Abstract Data Structure Recognition

René Dekker Frans Ververs

Department of Technical Mathematics and Informatics
Delft University of Technology
Delft, The Netherlands

Abstract

We present a framework for recognition of data structures in programs, to aid in design recovery. The framework consists of an intermediate representation and a knowledge base containing information about typical implementations of abstract data types. The framework is suited for recognition of data structures combined with their characteristic operations. Abstract data structures can be recognized partially, they can be recognized even if they are delocalized, and different independent interpretations of the same structures can be generated.

1 Introduction

Programmers do not create programs from scratch. They have a working knowledge of both programming techniques and techniques related to the application domain they work in [12, 18]. This knowledge results in the use of familiar patterns of usage, called *clichés* [16] or *plans*. The time required to train a programmer in a new application domain is primarily spent on learning the clichés common to that domain.

Experience shows that large programs easily grow out of control. Documentation is lost or grows out of date, and the structure of the program looses its inherent clarity. A programmer who has to maintain such a program, must first understand it. This means that he tries to recognize instances of clichés he is familiar with, and brings these into relation with each other. This work of *design recovery* is a major part of software maintenance [5]. The aim of our project is to automate this part using a knowledge based transformational approach. A recognizer attempts to transform the program into one expressed in terms of clichés, guided by a cliché data base.

Motivation

Design recovery attempts to raise the *abstraction level* of a program. Particular implementations of well-known clichés are replaced by references to the clichés in question. Among the most known and appreciated clichés are *abstract data structures*. For our purposes we define an abstract data structure as the instance of an *abstract data type*, that is, a data structure combined with its characteristic operations. Recognizing such abstract data structures in a program can substantially aid in understanding the program.

The presented system is not limited to complete abstract data structures. Data structure and associated functions will be recognized even if other operations on the structures are unexplained.

General design

The program itself is not transformed. The program text is first translated into a data flow representation. This is done to remove numerous implementation variations that are difficult to account for in the knowledge base. Also, this makes the recognizer independent of a particular programming language.

The clichés in the knowledge base are coded as rewrite rules that map data flow patterns onto other data flow patterns. The recognizer essentially matches parts of the data flow graph against the rules of the knowledge base.

The matching process is done by a chart graph parser similar to the one used by Lunz [13] and Wills [20], and the cliché knowledge base is in the form of a graph grammar. The graph parser is capable of generating independent and partial parses. This has important consequences for the recognition capabilities of the system.

The capability to generate independent parsers allows each piece of source code to be analyzed in different, independent ways. This capability is important for two reasons:

- Different interpretations of the code can coexist. These different interpretations help the maintainer to view the code from different perspectives.
- The same piece of code is often shared between different clichés. An example is a loop that is used for

1068-3062/94 $4.00 © 1994 IEEE

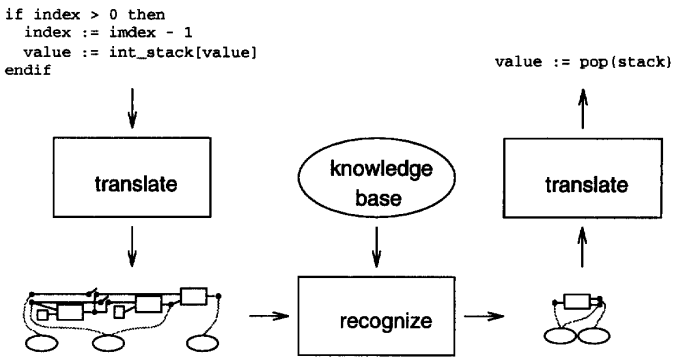

Figure 1: General design of the recognizer

both counting and summing the elements of a list.

The capability to generate partial parses allows the system to partially recognize the program. A data structure can be recognized amidst other, unrecognized code.

The recognized clichés are translated back into program code. This need not be the same programming language as the source code, it can be a completely different language. This opens the possibility to translate from an imperative language to an object-oriented language, for example. The result of recognition need not even be a programming language. It could be a description of the recognized constructs in natural language, there by the recognizer could be used as a documentation tool.

The next section of this paper describes the intermediate representation and how it is constructed from a program. Then the database and its use to recognize abstract data structures are described. We conclude the paper with a look at related work, and our conclusions.

2 The intermediate representation

Design recovery is difficult because of the variation in programs. Programs that solve the same problem can differ completely. Even programs that implement the same solution can differ substantially. Our intermediate representation has been chosen to minimize the variation in intuitive details. That is, two programs that *intuitively* implement the same solution to a problem are ideally represented by the same intermediate representation. When this representation is established, we use the knowledge base to establish *which* solution the programs implement.

Although the use of the term *intuition* makes this guideline understandably subjective, it must be clear that both the clear text program and the abstract syntax tree do not qualify as intermediate representations.

The following variations are tolerated by our representation:

- syntactic variation. Particular ways to express familiar constructs are abstracted away. This gives the representation programming-language independence.
- statement order variation. Only statement order that is important for the computation is preserved in the form of data flow dependence.
- loop variation. All kinds of loops are represented by recursion in the graph.

A familiar intermediate representation used extensively in program optimization is the *program dependency graph* [8]. This graph is a simple data flow graph where the nodes represent statements, and the edges that connect them represent possible data flow between the statements. It includes *control dependency* edges. These edges link each control statement to the statements whose execution is dependent on its condition. The intermediate representation used by Wills [20] can be seen as a variant of the program dependency graph where the control dependency edges are not given explicitly, but are represented by *attributes* of the nodes.

Although adequate for optimization, the program dependency graph is not ideally suited for recognition purposes. Especially the control dependency edges capture too much information that is irrelevant for recognition, and thereby hamper the recognition process. It is more important to know which data flows a control statement influences, then which statements's execution it affects.

Moreover, the program dependency graph requires other, more exotic, edges to adequately describe a program. These include *output* edges that specify definition

```
if index > 0 then
  index := index - 1
  value := int_stack[index]
endif
```

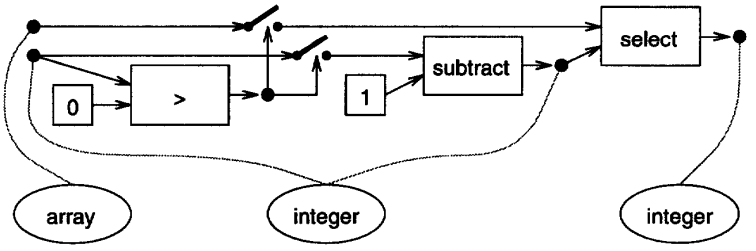

Figure 2: A sample of code with its graph representation

order dependencies between assignments. Such edges are not required in our graph.

Description of the graph

The graph we use consists of two parts. The first part describes the operational structure of the program and is similar to certain data flow graphs from literature [7]. It consists of *nodes*, *tie-points*, and *switches* connected by *edges*. The example in figure 2 shows all of these elements. Nodes are depicted as rectangular boxes, tie-points as little black circles, and the symbol for switches is inspired by the analogy with electric switches. Solid arrows depict the edges that connect the elements.

The second part of the flow graph describes the structure on the data level of the program. It features *containers*, depicted as ellipses. The two parts are linked by *associations* that connect tie-points with containers.

Nodes represent the primitive operations in the program, such as addition of two numbers, or selection from an array. Each node has an arbitrary number of distinct *ports*, divided into *input ports* and *output ports*. To each of the ports an edge may be attached. Nodes are *labeled* with a text string designating the operation represented by the node.

Switches represent the influence of control flow on data flow. They have two input ports—one data port and one switch port—and one (data) output port. Informally, the switch port determines whether or not the data flow between the data input and output can continue.

Tie-points do not have ports; edges are attached to them directly. Tie-points have the ability to *fan-in* and *fan-out*: multiple edges can come together at a tie-point, and multiple edges can originate at tie-points. Tie-points usually represent the assignment to a variable.

Edges are directed and run from output ports to input ports. They can also run to or from tie-points. They are not labeled and serve no other purpose than to connect nodes, switches, and tie-points.

Containers represent the variables in the program. They are connected to tie-points by associations. Each association connects a tie-point with a container. Multiple tie-points can associate to a container, but each tie-point associates to at most one container. Containers are labeled with an identification of the *type* of the variable they represent. This label is likewise called the *type* of the container. The container types are organized in a tree-shaped hierarchy, the purpose of which will be explained below.

Translation from program to graph

Translation from the program into the graph is done with conventional techniques [1]. The program is first translated into an abstract syntax tree. This tree is transformed into a control-flow graph, and finally the data-flow graph is constructed from the control-flow graph. In this last step switches must be introduced where data flow depends on control decisions. The place of these switches is determined using an algorithm similar to Ferrante's technique for determining control dependencies [8].

During graph construction, each basic operation in the program is translated into a node in the graph. Data dependencies between the operations become edges.

Each non-composed variable and each array becomes a container, but records are treated differently.[1] An abstract data-structure need not always be represented by one and only one record in the program. A logical data-structure may constitute only part of a record, or it may be distributed over various records. To cope with this *data structure variation* each field of a record is treated as if it were a separate variable. Thus each record field becomes one container in the flow graph.

Each assignment in the program becomes a tie-point with an association to the appropriate container. Assignments to records become sets of tie-points with appropriate associations. Tie-points are also introduced where fan-out

[1] Besides arrays and records, most programs contain dynamically allocated data structures. These structures are not treated yet, but we anticipate that the framework can be extended to cover common dynamic structures such as trees and linked lists without much difficulty.

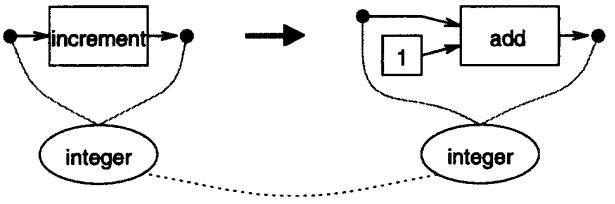

Figure 3: A simple rule

or fan-in is required but no assignment to a variable is present. This occurs if the value of a variable is used, but this value can originate from different operations. In this case a tie-point with an association to the container that represents the concerning variable is generated. Another case occurs if a conditional statement is used to switch multiple data flows. In that case a tie-point without association is generated with edges from the condition and to the various switches.

In order to produce a semantically sound data-flow graph, loops in the control-flow graph must be recognized. We choose to represent these loops as recursion in the data-flow graph. This makes the graph cycle-less, which is a desirable property in matching. We use conventional algorithms to detect *natural loops* in the control-flow graph. This implies that the graph must be *reducible* [2], which poses a restriction on the class of programs that can be treated with our system. It has already been shown however that non-reducible graphs are very rare, even among badly structured programs [1]. If this proves to be too severe a restriction, non-reducible graphs can also be transformed into reducible graphs beforehand [6].

3 Recognition

The knowledge base is constructed as a grammar of graph rewriting rules. Each rule in the grammar specifies how the graph of its left hand side may be rewritten into the graph of its right hand side. The rules in this paper are depicted in compliance with normal practice to specify their direction for generation purposes. As we are concerned with recognition, the rules are applied in reverse order; parts of the graph that match the right hand side of a rule may be replaced by the left hand side.

The nodes in a grammar are divided into terminal and non-terminal nodes. Terminal nodes represent the basic operations; the nodes in the program flow graph must all be terminals. Non-terminal nodes represent abstract operations built with these nodes. A similar distinction between terminal and non-terminal exists in containers.

Certain tie-points of both sides of a rule are designated as possible connections to the environment. They are called *peripheral* points. If a subgraph is replaced by the left hand side of a rule, the peripheral points form the connection to the surroundings of the subgraph.

In contrast to rules in a string grammar, it is not implicitly clear how this connection must be made. Therefore, each rule includes an *embedding* relation. This relation describes which containers of the left hand side correspond to the containers of the right hand side; it is represented by the dotted lines in figure 3. This relation, together with the restrictions on rules specified below gives enough information to determine which peripheral points correspond to each other.

The left and right hand sides of rules in the grammar are both flow graphs. The right hand side of a rule is said to *match* if it corresponds to a subgraph of the program flow graph. A rule *matches* if its right hand side matches.

In our chart parser the concerning subgraph is not actually replaced when a rule matches. Merely a separate view on the graph is created allowing the subgraph to be viewed in two ways: as the original or as the new subgraph. In practice, the parser simulates a *parallel parse* of the program. Separate parses of the same graph exist in parallel, and are extended simultaneously. This allows *sharing* of operations and data structure between different abstractions and can eventually generate multiple interpretations of the same structures or operations.

Correspondence

The right hand side of a rule need not correspond exactly with a subgraph in order to match. Certain variations in the subgraph are allowed. For example, an edge in a rule matches a path of edges, tie-points and switches in the graph. This prevents irrelevant assignments and control flow dependencies form interfering with structure recognition.
The following conditions govern whether a subgraph matches the right hand side of a rule:

- Two nodes match if they have the same label.

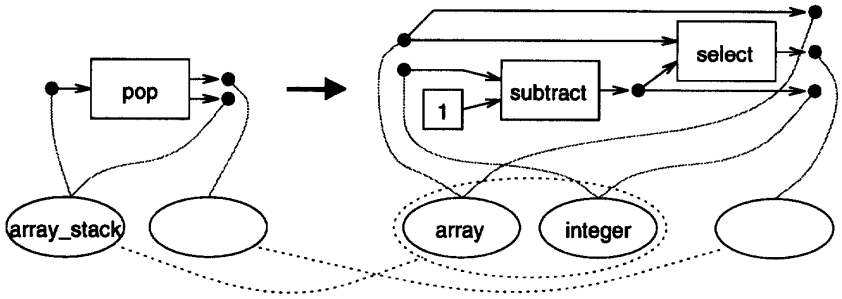

Figure 4: A stack pop recognition rule.

- A container in the right hand side of a rule matches a container in the graph if it has a type that is the same or an ancestor of the type of the container in the graph. They also match if either of them has no type. The hierarchy of container types is explained in the sequel.
- Two tie-points match if they have associations to matching containers, or if either of them has no associations.
- Two switches match.
- Two edges match if they are attached to matching things. That is, if an end-point of an edge is attached to a tie-point, the corresponding end-point of the other edge must be attached to a matching tie-point. If an end-point of an edge is attached to a port on a node or switch, the corresponding end-point of the other edge must be attached to the corresponding port of a matching node or switch.
- An edge in the right hand side of a rule matches with a path of edges, tie-points and switches in the graph if the edge and the path match corresponding to the matching conditions for two edges.
- The right hand side of a rule matches a subgraph if each of its elements matches with an element of the subgraph and vice versa.

Restrictions on rules

The left and right hand sides of a rule are subject to a number of restrictions. These restrictions are intended to make parsing more efficient and allow a simple embedding relation. They do not pose a severe limit on the ability to formulate effective rules for abstract data structure recognition. Due to the restrictions only *non-terminal single node* rules are valid. That is, the left hand side of a rule may contain only one node, which must be non-terminal. This allows the parser to be guided by non-terminals.

To achieve this goal the following restrictions are placed on the left hand side of a rule:

- Only one node may be present, and it must be non-terminal.
- All containers must be non-terminal.
- No switches may be present.
- The peripheral tie-points can be divided into two sets: *input points*, from which edges only originate, and *output points*, to which edges only lead.

Restrictions on the right hand sides are less severe. Only the restriction on the peripheral tie-points is valid for them.

Tupling

The grammar rules as described so far allow a (sub)graph of nodes to be rewritten into one node. In program terms this means that a set of connected operations may be abstracted into one operation. A similar ability can be formulated for data structures: the ability to rewrite a set of related containers into one container. We call this ability *tupling*.

In order to add this ability to our framework, the embedding relation must be extended. Besides relating single containers to other containers, the extended relation can associate single containers with tuples of containers. Figure 4 gives an example. The embedding relation associates the array_stack container on the left hand side with the array and integer containers on the right hand side.

If containers of a rule are typed, they can match only with containers in the graph that are typed identically, or whose types are related in the type hierarchy. But containers in a rule need not be typed. These containers match with any container, tupled or non-tupled, in the graph. This capability allows generic data structures to be recognized. Figure 4 gives an example of a recognition rule for a stack

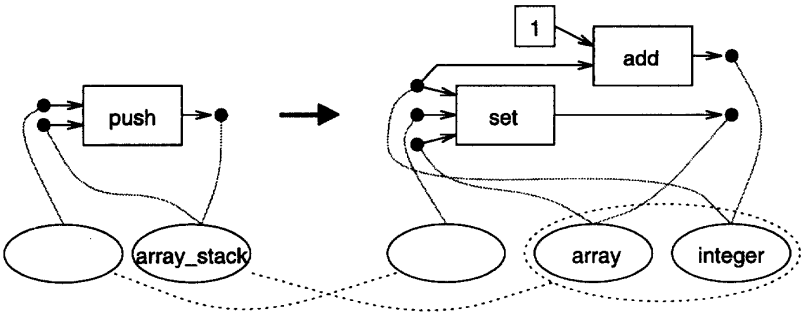

Figure 5: A stack push recognition rule

```
int_stack[index] := 4
index := index + 1

index := index - 1
value := int_stack[index]
```

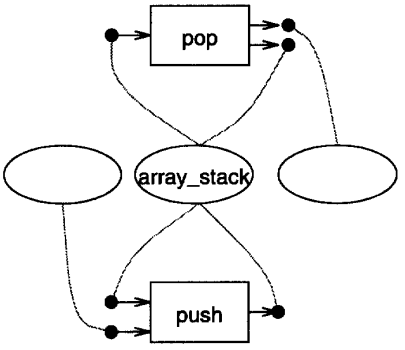

Figure 6: Recognition of a stack abstract data structure

pop operation. The container that represents the element that is popped is untyped. Therefore, this container can match with any container in the graph, whatever its type.

With simple embedding a one-to-one relation exists between the peripheral points of the left and right hand sides. Extended embedding complicates this relation, and therefore it complicates the way a replaced subgraph is attached to its environment. The peripheral points of the replacement can no longer correspond directly with tie-points of the original graph. The relation between the replacement and the original graph is captured in a *correspondence relation*. Due to tupling, each peripheral point of the new subgraph corresponds with a tuple of tie-points in the original graph. This relation may be visualized by special tupling and detupling operations between the tie-points of the original graph and the peripheral points of the subgraph.

Container sharing

Up till now, we have focussed mainly on recognition of operations. Tupling gives a method to build data structures guided by the recognition of *one* of its operations, but it does not relate a *set* of operations to the data structure. In order to recognize abstract data structures, we need to have a method to relate various recognized operations to each other and to the recognized data structure. Such a method is *container sharing*.

While recognition is progressing a record is kept of the correspondence of containers to variables. At the start, each container corresponds to one variable, but as more and more tupling rules are applied, more and more containers correspond to tuples of variables.

In principle, each application of a rule generates a new set of containers: the right hand side is conceptually replaced by the left hand side. With container sharing in effect, containers that have the same type and the same correspondence to tuples of variables are not generated separately, but a single instance of such a container is shared. Associations to these containers refer to the shared container.

As an example, given the stack pop rule depicted in figure 4 and the stack push rule depicted in figure 5 the two

code fragments on the left of figure 6 would be recognized as the abstract data structure on the right.

To elaborate, code fragments need not be successive for an abstract data structure to be recognized. They need not even be related by data flow. Merely the sharing of the same tuple of variables and the presence of characteristic operations constitute the presence of an abstract data structure.

The hierarchy of container types

Up till now we have only discussed the recognition of a single implementation of a data structure. Obviously, in order to be able to talk about an *abstract* data structure, we need to be able to recognize various implementations of the same structure. This capability is provided for in a very simple and direct way.

All container types are related to each other in a hierarchy of types. If a type implements an other type then it will be a child in the type-hierarchy. In our example, the array_stack type is an implementation of the stack type and therefore a descendent of stack in the hierarchy. The stack will have other children too, such as linked_list_stack, with its own recognition rules for the pop and push operations.

A data structure that uses a stack in its implementation, can specify the container type stack in the right hand sides of its rules. Such a container will match with any already recognized container that has a child of stack as type. Thus any already recognized implementation of a stack, whatever its form, can serve to recognize that data structure.

4 Related work

The work presented in this paper is highly influenced by and based upon the work of Linda Wills on the Recognizer [17, 20]. The Recognizer is part of the Programmer's Apprentice project that focusses both on recognition and on automatic synthesis of programs [19]. The aim of the Programmer's Apprentice is to create an intelligent assistant in all phases of the programming task. The project uses the Plan Calculus [15] as the intermediate representation, which is a graph with data flow and control dependency edges, akin to the program dependency graph.

Wills uses a representation derived from the Plan Calculus where control dependency edges are converted to attributes. She focusses on recognition of process structure and did not attempt to use this to recognize abstract data structures. Our recognition of operations is essentially the same as Wills's, but we added tie-points, containers and associations, in order to be able to recognize abstract data structures.

Kozaczynski et. al. [10, 11] use an augmented abstract syntax tree as the basis for program concept recognition. Among the augmentations is cross-reference, data flow and control dependency information. The Desire project [4] takes another approach. It uses informal information, such as identifier names, rather than formal information, such as syntax and semantics, to guide the search process. The search is applied to the program text itself.

Proust [9] is a program understanding system aimed at recognition of novice programs. It uses a top-down approach rather than the bottom-up approach of our system. Based on a functional description of the intended program, the system tries to match the functional goals against program pieces. Therefore, it requires the user to specify a top-level description of the program, before recognition determines how this description is implemented in the program.

All the above recognition systems are knowledge based. A non-knowledge based method aimed at recognition of *objects* is presented by Ong and Tsai [14]. They use heuristics, such as similarities in the way parameters of a function are used, to suggest possible data structures and associated functions. They do not recognize what the data structure represents, however, and therefore they give no insight into the role of the structure.

The program dependency graph is a well-known intermediate representation used extensively for both optimization and program slicing. It consists of a set of statement nodes connected by data flow and control dependency edges. As explained in section 2 the program dependency graph is not ideally suited for our purposes.

The data flow part of our graph resembles normal data flow graphs, such as used for data flow programs [7], and sometimes as intermediate representations [3]. The addition of containers and associations is a novel concept, however.

5 Conclusions

The framework presented in this paper is suited for recognition of abstract data structures: data structures combined with their characteristic operations. It accomplishes this by deriving the existence of a data structure from the presence of its operations.

It can recognize abstract data structures even if not all operations on them are explained. This partial recognition capability allows valuable information to be obtained about variables in a program even if the knowledge base falls short of recognizing all uses of the variables.

It must be emphasized that use of our framework generates a number of different and possibly independent views on the variables and operations in a program. In that, it delivers useful insight into possible interpretations of these

constructs, but it does not choose between them. Inevitably, some of these interpretations will be more appropriate than others. A selection criterion to choose between interpretations is not given in this paper.

A selection criterion can also help in enhancing the efficiency of the recognizer. With the aid of a criterion, the chart parser can detect promising possibilities in an early stage and concentrate its efforts on these, while reducing work on other options. Therefore, a selection criterion is a subject of ongoing research. Such a criterion will obviously depend on the percentage of variable uses covered by a certain interpretation and on the amount of operations recognized.

Another point of future research is recognition of dynamic data structures. It is foreseen that simple dynamic structures can be treated without much extension to the framework, but that more complex structures may need new concepts.

The major strength of our framework is its ability to base the search for a data structure on the recognition of its characteristic operations, not on the structure alone. This gives us the ability to recognize a useful and common concept: abstract data structures.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[2] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Austin, Texas, 1983.

[4] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, July 1989.

[5] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[6] J. Cocke and J. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the 2nd Hawaii International Conference on Systems Sciences*, pages 143–146, 1969.

[7] A. L. Davis and R. M. Miller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, Feb. 1982.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] W. L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, Mar. 1985.

[10] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, Dec. 1992.

[11] W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. In *7th KBSE Conference*, Sept. 1992.

[12] S. Letovsky. Cognitive processes in program comprehension. *Systems and Software*, 7:325–339, 1987.

[13] R. Lutz. Chart parsing of flowgraphs. In *Proceedings of the 11th Joint Conference on Artificial Intelligence*, pages 116–121, Detroit, Michigan, 1989.

[14] C. Ong and W. Tsai. Class and object extraction from imperative code. *Journal of Object-oriented Programming*, 6(1):58–68, Mar. 1993.

[15] C. Rich. A formal representation of plans in the programmer's apprentice. In *IJCAI Conference*, volume 7, 1981.

[16] C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM Press Frontier Series. ACM, New York, 1990.

[17] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

[18] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, Sept. 1984.

[19] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, Aug. 1988.

[20] L. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT Artificial Intelligence Laboratory, July 1992. AI Technical Report no. 1358.