks* (pbbs), for clichés. Each pbb specifies inputs, outputs, post-conditions, and pre-conditions. The structure of the library is provided by *implementation plans*, which are like implementation overlays in the Plan Calculus. They decompose non-primitive pbbs into smaller pbbs, linked by data flow and purpose descriptions. However, on the lowest level of their library (unlike that of the Recognizer), the pbbs are mapped to language-specific code fragments which are matched directly against the program text. Thus, this system also falls prey to the syntactic variation problem.

Murray's Talus [41, 42] uses an abstract frame representation (called an *E-frame*) for programs. The slots of an E-frame contain information about the program, including the type of recursion used, the termination criteria, and the data types of the inputs and outputs. This representation helps abstract away from the syntactic code structure by extracting semantic features from the program, allowing greater syntactic variability. However, listing all characteristics of the code in E-frame slots becomes very cumbersome and verbose as programs become more complex. This limits the extensibility of the system. This representation also fails to expose constraints (such as data flow constraints) in a way that facilitates recognition.

Letovsky's Cognitive Program Understander (CPU) [32, 31] uses a lambda calculus representation for programs. CPU is somewhat similar to LAURA in that it uses transformations to standardize (i.e., make more canonical) the program's syntax and to simplify expressions. However, Letovsky generalizes canonicalization to be the entire means of program recognition. Canonicalization involves not only standardizing the syntax of the program, but also standardizing the expression of standard plans (i.e., clichés) in the program. Recognizing a plan that achieves a particular goal is equivalent to canonicalizing the plan expression to the goal. So, CPU uses a single, general transformation mechanism for dealing with syntactic variability and for recognition. In contrast, the Recognizer uses a special purpose mechanism (the flow analyzer) to factor out most of the syntactic variability before recognition is attempted.

In order for CPU to localize clichés in a lambda expression so that a transformation rule can apply, numerous transformations need to be made to copy subexpressions and move them around the program. For example, function-inside-if ([32], p. 109) copies functional applications to all branches of a conditional and stored expressions are copied to replace each corresponding variable reference. This is expensive both in the time it takes to apply transformations and in the exponential space blow-up that occurs as a result. In the Plan Calculus, clichés are localized in the connectivity of the data and control flow graphs. In addition, the ability of the parser to generate multiple analyses enables the Recognizer to recognize two cliches whose implementations overlap without first copying the parts that are shared (as CPU needs to).

Another difference arising from the use of the lambda calculus formalism is in the types of clichés that can be expressed. Clichés expressed in the lambda calculus must be connected in terms of dataflow; a cliché containing two or more components with no dataflow interaction between them cannot be expressed. CPU's assumption is that clichés are tied together by dataflow, otherwise there is nothing bringing the results together. (One exception to this is a data abstraction plan in which the tupling operation is used to bind together multiple dataflows into a single value.) In the Plan Calculus, clichés are more general and can contain any number of disconnected subclichés (which may, for example, be tied together by control flow, rather than dataflow).

There is also a difference between CPU's transformations and the Plan Calculus's overlays. Simple transformations are very similar to overlays, but complex transformations often specify procedurally how to change the program. For example, the loop analysis transformation is a procedure. Loop cliches, such as filtering out certain elements from a list which is being enumerated, are transformed using a recursion elimination technique in which the patterns of dataflow in a loop are analyzed and classified as stream expressions. Then, based on dataflow dependencies, occurrences of primitive loop plans are identified and composed to represent the loop. (This is Waters' temporal abstraction technique [65, 66].) Overlays, on the other hand, are declarative. They can be used in both synthesis and analysis. Another difference is that while overlays are used only to represent implementation and specialization relationships between cliches, transformations are additionally used to reduce syntactic variability, simplify dataflow and arithmetic expressions, and perform beta-reduction.

Laubsch and Eisenstadt [28, 29] and Lutz [37] use variations of the Plan Calculus. Laubsch and Eisenstadt's system differs from the Recognizer in the recognition technique it employs. Lutz takes the same parsing approach as the Recognizer. (Both of these systems will be described in the next section.)

5.2 Recognition Techniques

Besides representational differences, the Recognizer differs from other current approaches in its technique for performing recognition. While most basically use a parsing approach, they differ from the Recognizer mainly in the type of heuristics they use in narrowing down the search for clichés, and the amount of knowledge about the purpose of the program they receive as input to help guide the search.

The heuristics used by recognition systems can be classified as being either *conservative* or *daring*. Conservative heuristics are algorithmically complete. They narrow down the search without making any assumptions which may cause some cliché to be missed. The Recognizer uses these types of heuristics in the form of constraints. Daring heuristics, on the other hand, boldly prune the search in a way that is not guaranteed to be complete. For example, to focus their search, Lukey's PUDSY [36] and Fickas and Brooks' system [17] use hints, such as mnemonic variable names typically used by instances of a particular cliché, or distinctive features, such as the interchanging of two array elements which suggests that a sort is being performed. Daring heuristics have advantages with regard to efficiency, since they tend to narrow down the search more than conservative heuristics. However, approaches (such as the Recognizer's) that use conservative heuristics are exhaustive and accurate.

Aside from the type of heuristics used, current recognition systems differ from the Recognizer in the amount of information about the program they receive as input. In contrast to the Recognizer's purely code-driven technique, most recognition systems use some type of description of the purpose of the program to generate expectations about its design. These expectations are then checked against the code.

Johnson's PROUST [22, 21] is a system which analyzes and debugs PASCAL programs written by novice programmers. It takes as input a description of the goals of the program and knowledge about how goals can be decomposed into subgoals as well as the relationships between goals and the computational patterns (clichés) that achieve them. Based on this information, PROUST searches the space of goal decompositions, using daring heuristics to prune the search. (For example, it uses heuristics about what goals and patterns are likely to occur together.) PROUST looks up the typical patterns which implement the goals and tries to recognize at least one in the code. The low level patterns which actually implement the goals are then found by simple pattern matching.

Ruth's system [53], like PROUST, is given a program to analyze and a description of the task that the program is supposed to perform. The system matches the code against several implementation patterns (clichés) that the system knows about for performing the task. Ruth's approach is similar to the Recognizer's in that the system uses a grammar to describe a class of programs and then tries to parse programs using that grammar. The differences are that Ruth's system makes use of knowledge about the purpose of the program (in the form of a task description) to narrow down its search and the program is analyzed in its textual form and is therefore parsed as a string. Another difference is that Ruth's system does no partial recognition. The entire program must be matched to an algorithm implementation pattern for the analysis to work.

Lukey's Program Understanding and Debugging System (PUDSY) [36] also takes as input information about the purpose of the program it is analyzing, in the form of a *program specification*, which describes the effects of the program. This description is not used, however, in guiding the search for clichés. Rather, PUDSY analyzes the program and then compares the results of the analysis to the program specification. Any discrepancy is pointed out as a bug. The analysis proceeds as follows. PUDSY first uses heuristics to segment the program into *chunks* which are manageable units of code (e.g., a loop is a chunk). It then describes the flow of information (or interface) between the chunks by generating assertions about the values of the output variables of each chunk. These assertions are generated by recognizing familiar patterns of statements (called *schema*), similar to the Recognizer's clichés, in the chunks (using daring heuristics, as mentioned above). Associated with each schemata are assertions describing their known effects on the values of variables involved. For chunks that have not been recognized, assertions are generated by symbolic evaluation.

Adam and Laurent's LAURA [1] receives information about the program to be analyzed and debugged in the form of a "model program", which correctly performs the task that the program to be analyzed is supposed to accomplish. LAURA then compares the graphs of the two programs and treats any mismatches as bugs. Since nodes are really statements of the program, the graph matching is essentially statement-to-statement matching. The system works best for statements that are algebraic expressions because they can be normalized by unifying variable names, reducing sums and products, and canonicalizing their order. The system heuristically applies graph canonicalizing transformations to try to make the program graph better match the model graph. It can find low-level and localized bugs by identifying slight deviations of the program graph from the model graph.

The system proposed by Fickas and Brooks' [17] starts with a high-level cliché describing abstractly the purpose of the program. From this, it hypothesizes refinements and decompositions to subclichés, based on its implementation plans (analogous to overlays in the Plan Calculus). These hypotheses are verified by matching the code fragments of the clichés on the lowest level of the library with the code. As mentioned earlier, the system uses daring heuristics to find the hypothesized clichés. While a hypothesis is being verified, other outstanding clues (called *beacons*) may be found which suggest the existence of other clichés. This leads to the creation, modification, and refinement of other hypotheses about the code.

Murray's Talus system [41, 42] is given a student program to be analyzed and debugged as well as a description of the task the program is supposed to perform. It has a collection of reference programs which perform various tasks that may be assigned to the student. The task description is used to narrow down the reference programs that need to be searched to find one that best matches the student's possibly buggy program. Heuristic and formal methods are interleaved in Talus's control structure. Symbolic evaluation and case analysis methods detect bugs by pointing out mismatches between the reference program and the student's program. Heuristics are then used to form conjectures about where bugs are located. Theorem proving is used to verify or reject these conjectures. The virtue of this approach is that heuristics are used to pinpoint relatively small parts of the program where some (expensive) formal method (such as theorem proving) may be applied effectively. However, the success of the system depends heavily on the heuristics that identify the algorithm, find localized dissimilarities between the reference program and the student's program, and map the student's variables to reference variables.

Looi's APROPOS2 [35] uses a technique very close to Talus's. It matches a Prolog program against a set of possible algorithms for a particular task. Like Talus, it applies a heuristic best-first search of the algorithm space to find the best fit to the code.

Many of the systems that receive information about the program's purpose along with its code have been developed in the context of intelligent tutoring systems for teaching programming skills. In this domain, the purpose of the program being analyzed is very well-defined. It can be used to provide reliable guidance to the program recognition process. However, in many other task applications, especially software maintenance, information about the purpose of the program and its design is rarely complete, accurate, or detailed enough to rely on wholeheartedly for guidance. On the other hand, existing partial knowledge about a program might provide useful suggestions to the recognition process if they are taken with a grain of salt. My proposed research will investigate how this partial knowledge of a program's purpose can be used to guide a code-driven process.

There are a few other recognition techniques that, like the Recognizer, are purely code-driven. These will be described in the remainder of this section.

Laubsch and Eisenstadt's system [28, 29] distinguishes between two types of clichés: standard (general programming knowledge) and domain-specific. Standard clichés are recognized in the program's plan diagram by nonhierarchical pattern matching (as opposed to parsing). Then the recognized clichés attach effect descriptions to the code in which they are found. Symbolic-evaluation of the program's plan diagram computes the effect-description associated with the entire program. Domain-specific library clichés are recognized by comparing the program's effect description to the effect descriptions of clichés in the library. This transforms the problem of program recognition into the problem of determining the equivalences of formulas. For programs written in a simple language (SOLO) which manipulates a global database, effect-descriptions do not contain quantified expressions. However, in general, proving the equivalence of formulas is extremely hard.

Letovsky's CPU [32] uses a technique called *transformational analysis*. It takes as input a lambda calculus representation of the source code and a collection of correctness-preserving transformations between lambda expressions. Recognition is performed by opportunistically applying the transformations: when an expression matching a standard plan (cliché) is recognized, it is rewritten to an expression of the plan's goal. This is similar to the parsing performed by the Recognizer, except that CPU does not find all possible analyses. Rather, it uses a simple recursive control structure in applying transformations: when more than one standard plan matches a piece of code, an arbitrary choice is made between them. Letovsky defines a well-formedness criterion for the library of clichéd plans which requires that no plan be a generalization of any other plan. Letovsky claims that if the library is well-formed, then this arbitrary choice won't matter, since recognizing one plan will not prevent the recognition of another. This relies on the fact that CPU performs a great deal of copying: if two clichés overlap in a program (e.g., as a result of merging implementations as an optimization), their common subparts are copied so that each cliché can be recognized individually without interfering with the recognition of the other cliché. Unfortunately, this leads to the problem of severe "expression swell."

It is still an open question in which situations it is better (or necessary) to carry along multiple possible analyses (as the Recognizer does) as opposed to committing to one analysis (which is less expensive). The power of multiple analyses may emerge in programs in which there are unrecognizable sections which lead to several ways of partially recognizing the program.

CPU has taken some steps toward dealing with data abstraction and side effects to mutable data structures. This will be very relevant to the proposed research.

Lutz [37] proposes using a program recognition process very similar to the Recognizer's. In addition, the system will use symbolic evaluation to deal with unrecognizable code. Lutz developed a graph parsing algorithm [38, 39] for use in his recognition system. The algorithm is a generalization of a chart parsing algorithm for strings. Brotsky's flow graph parsing algorithm is a special case of Lutz's algorithm.

A novel type of recognition is being pursued by Soni [58, 57] as part of the development of a Maintainer's Assistant. This system will focus on recognizing *guidelines* which constrain the design components of a program and embody global interactions between the components. For example, guidelines express relations between the slots of data structures and constraints on how they may be accessed or updated. This type of recognition is orthogonal to the recognition of clichés that I am studying. Once both types of recognition are developed further, it will be valuable to combine them as complementary processes.

A completely different approach to recognition is being investigated by Biggerstaff [3]. A central part of the recognition system is a rich domain model. This model contains machine-processable forms of design expectations for a particular domain as well as informal semantic concepts. It includes typical module structures and the typical terminology associated with programs in a particular problem domain. The goal of the recognition is to link these conceptual structures to parts of the program, based on the correlation (experientally acquired) between the structures and the mnemonic procedure and variable names used and the words used in the program's comments. A grep-like[2] pattern recognition is performed on the program's text (including its comments) to cluster together parts of the program that are statistically related.

The virtue of this type of recognition is that it quickly directs the user's attention to sections of the program where there may be computational entities related to a particular concept in the domain. While this technique cannot be scaled up to provide a deeper understanding, it provides a way of focusing the search of other more formal and complete recognition approaches, such as the Recognizer's. Like, Soni's recognition, it is orthogonal and complementary to my proposed research.

6 Related Work Using Attributes, Graphs, and Graph Grammars

The concept of an attribute grammar was formalized by Knuth [26] as a way to assign meaning to strings in a context-free language. Since then attribute grammars have been used extensively in pattern recognition (see the overview given by Tsai and Fu [63]), compiler technology [18, 23, 27, 13], the Cornell Program Synthesizer [8, 61], program derivation [45], and test case generation [10].

Graph grammars have also been used widely in automatic circuit understanding [60], pattern analysis, compiler technology, and in software development environments

[2]The Unix tool grep searches files for given regular expressions.

(see [11, 62] for several examples in these areas).

The research most related to the Recognizer involves attribute graph grammars. In pattern recognition, Bunke ([5, 6]) uses attributes to express nonstructural, semantic properties of a pattern and as a way to reduce the complexity of the grammar. Using context-sensitive attribute graph grammars in a generative (rather than parsing) technique, Bunke's system transforms an input image, such as a circuit diagram, into a graph which describes it. Constraints on attributes are expressed in the *applicability predicate* which is associated with each grammar rule. This predicate places conditions on the nodes and edges of the left-hand side of the production rule which must be satisfied in order for it to be replaced with the right-hand side during derivation. The productions in Bunke's system also have functions for synthesizing attributes, as do the rules of the Recognizer.

Farrow, Kennedy, and Zucconi ([14, 24]) present a *semi-structured graph grammar*, which can be used to analyze the control flow structure of a restricted class of programs for the purposes of optimization. The grammar may be used to derive structured programs containing the standard control structures: sequential statements, conditional statements, while-loops, repeat-until loops, and multiple-exit loops. They also provide a parsing algorithm that runs in time linear in the size of the graph. This algorithm is able to take advantage of the simplicity of the grammar which comes from the simplicity of control flow graphs. Kennedy and Zucconi discuss how the semi-structured grammar may be augmented by attributes to yield a number of applications in the area of global flow algorithms and graph-based optimization algorithms.

Engels, et al. [12] and Nagl, et al. [43] present a formalism, based on attribute graph grammars, for specifying graph classes and graph manipulations. They apply software engineering methods (e.g., abstraction, decomposition, and refinement) to writing graph specifications in this formalism. They call this methodology *graph grammar engineering* and apply it to a project for creating a software development environment (SDE) [34, 44]. The grammar that is engineered is not used for parsing. Rather, it is an operational specification of a software system. Attributed digraphs represent objects in the system being specified (the SDE). The rules define the functional behavior of the system by specifying all allowed transformations on these digraphs.

Acknowledgments

I would like to thank my advisor Charles Rich for helping me to define this thesis topic. I am also grateful for useful comments and suggestions from my thesis readers David McAllester, Peter Szolovits, and Richard Waters.

References

[1] A. Adam and J. Laurent. LAURA, A system to debug student programs. *Artificial Intelligence*, 15:75–122, 1980.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

[3] T. Biggerstaff. Design recovery for maintenance and reuse. Technical Report STP-378-88, MCC, November 1988.

[4] D. Brotsky. An algorithm for parsing flow graphs. Technical Report 704, MIT Artificial Intelligence Lab., March 1984. Master's thesis.

[5] H. Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 4(6), November 1982.

[6] H. Bunke. Graph grammars as a generative tool in image understanding. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 8–19. Springer-Verlag, October 1982. Lecture Notes In Computer Science Series, Vol. 153.

[7] W. Dally. The J-Machine: A fine-grain concurrent computer. In *Int. Fed. of Info. Processing Societies*, 1989.

[8] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *8th Annual ACM Symp. on Principles of Prog. Langs.*, pages 105–116, Williamsburg, VA, January 1981.

[9] V. Dhar and M. Jarke. Dependency directed reasoning and learning in systems maintenance support. *IEEE Trans. on Software Engineering*, 14(2):211–227, February 1988.

[10] A. Duncan and J. Hutchison. Using attributed grammars to test designs and implementations. In *5th Int. Conf. on Software Engineering*, pages 170–178, San Diego, CA, March 1981.

[11] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors. *Graph-Grammars and Their Application to Computer Science*. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[12] G. Engels, C. Lewerentz, and W. Schafer. Graph grammar engineering: A software specification method. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 186–201. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[13] R. Farrow. Experience with an attribute grammar-based compiler. In *9th Annual ACM Symp. on Principles of Prog. Langs.*, pages 95–107, Albuquerque, NM, January 1982.

[14] R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program data flow analysis. In *Proc. 17th Annual IEEE Symposium on Foundations of Computer Science*, Houston, Texas, 1976.

[15] Y. A. Feldman and C. Rich. Bread, Frappe, and Cake: The gourmet's guide to automated deduction. In *Proc. 5th Israeli Symp. on Artificial Intelligence*, Tel Aviv, Israel, December 1988.

[16] Y. A. Feldman and C. Rich. Principles of knowledge representation and reasoning in the FRAPPE system. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, Detroit, Michigan, August 1989. Submitted.

[17] S. F. Fickas and R. Brooks. Recognition in a program understanding system. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pages 266–268, Tokyo, Japan, August 1979.

[18] H. Ganzinger, R. Giegerich, M. Ulrich, and W. Reinhard. A truly generative semantics-directed compiler generator. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 172–184, 1982.

[19] R. Hall. Parameterizing a propositional reasoner: An empirical study. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, 1989. Submitted.

[20] R. Holt, D. Boehm-Davis, and A. Schultz. Mental representations of programs for student and professional programmers. In G. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*. Ablex Publishing Corp., Norwood, N.J., 1987.

[21] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[22] W. L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Trans. on Software Engineering*, 11(3):267–275, March 1985. Reprinted in

C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[23] U. Kastens, B. Hutt, and E. Zimmermann. GAG: A practical compiler generator. In *Lecture Notes in Computer Science Series.* Springer-Verlag, 1982.

[24] K. Kennedy and L. Zucconi. Applications of a graph grammar for program control flow analysis. In *4th Annual ACM Symp. on Principles of Prog. Langs.*, pages 72–85, Santa Monica, 1977.

[25] D. Knuth. *The Art of Computer Programming.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1973.

[26] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[27] K. Koskimies, K. Raiha, and M. Sarjakoski. Compiler construction using attribute grammars. In *SIGPLAN 82 Symposium on Compiler Construction*, pages 153–159, 1982.

[28] J. Laubsch and M. Eisenstadt. Domain specific debugging aids for novice programmers. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 964–969, Vancouver, British Columbia, Canada, August 1981.

[29] J. Laubsch and M. Eisenstadt. Using temporal abstraction to understand recursive programs involving side effects. In *Proc. 2nd National Conf. on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[30] S. Letovsky. Cognitive processes in program comprehension. In G. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop.* Ablex Publishing Corp., Norwood, N.J., 1987.

[31] S. Letovsky. Program understanding with the lambda calculus. In *Proc. 10th Int. Joint Conf. Artificial Intelligence*, pages 512–514, 1987.

[32] S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD Thesis.

[33] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), May 1986.

[34] C. Lewerentz, M. Nagl, and B. Westfechtel. On integration mechanisms within a graph-based software development environment. In H. Gottler and H.J. Schneider, editors, *Graph-Theoretic Concepts in Computer Science*, pages 217–229. Springer-Verlag, June/July 1987. Lecture Notes In Computer Science Series, Vol. 314.

[35] C.K. Looi. APROPOS2: A program analyser for a Prolog intelligent teaching system. Research paper 377, Dept. of AI, University of Edinburgh, 1988.

[36] F. J. Lukey. Understanding and debugging programs. *Int. Journal of Man-Machine Studies*, 12:189–202, 1980.

[37] R. Lutz. Program debugging by near-miss recognition and symbolic evaluation. Technical Report CSRP.044, Univ. of Sussex, England, 1984.

[38] R. Lutz. Diagram parsing — A new technique for artificial intelligence. Technical Report CSRP.054, Univ. of Sussex, England, 1986.

[39] R. Lutz. Chart parsing of flowgraphs. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, 1989. Submitted.

[40] J. Mostow. Toward better models of the design process. *AI Magazine*, Spring 1985.

[41] W. R. Murray. Heuristic and formal methods in automatic program debugging. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 15–19, Los Angeles, CA, August 1985.

[42] W. R. Murray. Automatic program debugging for intelligent tutoring systems. Technical Report 27, Univ. of Texas at Austin, Computer Science Dept., June 1986.

[43] M. Nagl, G. Engels, R. Gall, and W. Schafer. Software specification by graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 265–287, Haus Ohrbeck, Germany, October 1982. Springer-Verlag. Lecture Notes In Computer Science Series, Vol. 153.

[44] Manfred Nagl. A software development environment based on graph technology. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, pages 458–478. Springer-Verlag, December 1986. Lecture Notes In Computer Science Series, Vol. 291.

[45] R. Nord and F. Pfenning. The Ergo attribute system. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 110–120, Boston, Mass., November 1988.

[46] C. Potts and G. Bruns. Recording the reasons for design decisions. In *10th Int. Conf. on Software Engineering*, pages 418–427, Singapore, April 1988.

[47] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981. Reprinted in M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, Springer Verlag, 1984 and in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[48] C. Rich. Inspection methods in programming. Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD thesis.

[49] C. Rich. Knowledge representation languages and predicate calculus: How to have your cake and eat it too. In *Proc. 2nd National Conf. on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[50] C. Rich. The layered architecture of a system for reasoning about programs. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 540–546, Los Angeles, CA, 1985.

[51] C. Rich. Inspection methods in programming: Clichés and plans. Memo 1005, MIT Artificial Intelligence Lab., December 1987.

[52] P. Ritto. Parallel flow graph matching for automated program recognition. Working Paper 310, MIT Artificial Intelligence Lab., July 1988.

[53] G. R. Ruth. Analysis of algorithm implementations. Technical Report 130, MIT Project Mac, 1974. PhD thesis.

[54] R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

[55] H. E. Shrobe. Dependency directed reasoning for complex program understanding. Technical Report 503, MIT Artificial Intelligence Lab., April 1979. PhD thesis.

[56] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595–609, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[57] D. Soni. Maintenance of large software systems: Treating global interactions. In *Proc. of AAAI Spring Symposium*, March 1989.

[58] D. Soni. A study of data structure cliches for software design and maintenance. Working paper, Siemens Corporation, 1989. in preparation.

[59] J.C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy pascal programs. *Human-Computer Interaction*, 1:163–207, 1985.

[60] T. Tanaka. Parsing circuit topology in a deductive system. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, Los Angeles, CA, 1985.

[61] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Comm. of the ACM*, 24(9):563–573, September 1981.

[62] G. Tinhofer and G. Schmidt, editors. *Graph-Theoretic Concepts in Computer Science.* Springer-Verlag, June 1986. Lecture Notes In Computer Science Series, Vol. 246.

[63] W. Tsai and K. Fu. Attributed grammars – A tool for combining syntactic and statistical approaches to pattern recognition. *IEEE Trans. on Systems, Man and Cybernetics*, 10(12), December 1980.

[64] I. Vessey. Expertise in debugging computer programs: A process analysis. *Int. Journal of Man-Machine Studies*, 23:459–494, 1985.

[65] R. C. Waters. Automatic analysis of the logical structure of programs. Technical Report 492, MIT Artificial Intelligence Lab., December 1978. PhD thesis.

[66] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[67] S. Wiedenbeck. Novice/expert differences in programming skills. *Int. Journal of Man-Machine Studies*, 23:383–390, 1985.

[68] D. Wile. Program developments: Formal explanations of implementations. *Comm. of the ACM*, 26(11), November 1983.

[69] L. M. Wills. Automated program recognition. Technical Report 904, MIT Artificial Intelligence Lab., January 1987. Master's thesis.

[70] L. M. Wills. Automated program recognition. *Artificial Intelligence*, 1988. Submitted.