Towards an Intelligent Debugging System for Pascal Programs: On the Theory and Algorithms of Plan Recognition in Rich's Plan Calculus

Thesis

Towards An Intelligent Debugging System For Pascal Programs:

On The Theory And Algorithms Of Plan Recognition In Rich's Plan Calculus

Rudi Lutz B.Sc. M.Sc.

Thesis submitted in partial fulfillment of requirements for Ph.D. in Artificial Intelligence, April 1992.
(Revised February 1993)

The Open University
Milton Keynes MK7 6AA
U.K.

Date of submission : 23rd April 1992
Date of award : 15th March 1993

Abstract

This thesis presents work in progress on building an intelligent general purpose, domain-free debugging system for Pascal programs. This system (IDS) is based on Rich's surface plan/plan calculus graph formalisms, and this thesis develops theory and algorithms for performing program understanding within this framework.

This thesis is in three parts. the first part describes why an ability to do both plan recognition and general purpose reasoning is essential for debugging programs.

The second part of this thesis describes the plan calculus and shows how it offers the ability to combine both general purpose reasoning, and (graph-based) plan recognition. It then goes on to present a new polynomial time algorithm (a generalisation of traditional chart parsing for string grammars) for doing bottom-up, or top-down, analysis of surface plans. This algorithm is presented in the framework of restricted structure sharing flowgraph grammars developed for this purpose. To justify the use of such grammars this part of the thesis develops the theory of deterministic operations in the plan calculus, and also develops the theory of generalised control flow environments to justify the way control flow information is handled.

The final part gives an overview of IDS, and modifies the above algorithm to cope with various parts of the plan calculus that do not exactly fit into the framework we have developed. Thus this thesis gives a complete account of how to perform plan recognition in Rich's framework. This part of the thesis also describes the translation process from Pascal to surface plans, and presents a technique for translating plans expressed in Rich's notation, into suitable rules for

the parser to use. Finally it presents some preliminary work on exactly how a debugging system using these techniques might go about locating and fixing bugs in programs.

Table of Contents

Table of Figures

Acknowledgements

This thesis is dedicated to my daughter Ruth, without whom this thesis might have been finished a long time ago, but who has kept me in touch with what is important in life.

I would like to thank:

Marc Eisenstadt and Steve Isard for being willing to help and advise.

Caroline for putting up with me through the bad parts of writing this.

Liz for helping with Ruth when it was necessary.

PART 1
BACKGROUND

Chapter 1.

The Nature of Bugs and of Debugging, and the Need for Program Understanding.

1.1 Introduction and Overview of Thesis.

The research described here is aimed at creating an intelligent system capable of aiding programmers in the task of debugging programs. As computer hardware becomes cheaper the cost of producing and maintaining software is becoming the major factor in most applications. In addition, decreasing costs and increasing power of the hardware means that projects are becoming increasingly more ambitious and complex, and hence more error-prone and indeed there appears to be a "complexity barrier" [Winograd 1973] past which it is very difficult if not impossible to go. It is therefore becoming increasingly necessary to use the computer itself to aid the programmer in producing correct software. One approach to this is the attempt to build fully automated program synthesis systems [Barstow 1979, Manna and Waldinger 1979]. These would only require the programmer to provide a very high-level specification of what the program is to do and the synthesis system would automatically generate a correct program to carry out the programmer's intentions. Although much interesting work has been done on automatic program synthesis it seems unlikely that really usable systems will become available in the near future.

An approach much more likely to lead to a working system in the foreseeable future is that of the "programmer's apprentice" [Hewitt and Smith 1975, Rich and Schrobe 1978, Rich 1981, Waters 1982]. Such a system has a large amount of knowledge about programming. For instance it knows the various standard ways of sorting a list numbers. It also has knowledge about common data structures and implementation techniques etc. The system can then use this knowledge to relieve the programmer of many of the details associated with implementing various algorithms etc. However it would expect the programmer to supply the high-level algorithm to be used. It could also check code given by the programmer for consistency with what it knows.

An intelligent debugging system is more modest in its aims than a full programmer's apprentice which also aims to help programmers with the coding process itself. However it is clear that a debugging system needs to be able to understand and (possibly) suggest edits to a program in exactly the same way as a programmer's apprentice system. Furthermore, such a debugging system would be an essential component of a full programmer's apprentice system, and one can therefore quite easily imagine enlarging such a debugging system to a full programmer's apprentice. Partly because of this, building an intelligent debugging system is clearly a very large and difficult undertaking, and accordingly is as yet unfinished. This thesis will describe work currently in progress on building an intelligent debugging system (IDS) for Pascal programs. This work is heavily based on Rich's [1981] plan calculus which provides the formal underpinnings for the Programmer's Apprentice project [Rich and Schrobe 1978]]. Despite its power and expressivity however, full use of the plan calculus has been frustrated by lack of a proper algorithm for performing plan recognition, although Wills [1986, 1990] has done some work on this. Most of this thesis will be devoted to giving an account of, and justification for, a new algorithm (based on a generalisation of chart parsing applicable to graph-like structures known as flowgraphs) for doing this. The later part of the thesis will make some preliminary suggestions on how the information provided by the plan recognition process can be used to locate (and sometimes repair) bugs in programs. Much of the work described here has been reported elsewhere [Lutz 1984a, 1984b, 1986, 1989a, 1989b, 1991], but this account will provide more details, and place it all in context. The long term aim of this research [Lutz 1984a] is to build a system capable of helping the programmer to locate and fix errors in programs. IDS will ultimately attempt to integrate a whole variety of methods and sources of information (described below) to help it in debugging, rather in the way an expert programmer does. However, the work described here is an initial attempt towards a debugging system using only the basic techniques of plan matching combined with a theorem prover [Lutz 1984b] to arrive at a high-level description of what the program (or part of it) does and how it does it. So far only the plan matching part of this work is complete, but detailed scenarios based on this plan matching will be presented to show how logical inconsistencies (a notion made more precise below) can be found in programs, and how suggestions can be made as to how to correct the code, based on near-misses to known plans. These near-misses are quite explicitly found by our plan recognition algorithm. At the moment no attempt has been made to use information implicit in variable names and so on. Despite this it will be argued that IDS's approach is powerful enough to duplicate many of the capabilities of other debugging systems [e.g. Lukey 1978, Soloway, Ehrlich, Bonar, and Greenspan 1982, Johnson 1986] for syntactically correct programs, except for where these systems make use of problem specific knowledge. More importantly the power and generality of the representation methods used by IDS potentially enable it to deal with programs involving such things as recursive and non-recursive procedure calls, value and reference parameter passing, records and pointers, as well as the full range of data types available in Pascal. Thus IDS can deal with programs involving dynamic data structures implemented using pointers and records, unlike the systems mentioned above. IDS attempts where possible to be language independent so as to facilitate the application of any techniques developed to other programming languages. In doing this it draws heavily on the work of Rich, Schrobe, and Waters [Rich 1981, Rich and Schrobe 1978, Waters 1978, 1979, 1982] whose development of the Programmer's Apprentice project at MIT represents a major attempt at representing the knowledge underlying programming. To a lesser extent it draws on the work of Laubsch and Eisenstadt [1980] whose work on an intelligent debugger for SOLO programs bears some similarities to that described here. Their work has also been significantly influenced by that of Rich, Schrobe, and Waters.

Apart from the practical uses such a system would have, debugging is also an interesting area in its own right for Artificial Intelligence research. Programs are amongst the most complex objects available for study, and raise all kinds of interesting questions about the interplay between knowledge representation and general reasoning abilities. It is the contention of this thesis that in order to successfully reason about programs in order to debug them we need the ability to move smoothly from making use of "compiled" knowledge, expressed as rules capturing an experienced programmer's knowledge of programming techniques and algorithms, to general reasoning from first principles to cope with novel or unexpected features of programs, and we hope to show that the plan calculus as developed by Rich [1981] provides a suitable framework for doing this.

The organisation of the thesis is as follows. The rest of this chapter will discuss the nature of bugs, and of debugging, and will present the case that the ability to recognise occurrences of standard plans (programming clichés) in a program is a prerequisite for the ability to find and repair bugs in it. It will also be argued that the ability to reason in a general way about programming constructs is also vital. Chapter 2 will present a survey of other program understanding and debugging systems. Chapter 3 will give an account of Rich's [1981] work which provides the theoretical underpinnings enabling both plan recognition and general reasoning to be combined. Chapter 4 will extend the theory of the plan calculus in order to justify many of the operations performed by IDS's plan recognition system. Chapter 5 will give an account of the underlying algorithm used for plan recognition by IDS. It introduces the notions of context-free flowgraph grammars (based on Feder's [1971] plex grammars), and context-free structure-sharing flowgraph grammars, which are not only capable of fairly directly expressing many of Rich's ideas, but also could find applicability in other domains (e.g. electronic circuit analysis). It then goes on to describe a new (polynomial time) algorithm for recognising diagrams generated by such grammars. Chapter 6 will give an overview of IDS and describe the translation process from Pascal programs to Rich's surface plan formalism. Chapter 7 will describe the plan recognition module itself, including modifications needed in order to enable the algorithm described in Chapter 5 to cope with those parts of Rich's formalism which do not exactly fit the pure flowgraph formalisms introduced there. It will also describe the conversion of plans as described in Chapter 3 into flowgraph grammar rules suitable for use by the parser. Chapter 8 will give an account of IDS's understanding abilities on various example programs, and Chapter 9 will discuss how its approach to debugging using this mechanism can be used to find and repair simple bugs in programs. Finally Chapter 10 will give our conclusions, and summarise future work that still needs to be done.

1.2 A categorisation of bugs

It is possible to distinguish several different types of error that can occur in programs, and several authors have attempted to categorise these. As these authors all either use different terminology, or use the same terminology to mean different things, and because the motivations behind the different taxonomies are often different from that of this project, a new taxonomy will be presented here. Before doing this, however, a brief description will be given of some of these other taxonomies so that the similarities and differences between them will become more apparent.

The main distinction underlying the different taxonomies is between those motivated by an interest in the psychological processes which cause programmers to make errors, and those motivated by an interest in automatically identifying and repairing the errors, although there is of course some overlap between these. The work of Youngs [1974], and Schneiderman [1980] typifies the psychologically motivated work, while that of Goldstein [1974] and Millar and Goldstein [1977] typifies the second type of work. The work of the Cognition and Programming Project at Yale [Johnson, Soloway, Cutler and Draper 1983, Spohrer, Pope, Lipman, Sack, Freiman, Littman, Johnson, and Soloway 1985] falls into both camps to some extent but, as their work is primarily aimed at novice programmers and deals with the misconceptions novices have, does not fully address the kinds of errors more expert programmers make.

Youngs [1974] classifies programming errors into syntactic, semantic, logical, and clerical errors. Syntactic errors are errors in the use of the syntax of the programming language and are easily detected by compilers. Semantic errors occur when a program requires the computer to do something either impossible or contradictory e.g. read from a closed file. Normally such errors would give rise to run-time error messages. Logical errors occur when the program is a valid program in that it will run to completion without any obvious sign of there having been an error (such as an error message) but the program does not in fact do what it is supposed to. Clerical errors are due to such things as mistyping, or using a text editor carelessly. Clearly, a study of clerical errors is of great interest to someone designing text editors, but is of less interest to an automatic debugging assistant. The distinction between semantic and logical errors could form the basis for a debugging system (e.g. an expert system using run-time error message information) but, as will be seen later, the definition of semantic error can be widened to encompass many of the bugs Youngs would classify as logical, leading to a clearer relationship between the kinds of bug in a program, and the techniques needed in order to debug it.

Schneiderman's [1980] taxonomy depends much more on a psychological model of the programming process. In his model programmers begin by forming a mental model of the problem and its solution (the internal semantics), and then bring general programming knowledge, and knowledge of the specific programming language being used, to convert this internal description into an actual program to solve the problem. Schneiderman also makes the distinction between syntactic and semantic errors, and also assumes that syntactic errors are easily caught by compilers. In his taxonomy semantic errors include all other kinds of bug that can occur in programs, and he subdivides these into two further categories. The first of these corresponds to errors in the conversion process from the internal semantics to the actual program, and the second to an incorrect conversion from the problem to the internal semantics. Schneiderman points out that the first type of error is much more easily debugged than the second, which may involve a complete redesign of the solution, or of the programming strategy.

Goldstein's [1974] bug taxonomy is based on a theory of programming as a planning activity. This views programming as a process of finding a plan (a sequence of steps) which achieves some goal. Each step in the program may in turn have prerequisites that need to be true before the step can be carried out, thus giving rise to subgoals that need to be planned for. Under this view there are several kinds of bugs that can occur. The first of these is what Goldstein terms a linear main-step failure, and occurs when a sub-plan in the program fails to achieve the goals it is supposed to, independently of the other steps in the program. This kind of bug can be fixed by repairing this sub-plan in isolation. The second type of bug is what he terms a preparation error. This corresponds to the situation when the sub-plan concerned does solve its goals under certain assumptions about the program state on entry to the sub-plan, but these assumptions are not necessarily always true. To fix this kind of bug it is necessary to insert a new step which sets up the conditions assumed by the faulty sub-plan. The third kind of bug is what Goldstein terms a non-linear main step failure. This type of bug arises from non-linear interaction between sub-plans, and in general is very hard to fix. Sometimes this kind of bug can be fixed by suitably interleaving the steps of the sub-plans, or by adding extra steps to one or both of the sub-plans which are not strictly necessary to achieve the goals the sub-plan is designed to achieve, but which are there to eliminate the unwanted interaction. In addition Goldstein distinguishes between what he terms theory bugs, and what he terms procedure bugs. Theory bugs correspond to the case where the initial goals as derived from the problem statement are in error, often because of misunderstandings about the problem domain. Procedure bugs are bugs in the implementation of a program to solve the problem. All the above types of bugs are special cases of procedure bugs. To fix theory bugs an automatic system would also have to have knowledge of the domain, not just general programming knowledge.

Miller and Goldstein's [1977] taxonomy has much in common with that just described. They also view the programming process as a planning activity. However, they provide a "planning grammar" which specifies how plans may be combined to provide higher level plans. Given their planning grammar, they then distinguish between syntactic, semantic, and pragmatic bugs. Miller and Goldstein use the term 'syntactic' to describe the case where the basic plan grammar is violated. For instance, the grammar may specify that a particular plan needs a particular type of sub-plan at a particular place in a plan, and if this is missing then there is a syntactic error. Note that this is not the same as an error in the use of the syntax of a particular programming language. It really falls into one of the categories that Youngs or Schneiderman would term semantic or logical. Semantic bugs, according to Goldstein and Miller, occur when a syntactically (in their sense) optional component is missing from a plan, but is required by the nature of the problem the program is attempting to solve. These bugs fall roughly under Youngs' notion of logical error since they give rise to a fully functioning program which simply does not do what it is supposed to. Finally, pragmatic bugs occur when an inappropriate choice (with respect to the problem) of plan has been made. Again, this type of bug falls under Youngs' notion of logical error.

Johnson et al.'s [1983] and Spohrer et al.'s [1985] bug classification depends both on a library of stereotypical programming plans, and on the problem the program is trying to solve. Given these two dimensions they have developed a large catalogue of several hundred different types of bug, ranging from the fact that a particular plan needed to solve a particular problem is missing, to a description of all the different ways novices might mis-implement a specific plan. Because of the detailed nature of this catalogue they can give very specific advice to novice programmers on how to fix the bug, or on what misconception about looping constructs the novice is suffering from. However, following this particular route seems unrealistic for expert programmers given the likely combinatorial explosion in bug types. However, as part of a tutoring system, their approach works well, as evidenced by the success of Johnson's system PROUST [1986].

This project is using the taxonomy shown in Figure 1.1. The simplest distinction it makes is that between syntactic and semantic errors. Syntactic errors are easily caught by compilers, and this project will assume that the programs it is looking at are syntactically correct. In the case of novice programmers, all kinds of much deeper misconceptions may manifest themselves as syntactic errors, but as this research is aimed at more experienced programmers, it is assumed that error messages from the compiler will be sufficient to enable the programmer to fix this type of error.

Semantic errors correspond to the case where the program in some sense does not "do what it is supposed to". To arrive at this taxonomy it has been assumed that programmers, when confronted with a problem, come up with some sort of solution, or high-level design, corresponding to Schneiderman's internal semantics. Now there are two cases - either their design is correct, or it is not (this latter case corresponds to Goldstein's theory bug category, and to Schneiderman's category of bugs arising from an incorrect conversion to the internal semantics). In either case they now attempt to implement their design

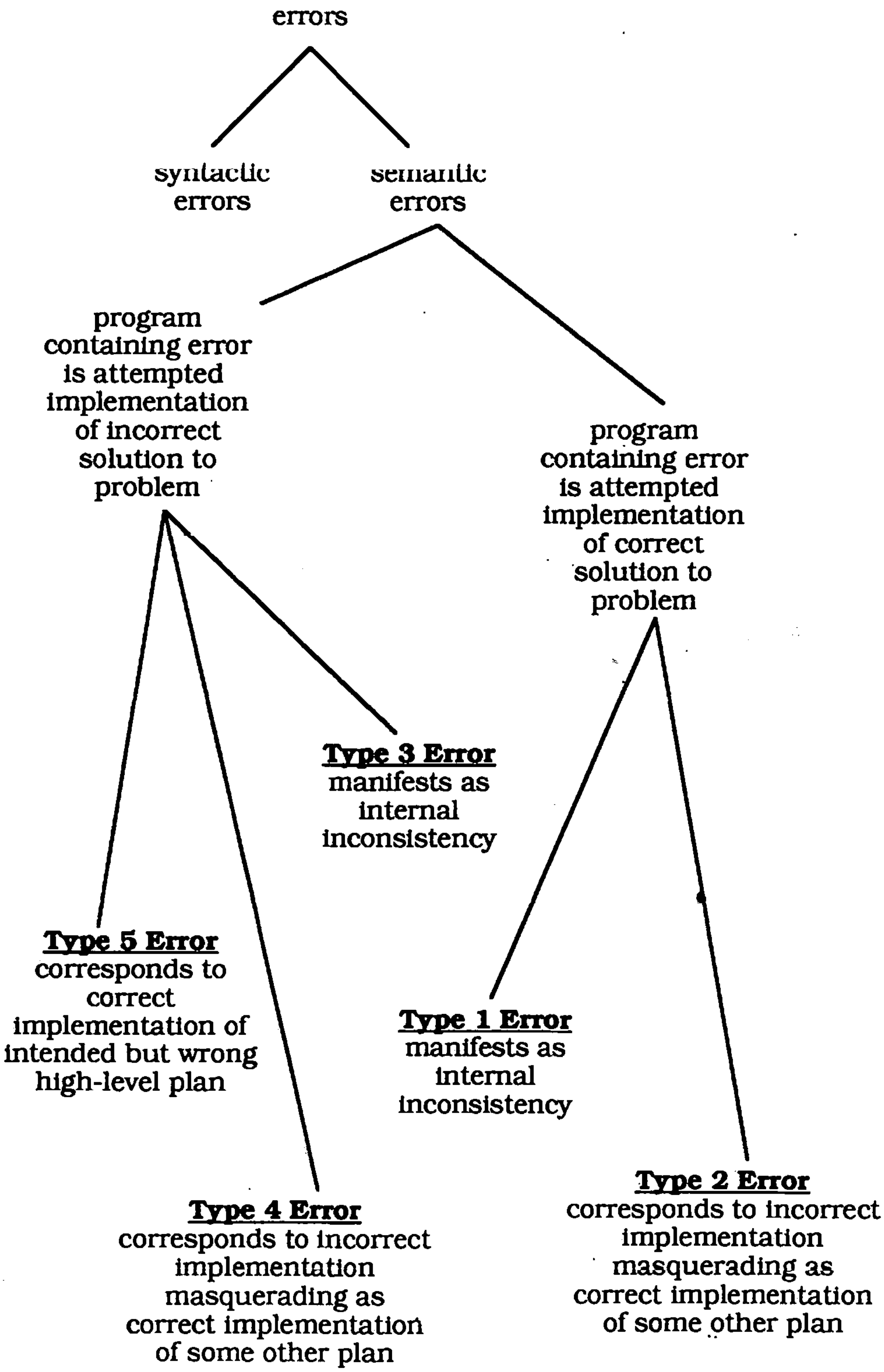

errors
syntactic errors
semantic errors
program containing error is attempted implementation of incorrect solution to problem
program containing error is attempted implementation of correct solution to problem
Type 3 Error manifests as internal inconsistency
Type 5 Error corresponds to correct implementation of intended but wrong high-level plan
Type 1 Error manifests as internal inconsistency
Type 4 Error corresponds to incorrect implementation masquerading as correct implementation of some other plan
Type 2 Error corresponds to incorrect implementation masquerading as correct implementation of some other plan

Figure 1.1 A taxonomy of Errors

as a program. In the case where their design is indeed correct but they mis-implement it (corresponding to Goldstein's procedure bug category, and to Schneiderman's class of errors which arise from an incorrect transformation from the internal semantics), there are two types of error that can occur:

Type 1. These errors manifest themselves as internal inconsistencies of one kind or another between parts of the program. Simple examples are trying to read from a closed device, or passing an unsorted list through to some piece of code implementing an algorithm which expects a sorted list as input. This type of bug includes Youngs' semantic errors as special cases, but widens the class of inconsistencies considered. As will be seen later, because this project is based on a library of programming "cliches", this category also corresponds to Miller and Goldstein's syntactic error category, since the plan library can be viewed as a grammar for correct programs, and the grammar does not permit this kind of inconsistency.

Type 2. Programs containing (only) this type of error manifest themselves as correct (internally consistent) programs for some other problem. An example might be if the programmer was trying to write some code to sort some numbers into ascending order, but used "<" instead of ">" so that their program sorted into descending order. Their code would be a perfectly valid solution to a different problem from the one they were trying to solve. Note that so far as an automatic debugging system is concerned there is no difference between the case where this happened because of what Youngs terms a clerical error, and the case where this happened because the programmer misunderstood the effect of the particular sorting plan being used.

Miller and Goldstein's semantic and pragmatic bugs both fall into this category.

On the other hand, if the programmer's high-level plan does not in fact solve the problem, there are again several ways in which errors can manifest themselves:

Type 3. These errors also manifest themselves as internal inconsistencies in the program. The programmer is trying to implement her design (even though it won't turn out to be the solution to the problem) and has made similar kinds of errors to those of type 1. Thus the design contains a theory bug, but Type 1 procedure bugs have also been made. An example might be if the programmer has decided that one way of accurately calculating the square root of a number x is to find the largest integer whose square is less than x, and the smallest integer whose square is bigger than x, and then do linear interpolation. This clearly is not a correct plan for computing square roots. Now suppose the programmer goes on to implement their plan, but makes an error in the code which looks for the two appropriate integers, which manifests itself as an inconsistency between various sub parts of this code. This is similar to a type 1 bug, but even if corrected the program would still not be correct so far as the programmer was concerned. In fact the corrected program would now contain a type 5 bug (see below).

Type 4. These errors are similar to those of type 2. The programmer has inadvertently managed to implement a different plan from the one intended. In this case the design contains a theory bug, but Type 2 procedure bugs have also been made.

Type 5. This corresponds to the case where programmers actually implement their proposed (faulty) solution correctly. In such a program all parts of the program do exactly what the programmer expects, but still does not solve the problem. This corresponds to a pure theory bug in Goldstein's terminology.

Notice that these error types are in increasing order of difficulty so far as locating and correcting them is concerned. Type 1 and type 3 errors are in some sense errors whatever the purpose of the program. Because of this it seems reasonable to try and at least locate these types of errors completely automatically. If it is assumed that the programs being debugged are not by complete novices, then it is likely that the program will be a reasonable attempt at an implementation of the programmer's intentions, and hence if a program contains only type 1 errors then it can quite possibly be debugged into a correct program automatically. At worst automatic debugging may turn the program into one with a type 2 error (which may or may not have been there initially). Programs with type 2 errors can be debugged by interacting with the programmer. This is because the programmer knows the intended high-level plan and hence should be able to give information about what various parts of the program should be doing. In contrast, although type 3 errors can also be automatically detected, removal will leave a program with type 4 or type 5 errors. Again the type 4 errors may be "corrected" by interacting with the programmer, but only to a program with a type 5 error. Type 5 errors can be very hard to fix, even for skilled programmers. This is because the program will not do what it is supposed to, in the sense that it doesn't solve the original problem, but all parts of the program do what the programmer expects since the program is a correct implementation of the design. Essentially fixing this type of bug can involve a complete redesign of all or part of the program.

1.3 Sources of Knowledge for Program Understanding and Debugging.

In order to debug a computer program it is necessary to develop an understanding of the program and to use this understanding to pinpoint the places where what the program actually does differs from what one would expect knowing the purpose of the program. It is helpful to think of an intelligent debugging system as an experienced programmer whose advice may be sought when one is faced with a bug one does not understand in a program. Such an expert programmer will use several techniques and sources of information to help him/her understand and debug the program. These include the following:

(1) **plan recognition** - this is the technique where by recognising the form of all or part of a program one can recognise what it does i.e. this is "programming cliche" recognition.

(2) **symbolic evaluation of, and general reasoning about, a piece of code** - this involves actually analysing code to see what it does, and arriving at a description (either formal or informal) of its effects. This technique is often used on novel code i.e. code that does not fit any cliche known to the programmer. This situation can occur for two reasons - either the program writer has thought of a new technique (relative to the techniques captured by the cliches known to the debugging programmer) to achieve some standard operation, or they are implementing code corresponding to no known operation but which is essential to achieve some precondition of the code which follows. In the first case, the debugger can compare the effect description she has built up of the code with the effects of known plans. If these are equivalent then effectively the debugger can treat the code as if a known plan had been used in the code instead of the unusual code. In this case this can then feed into the process of plan

recognition in the code. The second case is more problematical, in that the only description of the unusual code is the result of the symbolic evaluation. In this case the best that can be done is to try and show that the effect description does indeed achieve some precondition of the plan(s) it feeds into, in which case the role of this code in the program has been established. This will almost certainly involve some kind of general reasoning. Indeed, even in the case where two cliches (one of which feeds into the other) have been recognised, but which do not jointly constitute a 'higher-level' cliche, then again some kind of general reasoning will be necessary to show consistency, and establish the roles, of the two plans.

(3) **reasoning backwards** from the manifestation of an error to its source.

In applying these techniques the programming expert will use information from the following sources:

(i) interaction with the program's author.

(ii) meaningful variable names.

(iii) comments in the program.

(iv) program segmentation into functions and procedures.

(v) data input to and output from the program.

(vi) run-time error messages.

(vii) trace output.

Plan recognition and symbolic evaluation are used by the expert to gain an understanding of the program. This means that s/he would then be able to:

(a) describe the program in terms of some high-level plan of which the program is an implementation.

(b) describe how the various parts of the program are implementations of sub-plans.

(c) describe how these sub-plans interact to achieve the overall goal(plan) of the program.

Conversely if a programmer can do these three things one would then say s/he has an understanding of the program, and it is in this sense that the term "understanding" will be used from now on.

It is clear from this definition that the process of plan recognition is really the key to program understanding. Now, when an expert looks at a piece of code which, say, sorts a set of numbers, s/he does not at random try plans s/he knows until one is found which fits. The process of plan recognition is clearly guided by clues, and these clues come from items (i) to (iv) in the above list of of sources of information used by expert programmers. In particular things like the use of a procedure name "sort" should suggest that as a first attempt one should look at all the methods one knows for sorting. Obviously, program comments, information supplied by the program writer as answers to questions about the program, and the way the program is segmented can all provide useful information to guide the process of plan recognition.

However it is possible that some pieces of code might not match any plans known to the expert. In this case the expert would almost certainly try a process of symbolic evaluation (probably described by the expert by some term such as "mental evaluation") to try and see what that piece of code does. It may then become apparent that the piece of code does perform some known function, albeit in a different fashion to previously encountered methods. One can then say that the expert programmer has again recognised the plan for which this code was an implementation, the only difference being that this time the plan was recognised by reasoning about the code rather than just matching against a library of previously known plans.

When debugging programs there are essentially two processes used by expert programmers. The first of these is really an application of the process of program understanding described above. If, while attempting to understand the program, the expert finds examples of code which almost match known plans then s/he has almost certainly located an error, particularly if this near-match occurs where there is some inconsistency in the use of known plans. In this case edits can be suggested which would correct the plan. Alternatively a piece of code may be recognised as the implementation of some plan but this plan does not achieve what the program's author says it should. In this case code which implements the desired plan can possibly be suggested. At the very least the mismatch can be pointed out. This debugging method can be used to locate errors of types 1 (and 3) in programs.

The other debugging method uses the remaining sources of information available to the programmer i.e. the data input to, and output from, the program, system generated run-time error messages, and trace information. The above debugging method using plan recognition alone is really only used by programmers on smallish programs. When debugging larger and more complex programs the advice-giving expert will use the input/output data and trace information together with information from the program's author and/or the run-time error messages to locate the places in the code an error first became apparent. S/he would then attempt to "reason backwards" from this place to the place in the code which originally caused the error. This "reasoning backwards" process is really one of isolating all and only that code which could have affected the place where the error became apparent [Weiser 1982], and one can then reason about this subsection of the code using all the techniques of plan recognition etc. Expert programmers will often assume parts of the code to be correct and reason about the rest of the code using these assumptions. Only if the error can't be found, or if some piece of evidence leads to a contradiction will these parts of the code be examined in detail. In addition certain run-time error messages will suggest very specific errors to expert programmers, and they may well just search the code looking for a very specific piece or type of code which their experience tells them is often the cause of the particular run-time error in question.

In the long term it is intended that IDS should try to use all these techniques and sources of information in order to try and repair bugs of types 1 to 4 in programs. However, for the moment, effort has entirely been concentrated on the plan recognition process, as this not only provides much of the machinery needed to apply other techniques as well, but also provides much of the information that other debugging techniques would need. Things like use of variable names etc. could provide useful input to the plan recogniser to guide the search for plans, but this is really an efficiency issue, although of course inappropriate naming of variables and procedures could be considered a type of bug if looked at from a maintenance point of view. Plan recognition (combined with theorem proving and/or symbolic evaluation) really just gives one a handle on type 1 (and type 3) bugs, and it is on these that the rest of this thesis will concentrate. This approach should therefore perhaps be called *intention-free* debugging, and is based on the following assumption (which will be referred to as the **Normal Use Heuristic**):

> If a programmer uses a standard plan in a program then, as a first hypothesis, assume they are using it deliberately in order to achieve the results of operations commonly implemented by that plan, and that therefore the preconditions for these operations should be met.

This assumption is needed because, generally speaking, programs in themselves are not inconsistent, or incorrect. Programs are only incorrect with respect to the programmer's intentions, and if these are not available then showing a program to be incorrect is not possible. So, without a specification the best we can hope to do is to find internal inconsistencies in the code (type 1 errors). Although it is possible for a program to be internally inconsistent in such a way that simple reasoning about the source code might detect, this is not generally the case. An example where this might be true is in the case of a program which attempts to take the square root of a negative number (in a programming language without complex numbers). However, the program discussed in Chapter 9 with its pointer reversal bug, is not inconsistent in any direct sense. It will run, and will produce output, and conceivably could be what the programmer had intended, especially if all we were shown was its input-output behaviour. In order to find this bug by inspecting the code we need the Normal Use Heuristic, since it is only under this assumption that we can in general meaningfully say that plans and operations in a program are inconsistent with each other. Note that if we include single operations as plans in the above assumption then this covers things like the above sqrt bug as well. Without this heuristic, plan recognition, even combined with a plan based theorem prover, would not in general be able to find type 1 bugs in programs, since all we would be able to deduce is that the programmer had used some standard plans in a non-standard manner for reasons best known to themselves. This is of course possible since the programmer might for example have been trying to see what error message the system gives when an attempt is made to take the square root of a negative number, or they might deliberately have written some non-terminating piece of code in order to test something out repeatedly, knowing full well that they would need to interrupt the code externally (using <CTRL>-C say) in order to terminate it. It is not really reasonable to expect a debugging system to cope with this type of (normally temporary) code, and we will assume that the above assumption is reasonable for code submitted to IDS.

Chapter 2.

Other Work on Program Understanding and Debugging

By now a large number of systems have been proposed and/or built which perform program understanding and/or debugging. These systems vary greatly in the knowledge representation techniques that they use, their range of applicability, the languages they are intended for, the type of user they are aimed at, and so on. This chapter will discuss these systems, paying particular attention to the following questions:

(1) Is the system intended for use by novices or experts (or both)?

(2) How are programs represented internally by the system?

(3) How is programming knowledge represented?

(4) Does the system use general reasoning techniques such as a theorem prover or symbolic evaluation?

(5) How does the system locate errors?

(6) What strategies does the system use to repair errors once found?

(7) Does the system make use of a bug catalogue, or other library of common errors?

(8) What range of programs in the language and/or domain can the system cope with?

(9) What are the particular strengths and weaknesses of the system?

Having described all these systems we will then be in a position to draw some general conclusions about work on intelligent debugging and understanding, and this will enable a proper evaluation of IDS's approach. This will pave the way for showing (in Chapter 9) how some of these other systems' capabilities could be reproduced in a system using IDS's approach, and what may need adding to it to duplicate those that can't.

There are several main dimensions along which the systems described below can be classified. The first of these is essentially to do with generality i.e. what range of programs can the system deal with? This issue is related to how much knowledge the systems have to have *in advance* about the problem that the program it is being asked to debug is supposed to solve. Some systems only work on a very narrow range (two or three in some cases) of problems for which the system has extremely detailed knowledge of possible solution strategies (e.g. Ruth [1973, 1976], Johnson[1986]). Other systems only work for problems for which the system already has an example solution (e.g. Adams and Laurent [1980], Murray [1986]). On the other hand there are systems which attempt to be more general. These essentially fall into three categories - those that require a specification of some type to be submitted along with the program (e.g. Lukey's [1978] system PUDSY, and Goldstein's [1974] system MYCROFT), those that simply take a program and attempt to debug it as best they can (e.g. Wertz [1987], and IDS itself), and those that take a program and an example of its intended behaviour (an input/output pair) and then attempt to debug it (e.g. Shapiro's [1982] Prolog debugger).

The next factor which can be used to differentiate these systems is to do with their use of plan recognition versus their use of general reasoning abilities (e.g. reasoning about the semantics of programs using a theorem prover, symbolic evaluation). Of the systems that work on known problems the two most successful fall at opposite ends of this division. PROUST [Johnson, 1986] uses plan recognition alone, while TALUS [Murray, 1986] uses theorem proving alone. As discussed below, both of these systems present problems with respect to trying to increase their domain of applicability. One of the contentions of this thesis is that in order to widen the range of programs to which a debugging system can be applied the system will need to use both of these techniques. Lukey's system PUDSY [1978] was an early attempt to use both approaches. However, because its plan recognition technique was rather simplistic, it had to fall back on symbolic evaluation much of the time, and this rather limits a system's ability (in particular to cope with larger programs due to problems of scaling).

The systems which make use of theorem proving/symbolic evaluation can be subdivided into two further categories - those which try to show that a program meets a specification (e.g. PUDSY, MYCROFT, and Eisenstadt and Laubsch's [1981] SOLO debugger), and those which attempt to show equivalence of a program to a standard reference program (e.g. LAURA [Adams and Laurent, 1980], and TALUS[Murray, 1986]).

The plan recognition based systems can also be subdivided into two categories - those which essentially use top-down recognition (analysis by synthesis) (e.g. PROUST, Ruth's system [1973]), and those which use bottom-up plan recognition (e.g. IDS itself, Wills' system [1986, 1990], and PUDSY).

Another aspect of these systems which can be used to differentiate them is in the techniques they use for suggesting repairs to programs. In particular, do they use general techniques, or do they look for specific errors, and make use of "canned repairs" via the use of a "bug library".

2.1 MYCROFT

One of the earliest attempts at a debugging system was that of Goldstein [1974]. This system, MYCROFT, was aimed at debugging simple LOGO programs which drew pictures. The input to the system consisted of a program and a detailed description of the picture it was supposed to draw. MYCROFT generated a description (plan) of what the program actually did, and by comparing this with the input description of the desired picture, attempted to eliminate bugs. Although an interesting proposal MYCROFT was never fully implemented, but it did at least point the way for later work in that it attempted to formulate the debugging process as one of plan recognition and repair.

2.2 Ruth's Work

An early precursor of the work reported here is that of Ruth [1974, 1976]. His system, implemented in LISP and CONNIVER, was capable of recognising simple programs as being implementations of certain algorithms to perform known tasks. As this system was intended for use by novices in a teaching environment the problems the programs were supposed to solve were known in advance, and the teacher would supply details of common solutions *and all their variants* to the system. These would be expressed in a very simple programming language containing assignments, conditionals, and loops, with numbers and arrays as the only data types. These algorithm schema can be thought of as constituting a plan library. Student programs are translated into the same simple language, and Ruth's system would then attempt to match a student's program against its internal knowledge of possible solutions (PGMs). Before passing the student program through for full analysis a pre-pass would be made to catch any “trivial” errors. These include such things as parameterless loops without an exit, use of uninitialised variables, and irrational code such as assignments to a variable followed by another assignment to the same variable before the first value has been used, or unreachable code. These are not repaired, merely reported. Once a program passes this initial phase, the main analysis uses a process of "reverse synthesis". This process is analogous to that of a parser in a compiler, which recognises programs as syntactically valid. In this case possible parses are attempted in a top-down best-first fashion. The system attempts to match a program against the PGMs, at any stage the PGM which matches best being chosen as the candidate for further matching. Each PGM consists of an ordered list of actions (assignments, conditionals, or loops) which the student program should perform. In order to capture the possible variability in the way each of these actions could be implemented, Ruth's system makes use of symbolic evaluation, non-linear programming techniques, and program transformations in its matching process. For example, if the PGM contains a sequence of assignment statements, these are symbolically evaluated to give an expression E for the value of each of the variables. If the student's program also contains a sequence of assignments at this point, these too are symbolically evaluated to give an expression E′. After making appropriate substitutions for variables in E of variables that occur in E′, the expression E-E′ is then passed over to the MACSYMA system for simplification. If it simplifies to zero the expressions are equivalent so the match is successful. To match conditionals the system first tries to match the tests in the two conditionals. Since these are all of the form exp1<exp2, or exp1=exp2, etc. and the system has the results of the symbolic evaluation for the variables prior to the test being carried out, the system uses nonlinear programming to determine if the tests are equivalent or not. This is not as general as a theorem prover, so won't

always work, but is easier and faster. In matching loops, the matcher has knowledge of various program transformations that preserve the functionality of loops, and can use these while attempting to match. Failures to match are then taken as indications of errors, which can be of two main types - recoverable errors, and non-recoverable errors. Non-recoverable errors are taken as a sign that matching against another PGM should be resumed. Recoverable errors are those that the repair of which would not necessitate major changes to the source code. In this case, a note is made of the error, and the matching process continues as if the error had not been found. At the end of this process the PGM which matched with the least errors (or the least serious errors) is treated as representing the programmer's intentions, and a report is issued spelling out what algorithm the system thinks the student was attempting, what the errors were, and how they can be repaired. However, it should be stressed that Ruth's system was by no means general purpose - it has to know which problem the student is working on in order to narrow down the number of applicable PGMs. Apart from this Ruth's system suffered from many other limitations. Some of these are related to the poverty of the programming language used. More serious though is the linear nature of the PGMs. At their top-level they consist of a sequence of actions to be carried out. The system is not capable of realising that the order of many of these is often unimportant (an exception is in the case of a sequence of assignments). The PGMs should be structured more as a partial ordering. Furthermore, to be a general program understanding system, there would have to be more information about the commonalities between various algorithms, not simply information about the commonalities between variants of algorithms for performing the same task. The system would also need to know about data structuring techniques and so on. If PGMs had these extra capabilities they would probably be much more like the plan diagram formalism used by IDS, and in this sense Ruth's system can be seen as an early attempt at a system rather like IDS.

2.3 LAURA

Another system aimed at debugging student programs is LAURA, written by Adam and Laurent [1980]. Their system for debugging simple FORTRAN programs is in some ways similar to Ruth's system in that again the problems that will be presented to the system are known in advance. However, rather than having someone (the teacher) attempt to provide details of all known solutions (and their variants) for each task as in Ruth's system, LAURA just requires a single correct program for each task. This is then represented internally by means of an annotated control flow graph. The input student program is also represented in this form. It then applies various program transformation techniques to either or both of these graphs in an attempt to recognise the student's program as functionally equivalent to the known correct solution. Should the system find that it cannot transform the two graphs into each other it then used the mismatches to point out errors. This system too performed reasonably well on a few limited examples. However, control flow graphs are not really expressive enough to make reasoning about programs, and recognising plans easy. As a result, their system was more like a system for proving program equivalence, than like one which attempts to understand the program in the sense described in Chapter 1.

2.4 PROUST

Of more immediate concern to this project is the work of the Cognition and Programming Project group at Yale University [Ehrlich and Soloway 1982, Soloway, Bonar, Woolf, Barth, Rubin, and Ehrlich

1981, Soloway, Ehrlich, and Bonar 1982, Soloway, Bonar, Ehrlich, and Greenspan 1982, Soloway et al. 1982, Johnson 1986]. This group is also attempting to create a debugging system for Pascal programs, and like the systems discussed above is aimed at novice programmers. Much of their work is done from the point of view of cognitive psychology since they are interested in finding out exactly what knowledge and understanding expert programmers have that novices don't. They are also building into the system a cognitive model of the student so that helpful advice pin-pointing exactly the student's misconceptions can be given. The knowledge of their system is represented internally in a frame-like formalism (similar to that in [Goldstein and Roberts 1977]) orientated very specifically towards Pascal. Their system PROUST [Johnson 1986], (and its predecessor MENO-II) had a large amount of problem-specific knowledge enabling it to perform reasonably on programs for a few very specific problems. This knowledge is plan based, but the plans are expressed as program schemas (with extra information about preconditions, and about what *goals* the plan can be used to achieve), so the plan matching process is essentially one of matching against the source code. This representation of plans has three main drawbacks:

(i) It makes it difficult to apply to languages other than Pascal, since all the plans would need to translated. This translation may well prove difficult since much Pascal specific knowledge is embedded in them.

(ii) It makes it hard to *verify* plans as reasoning about code fragments is hard, especially as no attempt has been made to really give a semantics to the representation. Of course this could be done, but would require a lot of extra machinery (perhaps borrowing from one or more of the existing methods of describing the semantics of programming languages e.g. denotational semantics, or operational semantics).

(iii) It makes it hard to reason about novel parts of the code that do not match known plans. Of course, if the plans were given a semantics as just suggested, then whatever underlying programming language semantics one had used could also be used to reason about novel pieces of code. In this case the PROUST approach would turn into something rather like a language specific version of the plan calculus.

(iv) The approach is prone to what Wills [1991] calls the 'syntactic variability' problem. This is that, once programs start getting more complicated, there are often a very large number of ways in which the syntax of the language will allow the programmer to express a computation. Some of the differences are very superficial e.g. use of different variable names, differing order of statements in cases when the order does not matter, breaking up expressions into sequences of assignments, and so on. However, when combined, these superficial differences can make it very hard for a system based on code template matching to actually find appropriate matches. The solution in PROUST is to have large numbers of templates that attempt to capture all the likely variants. This is only really possible by limiting the system to a few smallish problems.

Because of the lack of semantics of PROUST's representation methods, it is not really possible for it to recognise inconsistencies in arbitrary programs. Instead, the system has to know in advance what goals have to be achieved by the program (and how these goals inter-relate), and can then check that the program contains plans which achieve all these goals. If a program fails to contain a plan to achieve some goal then a bug has been found. In order to give debugging advice

PROUST then makes use of a very large *bug library*. This contains commonly occurring buggy versions of plans, also expressed in the same code template form. If any of these occur in a buggy program, then very specific debugging advice (stored with the bug rules in the library) can be given. This approach enables PROUST to give astonishingly pertinent advice even for some of the truly bizarre errors that novice programmers make. However the price paid is high - the bug catalogue is very large. Even though PROUST can only deal really effectively with one simple programming problem (involving not much more than computing the average of the non-negative numbers read in from a file or terminal), and rather less effectively on a few other simple problems), the PROUST group have catalogued something like 800 different bugs that novices make [Johnson et al. 1983, Johnson and Soloway 1985]. This explosion of bug types, and hence in the size of the bug library, make it seem unlikely that this approach can be extended to much larger and more complex programs, let alone arbitrary programs. So, although PROUST is extremely successful at tutoring students on the few problems for which it has a goal description, and for which it has both the relevant plans and bug rules, it seems unlikely this can be much extended as it stands. Furthermore, as yet their system cannot cope with programs involving such things as procedures, or pointer variables, and in order to cope with these it would need a much richer plan representation method allowing both for the fact that plans could be split between procedures, and for reasoning about complex, more abstract, data types.

2.5 PUDSY

Lukey's system PUDSY [Lukey 1978] also aims to debug Pascal programs. Like IDS, PUDSY attempts to combine plan recognition and symbolic evaluation as integrated tools for program understanding.

PUDSY attempted to use clues such as meaningful variable names to help it understand programs, and also used "irrational code" (e.g. an assignment overwriting the result of a previous assignment before the value of the first assignment has been used) as an indication of bugs in a program. Plan recognition in PUDSY is achieved using a variant of the program schemata idea, but as this idea is not nearly as powerful as the surface plan formalism discussed in Chapter 3, and suffers from many of the same limitations as PROUST, this does not lead to such a rich hierarchical description of the program. Because of this lack of power and generality in PUDSY's representational methods the main technique used by PUDSY is that of symbolic evaluation of the source code. However symbolic evaluation is not really powerful enough on its own to provide the basis of a program *understanding* system and accordingly PUDSY can only deal with relatively simple programs e.g. simple sorting programs Finally it should be noted that, unlike IDS, PUDSY required a formal specification of what the program or procedure was supposed to do. This specification was included at the start of the program.

2.6 Eisenstadt et al.'s SOLO Debugger

The debugging system of Eisenstadt et al. [Eisenstadt and Laubsch 1981, Eisenstadt, Laubsch, and Kahney 1981] is aimed at novices, this time in the context of teaching AI programming using the language SOLO to psychologists. They also rely heavily on the plan diagram concept, and also use symbolic evaluation as a useful tool in program understanding. However their system is again not general purpose because they know in advance what problems the students will be tackling. Additionally the language SOLO is very impoverished (deliberately so, as they are interested in teaching very basic ideas of programming to essentially non-technical students). Accordingly they can use a generalisation of Ruth's technique to understand programs. The teacher supplies an "effect description" of the desired program. By symbolic evaluation an effect description is obtained for the student's program. The system then attempts to match these two effect descriptions. However it must be stressed that again the teacher has to supply knowledge of all likely methods of solution to the system.

2.7 ITSY

Domingue's system [1987] ITSY aimed to tutor novice LISP programmers. Like IDS, it too was based on the plan diagram formalism, but, because ITSY did not have a general plan recognition module, it did not try to understand programs as described earlier. Instead it had a library of "error clichés", expressed in the same surface plan formalism, and attempted to find occurrences of these stereotypical errors in programs by matching against the surface plan for the program. An occurrence of one of these was then used by the system to guide the tutoring of the student. So much of the work in ITSY was really concerned with analysing the kinds of errors that novices make in simple LISP programs, and with devising appropriate tutoring strategies. In this sense it actually has more in common with PROUST (although it did not have the same detailed hierarchy of plans and sub-goals) despite the fact that it used the plan diagram concept. However the idea that novices make errors which are themselves clichés, and that these can be expressed in the plan diagram formalism, is one which may well need to be incorporated into IDS were it to be used as the basis of a tutoring system for novices.

2.8 AURAC

Hasemer's system AURAC was also aimed at tutoring novice SOLO programmers. It too had a library of clichés, stored as template *patterns* corresponding to SOLO source code (independent of the variable names chosen by the programmer) representing such things as correctly formed switches, conditionals, and loops. In a later version of the system there were also some algorithmic clichés for specific tasks. No attempt was made to translate the programs into some other representation. This approach worked for AURAC because the tasks the student programs were trying to perform were relatively simple, enabling the system to have a library of most of the clichés the students would need. Clichés could be recognised even when spread over large numbers of program segments, and AURAC also had a system of "expectations" and "satisfactions" which were task independent, but ensured that no unexpected changes were made to the SOLO database. Being so language specific, it is hard to see how this approach could be extended to richer languages and more complex problems, but it does represent an early attempt to perform the same task as IDS i.e. debug programs with little or no knowledge of their purpose.

2.9 TALUS

TALUS [Murray 1986] is a debugging system for student LISP programs, and as such it is one of the most successful, fully implemented systems to date. It had information about the tasks the students had been set, this time in the form of a correct program. It used symbolic evaluation and a theorem prover to try and show that its stored program was equivalent to the one the student had submitted. It represented programming knowledge (i.e. common solutions) in a frame-like formalism. TALUS used program transformation techniques to simplify the student program, and if the student program failed on a pre-stored set of example input-output pairs, or if it could not be proved to be equivalent to the stored solution program then an error was assumed to be present. In this case the stored, correct program was transformed until it matched the student program as closely as possible, and then this closely matching version was used to suggest repairs to the student program. Because TALUS utilised a very general technique it was quite successful on a large variety of problems (18 in all), but it could not cope with global (free) variables in function definitions, or with operations with side-effects (except in special cases). However, despite its claims not to use any plan matching, in essence TALUS used a very simple form of plan identification, before starting on the process just described. This plan identification was essentially based on extracting a few features from the program being analysed (e.g. recursion type), and these features were then compared against the features stored against the known solutions. How well the features matched was then used to identify which of the known problems/possible solutions the student program was most likely to be an attempt at. Because this feature matching is quite impoverished (like most pattern recognition work based on feature matching), the system could get confused. This resulted in TALUS having its own version of error clichés for some problems. These were "solutions" that Murray found students came up with quite often for certain problems (in particular when they had mis-interpreted the problem).

2.10 PHAENARETE

Wertz's [1987] PHAENARETE system is a general purpose debugging system, written in LISP. It aims to debug novice LISP programs without any program specification, and with no prior knowledge of the task the programs submitted to it are supposed to perform, and makes no use of comments or of user-defined variable names. It works directly on the program text, using as its main technique a syntax checker (including a powerful spelling corrector) and a variant of symbolic execution which Wertz calls meta-evaluation. PHAENARETE represents its knowledge of programming in three forms, corresponding to three different types of knowledge. Its knowledge of syntax is held in the form of procedural *specialists*. There is one procedural specialist for each built in function of the LISP the system is aimed at. Each specialist contains information on the number of arguments the function takes, and on the types of the arguments. Its knowledge of what constitutes "good form" in programs (e.g. a recursive procedure should contain at least one control flow path through it which is not recursive, and that this should in general precede any recursive calls) is held as a set of procedural *pragmatic rules*. Its knowledge about the program being analysed (built up as the analysis proceeds) is held as a set of *cognitive atoms* in a database. the set of cognitive atoms essentially constitutes a semantic net [Quillian, 1968] and stores information about the variables and their inter-relationships, especially type information. The strategy used by PHAENARETE is essentially one of iterative improvement. It makes multiple passes over the program, making changes each time, until the resulting program remains unchanged. The first pass essentially invokes the spelling corrector, and checks the type and number of arguments to each call of a function (using the specialists). Where inconsistencies are found a call is made to a corrector to modify the program. If the type of some argument is unknown a type consistent with the relevant specialist is hypothesised. This first pass can sometimes add things like a missing COND to the program if the syntax suggests that this may have been intended. Subsequent passes essentially use the meta-interpreter to propagate types and values of variables through the semantic net checking for consistency. Where inconsistencies are found PHAENARETE has heuristics (based on common errors that novice programmers make) to modify the program. It can also use information from the meta-interpreter to check that variables which are involved in exit tests from loops or recursions converge to the exit value. Where they do not PHAENARETE has a set of heuristics to enable it to change the program in such a way that they do. In essence, PHAENARETE combines searching for error clichés with canned repairs. Many of its error clichés have much in common with the bug categorisation developed by Johnson et al. [1983], making it quite a useful system for novices. However, because PHAENARETE makes no attempt to understand a program in the way IDS or PROUST do, it cannot locate deep semantic errors. It can find and repair syntax errors, can deal with such things as uninitialised variables, unreachable code, and can find and repair simple cases of non-termination, but unfortunately sometimes turns a program containing such surface errors into programs with deep semantic errors. This is because it has no way of recognising that bits of code are actually near-misses to higher level plans, and hence cannot use this extra information to guide it in the repair process. All the examples discussed by Wertz [1987] are very small, and basically quite simple, so that often the strategy enables the system to repair programs which start off in an extremely garbled state. However, because of the (quite likely) possibility that the program will be turned into something correct (in the sense that it is syntactically correct, will run, and has "good form"), but which does not do what the programmer intended, it is quite hard to imagine this system being extended to cope with larger more complex examples without the addition of some kind of plan library.

2.11 Elsom-Cook's Lisp Debugger

Elsom-Cook's [1984] Lisp debugger was also aimed at tutoring novices. His system did not try to do plan recognition as such, but rather concentrated on such things as whether or not the student understood the concepts of function application, recursion and so on. It translated programs into its own internal representation (corresponding to a form of operational semantics for the program), and passed the result through to a special interpreter for this representation. Each primitive action in this representation had preconditions that had to be satisfied, and the interpreter would check that these held at the time of execution. Violations would trigger a tutoring session. Much of Elsom-Cook's work was really devoted to tutoring strategies, although the attempt to represent programs in a syntax independent form with some sort of underlying semantics has a certain amount in common with the work described here. However, his representation was not nearly as rich as the plan calculus (it can perhaps be seen as equivalent to just the IOSpec part of the plan calculus, discussed in Chapter 3, restricted to simply the basic machine operations), and as a result can not really be envisaged as scaling up to deal with large complex programs where the errors involve abstracting away from the underlying data structures and operations to high-level operations on much more abstract data types.

2.12 The Programmer's Apprentice Project

The Programmer's Apprentice Project [Rich and Schrobe 1978, Rich 1981, Waters 1978, 1979, 1982] is really the project which has provided the inspiration for, and the basic underlying theory for, the work described in this thesis. However, discussion of the relevant parts of this project will be given where appropriate in the rest of this thesis, and so will not be repeated here. In particular, most of Chapter 3 is devoted to an account of the surface plan/plan calculus knowledge representation technique developed by the above researchers.

2.13 SNIFFER

There have been several other debugging systems which have attempted to use the plan diagram formalism as their basis, most of which have been hampered by the lack of a general plan recogniser. Some of these have been mentioned above. One of the first such systems was Shapiro's [1978] SNIFFER system. SNIFFER had three basic components - a primitive "cliché finder", a "time rover" which recorded a program's execution history, and a set of "sniffers" each of which was essentially an 'expert' on a particular type of bug. Each sniffer utilised information from the time rover and from the cliché finder. However, Shapiro concluded that plan diagrams could not be fully effective for debugging programs until a full plan recogniser was available which could cope with such things as plans being spread out over more than one program segment. It would be interesting to re-attempt SNIFFER's approach using the work described in this thesis as the cliché finder.

2.14 Shapiro's Prolog Debugger

Shapiro's [1982] system (or rather collection of algorithms!) is rather different to the systems already described. For a start it does not try to recognise plans, or derive symbolic descriptions of the code, but instead relies on examining in detail the actual execution of the program on a specific input/output example which is known to be incorrect. In essence it is a sophisticated version of the well-known "fence-posting" technique that programmers use when trying to debug programs. This technique involves examining output from the program "between" places where the program was known to be functioning correctly, and a place where it is known to be wrong. Initially these two places are often the beginning and end of the program. If the program is functioning correctly at the place just examined, then output is examined between there and the place where it is incorrect. This process is continued until the place responsible for the incorrect behaviour is found. Shapiro's algorithms are only directly applicable to logic programming languages (in particular Prolog), and so are not directly of interest to this project, although Renner[1982] has done some preliminary work on trying to extend this approach to other languages.

Once the error has been located (in a specific clause of a Prolog program), Shapiro's algorithms then have a variety of heuristic techniques for repairing the clause in such a way that it will now give the correct behaviour on the buggy example, and also on all other examples of input/output behaviour that it has previously seen, and been told are correct. These heuristics are guaranteed to *eventually* give a correct program if the system sees enough input/output examples. Although interesting, this approach does not seem to be applicable to anything more than small simple programs, since the user is unlikely to be able to provide the correct value for some variable if this may involve hundreds or thousands of elements in some complicated data structure.

Interestingly, this system is not confined to program debugging. It can also be used for inductive program synthesis. By presenting the system with examples (input/output pairs) of the desired behaviour of the program it is able to synthesise a program that will work correctly at least on all the examples it has seen so far, and which is guaranteed to converge on the desired program after enough examples have been seen. The technique used by the system is to start with an empty program (one with no clauses), and then to debug it. As each new example is presented the program as it stands is either left unchanged if the program works on this new example, or is debugged into one that will work on this example in addition to earlier ones.

2.15 Summary and Conclusions

Murray [1986] makes a very strong case in favour of his program verification approach to program debugging, while Johnson [1986] argues equally strongly for plan based program debugging. How can these two viewpoints be reconciled? The answer is that both techniques will be needed in a debugging system which is intended to be general purpose. Rich's[1981] plan calculus is the only currently available knowledge representation technique which offers the potential ability to move smoothly between both techniques. Both PROUST and TALUS are used (or are intended for use) solely in a tutoring environment. In this case the systems have available to them a set of possible solutions against which to check the student's program. So in the case of PROUST the system can check that the student's program contains the plans to achieve the same goals as the target program, while TALUS can try to prove equivalence of the student program to the target program. However, in a situation where a target program is not available, the TALUS approach reduces to program verification which is extremely hard and combinatorially explosive, even assuming that a formal specification is available. The problem with a pure plan based approach is that apart from the basic plans needed, one also needs an enormous number (probably combinatorially explosive!) of plans or program transformations to capture minor implementation variants of the same algorithm. The theorem proving approach can get round this because of its generality- these variants can often quite easily be proved equivalent to some standard reference plan. Working within the plan calculus framework we can get the best of both worlds. First of all, we don't necessarily need to know the problem in order to try and understand the program since bottom up plan recognition can lead to an understanding of much of the program. Furthermore, as discussed in Chapter 5, this plan recognition can be viewed as an efficient form of theorem proving. Additionally we can always call on a theorem prover to deal with those parts of the program that do not exactly match known plans. Additionally, it turns out that the near-misses found by the plan recognition process can be regarded as failed attempts to prove some theorem, for which the steps in the proof have been conveniently grouped together. The missing parts of the near-miss can be regarded as telling us how the theorem failed to be proved. Near-miss correction as proposed in this thesis can then be viewed as very similar in some sense to Murray's technique of debugging a program by repairing the proof of its equivalence to some reference function. The rest of this thesis will largely concentrate on the theory and algorithms for performing plan recognition within the plan calculus.

PART 2. THEORY AND ALGORITHMS

Chapter 3.

The Plan Calculus

This chapter will give an account of the plan calculus as developed by Rich [1981]. This is a knowledge representation technique for programs and programming knowledge in procedural languages with the advantages that it has both an intuitive appeal and a rigourous axiomatic interpretation enabling sound formal reasoning to be applied. We will begin by giving an intuitive account of the representation, and will then present its formal semantics. The intuitive interpretation of the formalism defines programs and plans as control and data flow graphs. This enables one to regard the plan library as a graph grammar, and the recognition process as a parsing process, and chapters 5 and 7 will present our graph parsing algorithm for doing this. It will also be shown how this apparently heuristic approach corresponds to theorem proving if we take the axiomatic view of the plans. This gives our graph recognition algorithm a sound theoretical basis, unlike other systems which do some kind of heuristic plan recognition (e.g. Johnson [1986]). It will also be shown how the formal account justifies some of the more intuitively based modifications ("hacks") we have made to the parser, and we will also give an account of the algorithm we have used to make Rich's notation for plans "machine-readable". Other attempts to use Rich's framework for program understanding [Wills, 1986,1990] use only a subset of his plan library, and the plans have been converted to graphs for parsing by hand. Our system can read the plans in their frame-like notation, and convert them automatically to appropriate graph structures. This conversion process will be described in Chapter 7, and in itself makes the plan calculus a lot more usable.

3.1 An Informal Account

3.1.1 Surface Plans

This project draws upon Rich's [1981] work in that the program is represented not by its source code but by a surface plan. This is essentially a representation of the program in terms of its control and and data flow. By representing this graphically we can think of the surface plan as a control and data flow graph. For example, Figure 3.1 shows the surface plan corresponding to the piece of code below:

```
if x<y then
     z:=x+y+1
else
     z:=z-1;
```

In this diagram control flow is represented by thick (bold) arrows and data flow is represented by ordinary (thin) arrows. Basic operations such as applying a binary function(**@binfunction**) are shown by boxes with the operation indicated inside, and with the relevant function and arguments as inputs to the box. The result is indicated as an output from the box. Tests, which determine which way control flow will go depending on the result of applying a binary relation (**@binrel**) to two arguments, or applying a predicate (**@predicate**) to a single argument, are also shown as boxes, with the relevant relation (e.g. ">") or predicate (e.g. "=nil") indicated as the first input, and the appropriate arguments as the other inputs[1]. The YES/NO subpartitions of a test box indicate which way control flow will go depending on the result of that test. Data flow between two such boxes indicates that data values

[1] Sometimes, when it is convenient, test boxes will be shown with the predicate or binary relation indicated in the test box, thus reducing the apparent number of inputs, but this should be seen as merely "syntactic sugar" for the more precise representation.

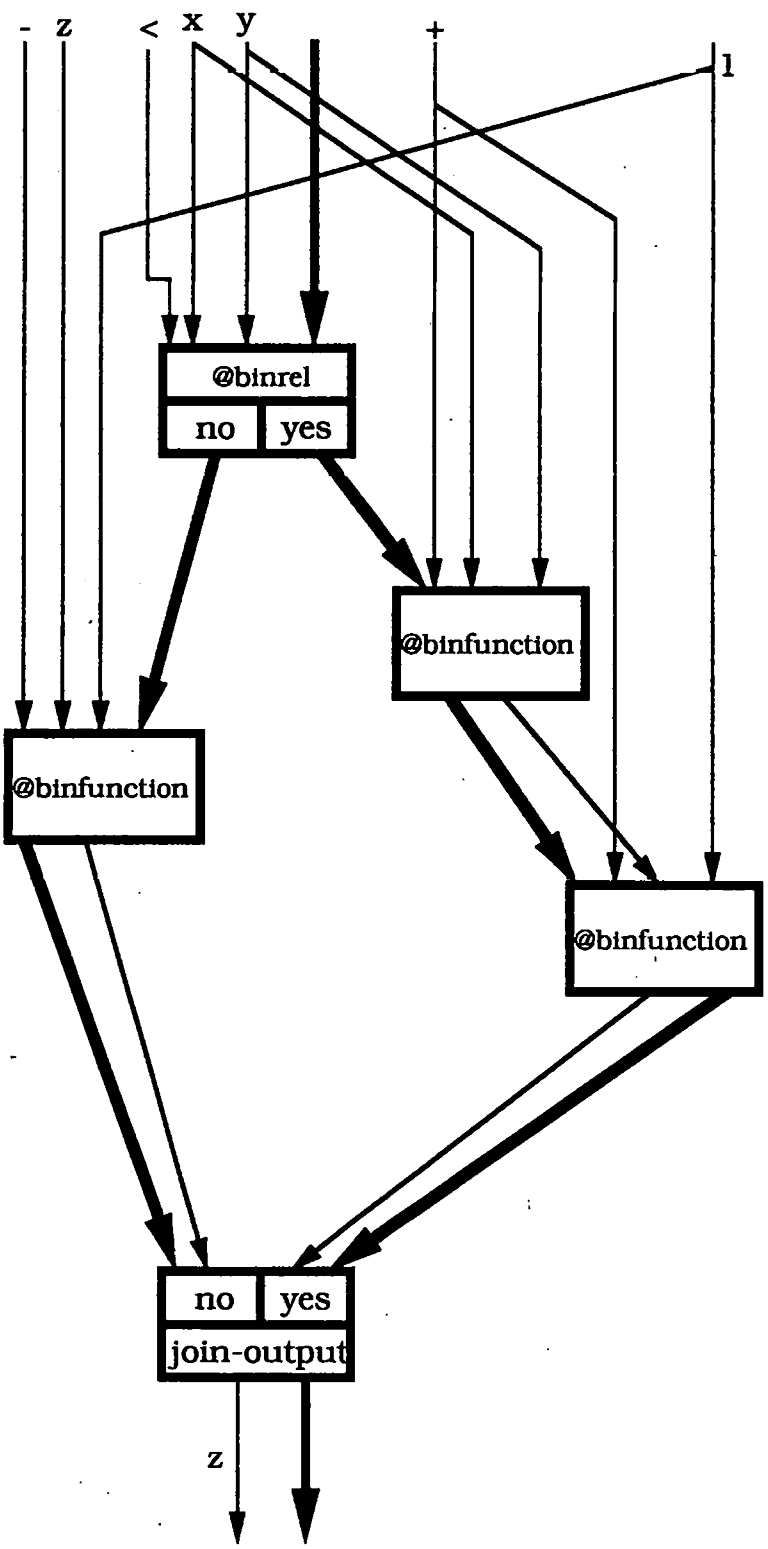


Figure 3.1 A Simple Surface Plan

produced by one are used as inputs to the other. Control flow between two boxes indicates that the first box is to have its action performed before the action corresponding to the second box. We will often adopt the convention however, that control flows will not be shown when they are actually implied by the data flows. One way of interpreting such boxes is to regard each box as being a processor in its own right. Such a processor is "activated" when it has received input on all its incoming data flow arcs, and it has also received a "control flow token" on its incoming control flow arc if it has one. When it is activated it outputs suitable values on its output data flow arcs and outputs a control flow token on its output control flow arc if it exists. These values and tokens then pass along the arcs to activate other boxes. Where an arc subdivides the value on the arc is transmitted along all branches of the arc. Processors corresponding to tests generate a control flow token on either the YES output or the NO output depending on the values of the inputs to the test and the actual test applied. Now the reason for applying tests to data is generally so that one can generate different values depending on the result of the test and then use these values later in the program. To enable later processors to make use of whichever values are actually generated after a test we need somehow to make all of these available as potential inputs to other processors. Th:s is done using join-outputs boxes which reconnect the separate data flows corresponding to divergent control flow routes at tests.

Program segments are also indicated by boxes. However these boxes perform a more complicated operation than basic ones. Subsegment nesting is shown by nesting of boxes. Looping constructs have been translated to a recursive representation. For example, a **while** statement of the form

```
while condition do action;
```

will be interpreted as if it were just a procedure call

```
loop(......)
```

where the procedure loop is defined by:

```
procedure loop(......);
begin
    if condition then
    begin
        action;
        loop(......);
    end
end;
```

This has the advantage of removing cycles from the surface plan thus making subsequent analysis easier. This kind of recursive subsegment nesting is indicated in plan diagrams by a spiral line connecting the outer segment to its inner recursive copy. An example of this representation of loops can be seen in Figure 3.2 corresponding to the code below:

```
sum:=0;
while not(eof) do
begin
    read(n);
    sum:=sum+n;
end;
```

Labels corresponding to variable names in the original Pascal program have been attached to the data flow arcs as an aid to understanding the diagram and in order to enable IDS to converse with users and suggest edits in terms of the variable names with which they are familiar.

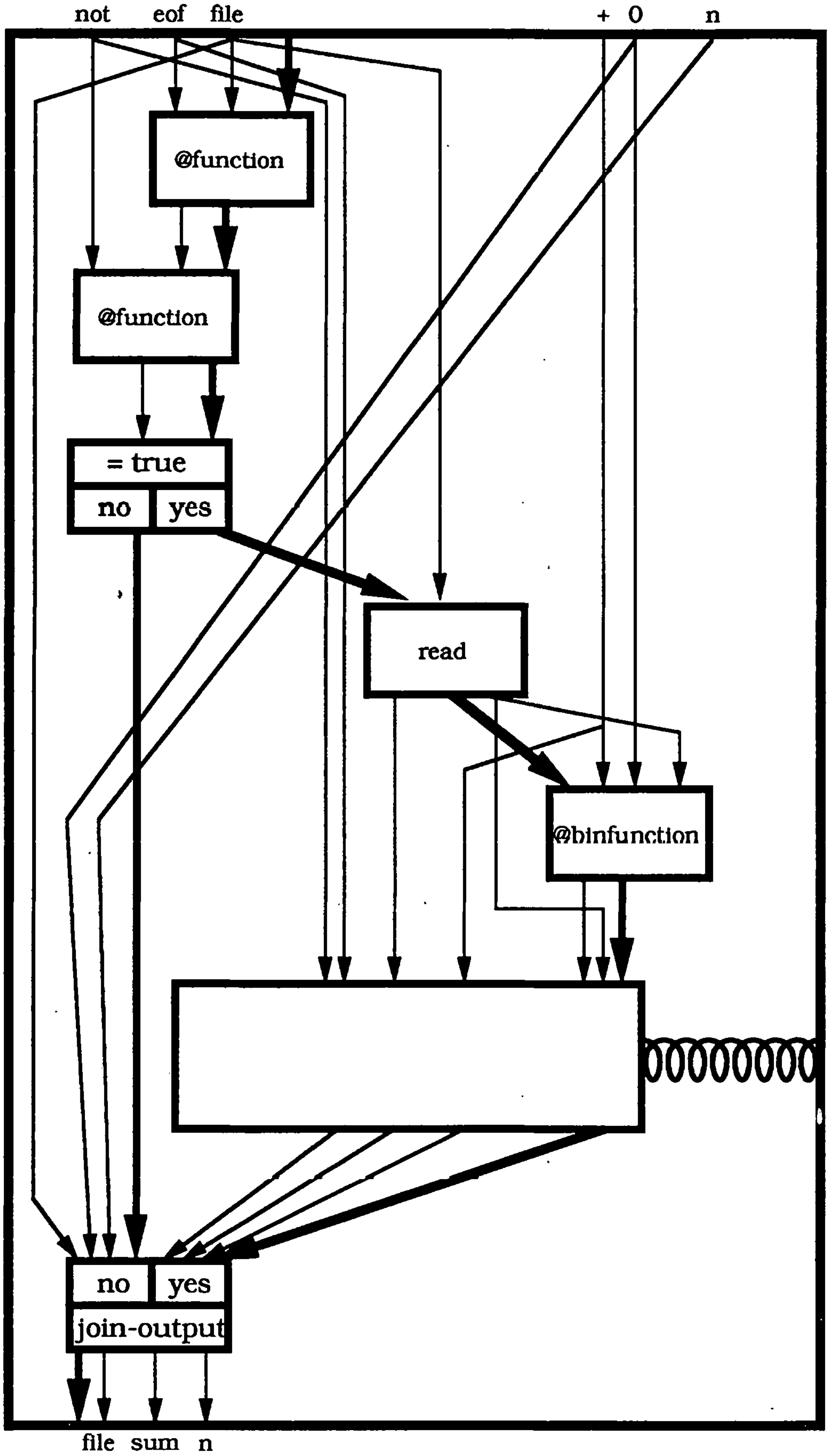


Figure 3.2 Surface Plan for Simple Loop

Such surface plans can then form the basis of a hierarchical system of representations of the program. To go from a lower level to a higher level in such a hierarchy it is necessary to recognise some sub-graph at the lower level representation as performing some known function. The subgraph can then be replaced by a segment quite explicitly representing the function performed. In this way the program is progressively represented by higher and higher level segments, until at the top level the program is represented by a simple graph (possibly a single segment) performing the overall function of the program. Using this and the technique of temporal abstraction [Waters 1978, 1979, Rich 1981] which enables one to reason about a set of sequentially generated items as a single collection it will be shown in Chapter 8 that it is possible to recognise that the program shown on the next page reads numbers from the terminal (or a file), sorts them into ascending order using a list structure, and then outputs the sorted set of numbers.

3.1.2 The Plan Library.

In order to do the hierarchical recognition just described a library of commonly occurring programming cliches (whole algorithms and even code fragments) is needed, stored in the same plan diagram formalism. This project has taken as the basis for its library that developed by Rich [1981], although it has added some new plans of its own. In principle such a plan library can be used in two ways:

(i) To enable programmers to specify their code at a high level in terms of plans in the library, leaving the system to actually implement the code in the desired language, and

(ii) To analyse code written by a human programmer giving the system a high-level understanding of what the code does and how it

```
program sort(input,output);
type listelement = record
            numb : integer;
            next : ^listelement;
            end;
      plist = ^listelement;
var head, p : plist;
    n : integer;

procedure addtolist(n : integer; t : plist);
var  p : plist;
begin
    new(p);
    p^.numb:=n;
    if  t = nil then
        begin
            p^.next := head;
            head:=p;
        end
    else
        begin
            p^.next:=t^.next;
            t^.next:=p;
        end;
end;

procedure findplace(n: integer; var p: plist);
var  t : ^listelement;
    found : boolean;
begin
    if  head^.numb > n  then
        p:=nil;
    else
        begin
            p:=head;
            t:=p^.next;
            found:=false;
            while not found do
                if  t <> nil then
                    if  t^.numb <= n then
                        begin
                            p:=t;
                            t:=t^.next;
                        end
                    else found:=true
                else found:=true;
        end;
end;

begin
    head:=nil;
    while not eof do
        begin
            readln(n);
            if head<>nil then findplace(n,p)
            else p:=head;
            addtolist(n,p);
        end;
    p:=head;
    while p<>nil do
        begin
            writeln(p^.numb);
            p:=p^.next;
        end;
end.
```

does it, in terms of what the overall goal of the code is, and which parts of the code achieve which sub-goals.

Although there has been some success at the first of these goals [Waters 1982], the second has until recently been frustrated by the lack of a suitable plan recogniser which could analyse plan diagrams into their constituent plans, although Brotsky [1984] has done some work on this. This project has developed [Lutz 1986, 1989] a generalisation of traditional chart parsing techniques [Thompson and Ritchie 1984] which can perform this recognition task, while Wills [1986, 1990] modified Brotsky's [1984] algorithm so that it is now similar in some ways to a chart parser, although it does not run bottom-up quite so naturally. Both Brotsky's and Wills' work will be discussed in detail in Chapter 7. More recently, Wills (personal communication 1992) has switched to an algorithm substantially based on that described here and in [Lutz 1989].

In the plan library there are several different kinds of information. These are:

1. Definitions of primitive operations e.g. **@function** which takes as input a function and an object in the domain of the function and applies the function to the object producing an object in the range of the function. Similarly **@binfunction** takes a binary function and two objects as inputs. In addition there are definitions of common operations on various types of object e.g. **set-add**, a binary function taking a set and an object as inputs and producing a new set (equal to the input set with the object added to it) as output. Another commonly used operation is **newarg**, which takes as input a function and two objects. Its output is another function equal to the input function in all ways except that its value when applied to the first of the two objects is

equal to the second of the two objects. A related operation is **#newarg**(read impure newarg), which behaves exactly like newarg except that the output function is specified to be the same object as the input function i.e. they are the same object with different behaviour at different times, rather than different objects. This is used for representing things such as array updating, record field updating, list surgery and other operations that change objects by side-effect.

Each primitive operation has associated with it preconditions and postconditions. Preconditions are conditions that must be satisfied by the inputs of an operation for a use of it to be valid. Postconditions are conditions satisfied by the outputs of an operation given that its preconditions have been satisfied.

2. Definitions of primitive objects (and their properties) known to the system e.g. built-in functions, predicates, binary functions and relations. An example is the binary function for addition, **plus**, with the definition specifying such things as type information for its inputs and outputs, the facts that it is associative and commutative, and that it has identity element 0, and so on. Other functions specified here are such things as **car** and **cdr** (for Lisp), and **plus-one** (a function which increments an integer by one).

3. Definitions of primitive data-types e.g. **integer** and **binfunction.** Subtype information is also specified.

4. Temporal plans specifying algorithms or commonly occurring code fragments. Each such plan not only has control and data flow information associated with it, but may also have additional constraints specifying other relationships that must hold between parts of the plan. Note that in general we will consider control flow constraints to be such extra constraints on a plan considered as a

pattern of data flows. In this way irrelevant control flow information occurring in actual programs will be ignored, since the plans will only specify essential control flow constraints. A good example of a temporal plan is **trailing-generation-and-search**, shown in Figure 3.3. This plan captures the data flow pattern common to code which searches data structures for an object satisfying some predicate, keeping track of both the object it is currently examining, and also the previous object. An example of such (pseudo-) code is shown below (with the underlined code corresponding to the plan):

```
repeat
    begin
        c:=p^.next;
        .
        .
        .
        quitloop if pred(c) or c=nil
        p:=c;
    end
endrepeat;;
```

In this code the composite function ^.next corresponds to what we have labelled .action.op in the figure, p corresponds to .previous and c to the output from the **@function** node (labelled .current). The predicate .exit.if.criterion in the figure corresponds to the exit test pred(...) in the code.

5. Data plans specifying how compound data-objects are built up out of more primitive ones. Typical examples of this are plans such as **iterator** and **labelled thread**. An **iterator** consists of an object, and a function whose domain and range are both equal to the type of the object. Such a data object can be used to generate an entire sequence of objects, starting with the initial object, applying the function to it to get the second, applying the function to the second to get the third

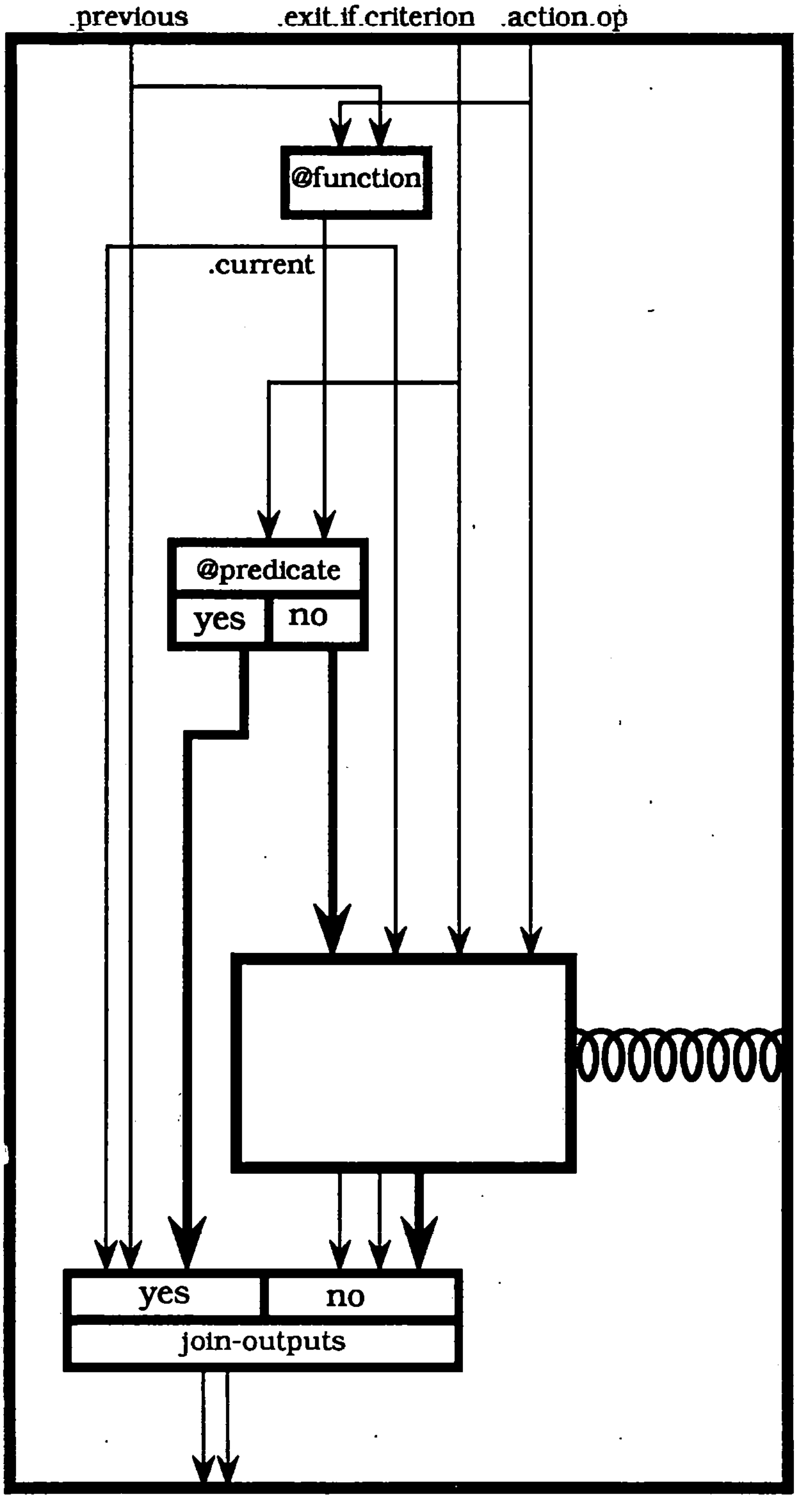


Figure 3.3 Trailing Generation and Search

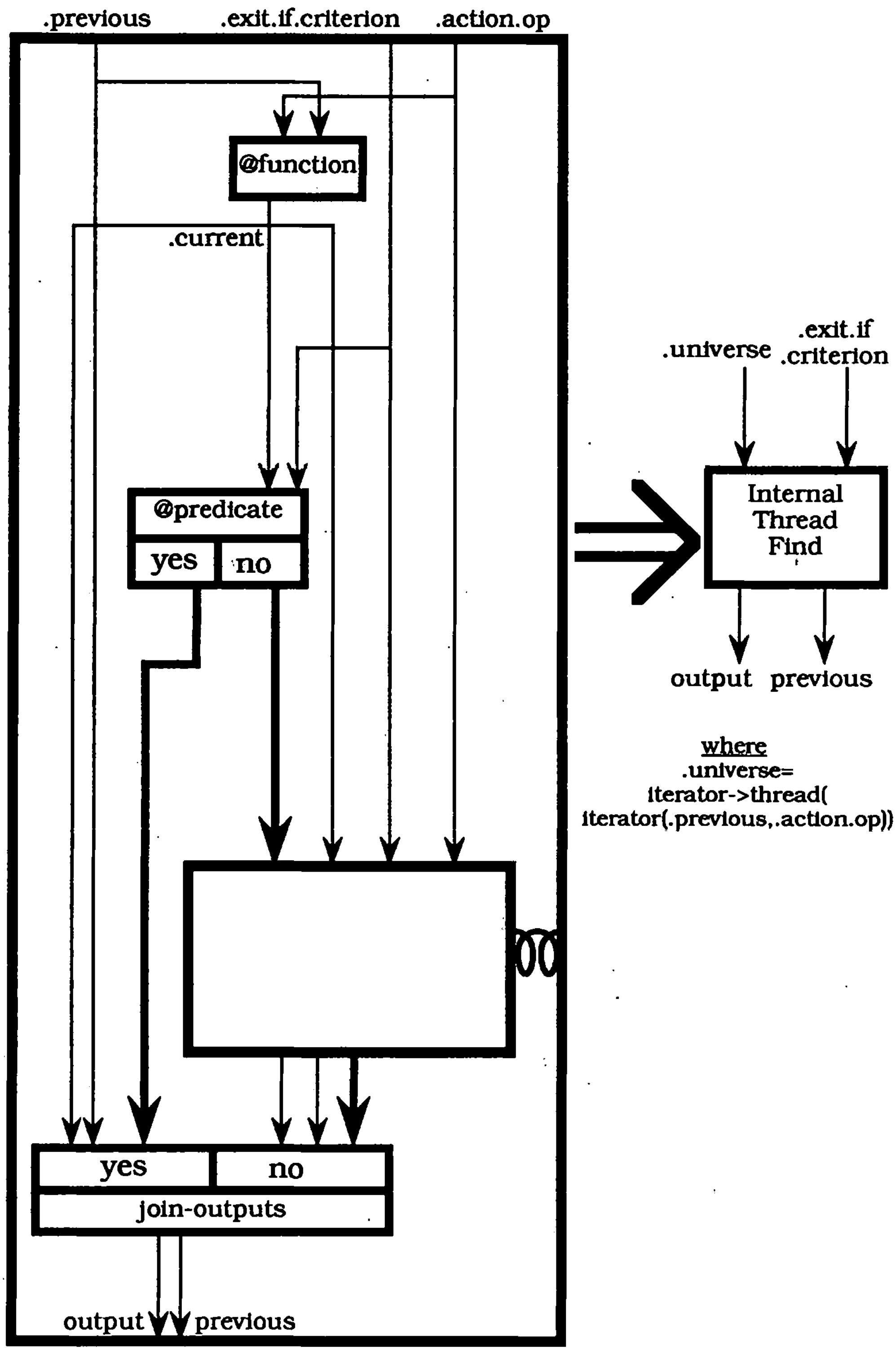
and so on. This is a remarkably common construct in programming. For instance using the number 1 as the initial object and the function **one-plus** (which adds one to its argument) gives an iterator which generates the natural numbers. Alternatively, starting with a list as the initial object and the Lisp function **cdr** as the function lets one generate successive tails of a list as is so often done in programming. The **labelled thread** data type also consists of two parts - a function, and another compound data type, a **thread**. A **thread** consists of a set of objects and a second injective function mapping the set to itself. The set should be thought of as a set of nodes, and the second function as a successor function which takes a node and returns its successor. So a **thread** is really a linear graph-like structure. The other function needed with the **thread** to make a **labelled thread** is the labelling function. Its domain is the set of nodes in the **thread**, and it produces values to be associated with each node. A typical example of a **labelled thread** is a Lisp list. The (pointers to) dotted pairs can be thought of as making up the set of nodes, the **cdr** function is the successor function, and the **car** function gives us the value associated with each node. Again the definitions of such data types specify constraints that must be satisfied by the parts of the structure. These could be just simple type information, or could be more complex relationships e.g. that a **thread** must not have a cycle i.e. that there is no way one can get back to anywhere earlier in the thread by repeatedly applying the successor function.

6. Data Overlays specifying how some data object may be viewed as an implementation of an object of some other type. These specify how it is possible to view one type of data object as another. For instance, the data overlay **iterator->thread** specifies that an iterator can be regarded as a thread by treating the set of objects generated by the

iterator as the set of nodes in the thread, and the function part of the iterator as the successor function for the thread. This overlay and others like it, play a crucial role in **temporal abstraction,** which bridges the gap between temporal sequences of objects as produced by loops and iterators, and other non-temporal data structures such as **threads**, **lists**, and directed graphs(**digraphs**). In many of the diagrams that follow no distinction has been made between **iterators** and **threads**, in that an initial object and a successor function have been grouped together as a single object and treated as a **thread**. This is purely to avoid having to show multiple levels of analysis in diagrams which are already complicated enough.

7. Temporal Overlays specifying how a temporal plan may be regarded as an implementation of some operation. A good example of this is the overlay **trailing-generation-and-search->internal-thread-find**. This overlay captures the idea that the pattern of code in **trailing-generation-and-search** can be used to implement an **internal-thread-find** operation which given a **thread** and a **predicate** as input returns the first node in the **thread** satisfying the **predicate** (it also returns the previous node). Now of course it should be noted that the **trailing-generation-and-search** plan makes no reference to **threads**. So the overlay has to specify which objects being input to the searching plan can be viewed as constituting a **thread**., and in what way. This is shown in Figure 3.4, where the constraint underneath the diagram indicates that the .previous object and the .action.op function input to the **trailing-generation-and-search** plan can be grouped together as an **iterator** which, when viewed as a **thread** using the **iterator->thread** overlay, forms the input to the **internal-thread-find** operation.

Temporal abstraction, mentioned above, was one of the most important notions introduced by Rich and Waters and subsequently



where
.universe=
iterator->thread(
iterator(.previous,.action.op))

Figure 3.4
Trailing Generation and Search->Internal thread Find

further developed by Eisenstadt and Laubsch [1982]. As alluded to above, this enables one to view a series of data objects computed in a temporal sequence (e.g. the values of a variable as a loop is executed) as a single data object. This is similar to the point of view programmers have when considering a sequence of reads from a file - they can switch between regarding the file as a single stream-like object, or they can think of the temporal sequence of values produced by the reads. It is the formalisation of this reasoning which enables the Programmer's Apprentice project to take a really high level view of many programs and to recognise the common pattern (e.g. filtered iteration) behind many different programs and implementation techniques.

There are several advantages to this method of representing programs and plans in terms of their control and data flow. These include:

a) The program representation is no longer dependent on the specific variable names chosen by the programmer. Therefore at least one type of superficial variation between programs has been removed. It should be noted in this connection that from a theoretical point of view variables in a programming language are just a device for ensuring a desired data flow - it is the data flow that is important, not the variables.

b) Many (not all!) structurally different programs can be represented by the same surface plan. Thus some other superficial differences between programs can be eliminated by this technique. Of course it is not possible for any representation to be completely canonical with respect to program equivalence. If it were, we could use the representation to solve the program equivalence problem (by translating the programs into the representation, and then checking that the representations are the same) which is a well-known non-decidable problem.

c) It is a language independent representation. Therefore a lot of the reasoning etc. applied to a program is applicable to programs in any language. There does, however, need to be a language dependent translation process to go from the source code to the surface plan of a program. In order to be able to suggest edits and so on, a language dependent plan to source code translator is also needed. These two modules apart, most of the rest of the plan recognition system is written in a language independent fashion.

d) Plans can be combined in a linear fashion to form new plans without the component plans interfering with each other. All that needs to be done is to connect the output data (and control) flows from the first plan to the appropriate input data (and control) flows of the second, and provided that any special constraints required by the plans (e.g. type restrictions on the inputs etc.) are satisfied the resulting plan will be a valid combination of the two original plans.

e) Plans can easily be combined in such a way that they share common sub-plans. This makes it easy to represent and reason about the common programming practice of not recomputing values which have already been computed.

This ability to easily combine plans enables one to form a library of commonly occurring plans which can then be put together to make up programs. The recognition process is then primarily one of seeing how a given program is actually built up from combinations of these known plans, and debugging can partly be seen as the attempt to understand the program using the above recognition technique, noting any near matches to known plans. Later chapters will describe these processes in more detail.

3.2 Semantics

In addition to the many desirable properties described above, this representation of programs also has the advantage of having a properly defined semantics. As we have presented it so far plan diagrams form a reasonably intuitive and powerful knowledge representation technique for programs and programming knowledge. However, AI has been bedevilled at various times with intuitively based knowledge representation techniques that foundered in various ways until they were given a proper semantics (semantic nets are a case in point), often based on logic in some way. This section will give a brief overview of the semantics of the plan calculus (readers interested in a full description are referred to Rich [1981]) by discussing the **trailing-generation+search** plan and the **trailing-generation+search->find** overlay described earlier. To do this will actually involve looking at quite a large subset of the plans and overlays we will make use of in later chapters.

3.2.1 The Compact Notation

3.2.1.1 The Basic Notation

Rich [1981] defines all the plans and overlays etc. used in the plan calculus in a compact text-based notation. Some examples of this notation (including the **trailing-generation+search** plan, shown earlier in Figure 3.3) are given below:

IOspec **@function** / .op(**function**) .input(**object**) => .output(**object**)
preconditions .input∈domain(.op)
postconditions apply(.op,.input)=.output

```
Temporal Plan cond
        roles .if(test) .end(join)
        constraints cflow(.if.fail, .end.fail)
                    ∧ cflow(.if.succeed,.end.succeed)
```

```
Temporal Plan trailing-generation+search
        extension iterative-generation trailing-search
        roles .current(object) .previous(object) .exit(cond)
            .action(@function) .tail(trailing-generation+search)
        constraints .current=.action.output ∧ .previous=.action.input
```

There are several points to be made about this notation. Firstly, it is frame-like. Each computational object is defined, first by giving its type and then its name. For example **cond** and **trailing-generation+search** are defined to be temporal plans, while **@function** is defined to be a basic operation of the calculus (using the keyword *IOspec*). Then each definition has various "slots" associated with it giving other information. For instance, **@function** is defined to be a primitive action with two inputs (the first of which will be referred to as .op and is constrained to be a function, and the second of which will be referred to as .input and can be any object), and one output which will be referred to as .output. Then **@function** has a *preconditions* slot which states that for an occurrence of **@function** in a program to be valid the .input value must be in the domain of the function, and a *postconditions* slot which states that if the preconditions are satisfied for an instance of **@function**, then .output for that instance will be equal to the result of applying the function .op to the input .input. It should be noted here that all basic operations of the calculus (e.g. **@function**) have two situations associated with them - an input situation (.in) and an output situation (.out), denoting the state of the program (i.e. the values of all objects and data structures etc.) in the program at the time the action is activated, and after the action has been done. We will use the symbol ⊥ to denote a situation that is never reached. So for example, in a conditional, if the then part is performed as a result of evaluating the test, then all the actions in the else part have ⊥ as their input and output situations.

The temporal plan **cond** corresponds to an abstract conditional. It has two component parts (referred to as roles) - a **test** (referred to as .if), and a **join** (.exit). **Tests** all have two output situations associated with them (.succeed and .fail). **Joins** (whose function is to reconnect diverging control flows from **tests**) have two incoming situations (.succeed and .fail), and the constraints in the **cond** plan simply state that control must ultimately (after any actions that may or may not be performed on the succeed or fail sides of the **cond**) flow from the .succeed situation of the **test** to the .succeed situation of the **join**, and similarly for the .fail control flows. Note that nothing is said at this level of abstraction about what actions (if any) actually happen on the succeed side or the fail side of the **cond**, and accordingly nothing is said about what divergent data flows may be reconnected at the join. If a plan involves a **cond**, but also involves some actions which produce data values to be connected at the **join** then the plans will often specify a tighter restriction on the **join** of the **cond**. In particular they may specify that the **join** is a **join-outputs** (which reconnects a single data flow from each side of the conditional) or a **join-2-outputs** (which reconnects two such data flows from each side). Examples of this can be seen in some of the plans given below.

The definition of **trailing-generation+search** is more complicated. It too has a *roles* slot, specifying what actions and objects are involved in the plan, and what types these actions and objects must have, and how they will be referred to when we need to refer to parts of the plan. In this case we see that **trailing-generation+search** has five roles - two **objects** (.previous and .current), a **cond**, an **@function** action, and another **trailing-generation+search**, making this into a recursively defined plan. It also has a *constraints* slot, specifying extra information that must hold between (or about) the roles in order for a collection of roles of suitable types to constitute an instance of **trailing-generation+search**. In this case it specifies that the .previous object is the input to the .input port of the **@function** .action. Similarly, the .current object is the value coming from the .output port of the **@function**.

3.2.1.2 Inheriting Constraints Via Specialisation and Extension Links

If the above were the complete definition, then the resulting plan would be that shown in Figure 3.5, which clearly is not the whole **trailing-generation+search** plan shown earlier (Figure 3.3). However, the definition of **trailing-generation+search** also has a *extension* slot. This gives the names of other plans to which this plan is related (by having extra roles and (possibly) constraints) and from which it inherits other constraints. In this particular case, **trailing-generation+search** is defined to be an extension of both **iterative-generation** and **trailing-search**. These plans are given below:

```
Temporal Plan trailing-search
        extension iterative-search trailing
        roles .current(object) .previous(object) .exit(cond)
                .tail(trailing-search)
        constraints instance(join-two-outputs, .exit.end)
                        ∧ .current=.exit.if.input
                        ∧ .previous=.exit.end.succeed-input-two
                        ∧ .tail.exit.end.output-two=.exit.end.fail-input-two
```

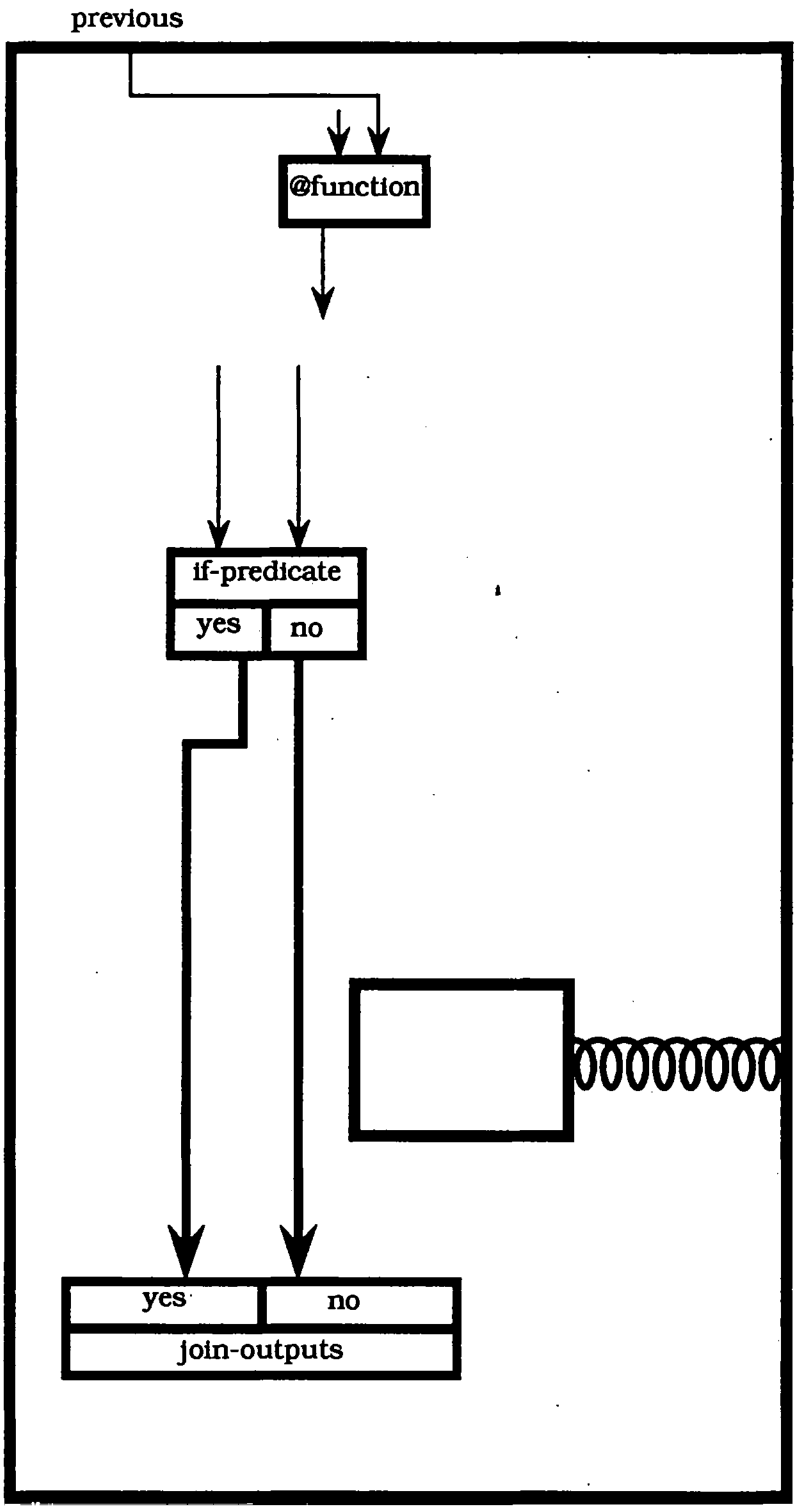


Figure 3.5 Partial Trailing Generation and Search

Temporal Plan **iterative-generation**
    specialization **iterative-application**
    roles .action(**@function**) .tail(**iterative-generation**)
    constraints .action.output=.tail.action.input

which, as can be seen, add several constraints (and hence arcs of the graph).The **iterative-generation** plan corresponds to the pattern of underlined code below:

```
repeat
    begin
        c:=p^.next;
        .
        .
        .
        p:=c;
    end
endrepeat;
```

and represents the common technique of iteratively applying a function to generate a sequence of objects, each of which is the result of applying the function to the previous object, while the **trailing-search** plan corresponds to the following code:

```
repeat
    begin
        c:=......;
        .
        .
        .
        quitloop if pred(c) or c=nil
        p:=c;
    end
endrepeat;
```

which represents iteratively producing some item and testing it for some condition holding, while keeping track of the previous item. The first thing to note about this is that where one plan is an *extension* of another then the roles they have in common are given the same names. This makes it easy to propagate constraints from a plan to an *extension* of that plan. Now, **trailing-search** is itself defined to be an *extension* of **iterative-search** and **trailing**. These are shown below:

```
Temporal Plan trailing
        extension single-recursion
        roles .current(object) .previous(object) .tail(trailing)
        constraints .current=.tail.previous
```

```
Temporal Plan iterative-search
        specialization iterative-termination-predicate
                       iterative-termination-output
        roles .exit(cond) .tail(iterative-search+nil)
        constraints .exit.if.input=.exit.end.succeed-input
```

Trailing itself is an *extension* of:

```
Data Plan single-recursion
        roles .tail(single-recursion+nil)
```

which adds no further constraints. However, **iterative-search** is defined (via its *specialization* slot), to be a *specialisation* of two other plans i.e. **iterative-termination-predicate** and **iterative-termination-output**. *Specialisation* is another method whereby one plan can inherit from another, but unlike *extension* no extra roles are involved. A plan P1 which is a *specialisation* of another plan P2 has the same roles and constraints as P2, and also has additional constraints. **Iterative-termination-predicate** and **iterative-termination-output** are shown below:

```
Temporal Plan iterative-termination-predicate
        specialization iterative-termination
        roles .exit(cond) .tail(iterative-termination-predicate+nil)
        constraints instance(@predicate,.exit.if)
                  ∧ .exit.if.criterion=.tail.exit.if.criterion
```

```
Temporal Plan iterative-termination-output
        specialization iterative-termination
        roles .exit(cond) .tail(iterative-termination-output+nil)
        constraints instance(join-output,.exit.end)
                     ∧ .exit.end.fail-input=.tail.exit.end.output
```

Both these plans are *specialisations* of **iterative-termination**:

```
Temporal Plan iterative-termination
        extension single-recursion
        roles .exit(cond) .tail(iterative-termination+nil)
        constraints [.tail=nil ↔ .exit.if.succeed≠⊥ ]
                        ∧ cflow(.exit.if.fail,.tail.exit.if.in)
                        ∧ cflow(.tail.exit.end.out,.exit.end.fail)
```

Returning to **iterative-generation**, it in turn is a *specialisation* of **iterative-application**:

```
Temporal Plan iterative-application
        extension single-recursion
        roles .action(@function) .tail(iterative-application)
        constraints .action.op=.tail.action.op
                      ∧ cflow(.action.out, .tail.action.in)
```

which adds the final constraints to the graph. In this way it can be seen that plan diagrams acquire their data and control flow arcs by a complicated process of inheriting them from other plans.

3.2.1.3 Overlays and Data Plans

Now we can turn our attention to the overlay **trailing-generation+search->find**, which states how an instance of a **trailing-generation+search** plan in a program can be viewed as an **internal-thread-find** operation as discussed earlier in this chapter. We begin by giving the appropriate definitions of the various data structures involved:

```
Data Plan digraph
      roles .nodes(set) .edge(binrel)
```

This defines a directed graph to be a set of nodes together with an edge relation on the set of nodes. **Trees** are defined to be a *specialisation* of **digraphs** such that they have a root (i.e. a node such that every other node can be reached from it following edges of the graph), and are non-cyclic:

```
Data Plan tree
      specialization digraph
      roles .nodes(set) .edge(binrel)
      properties ∀G instance(tree,G) ⊃ [∀xy root(G,x) ∧ root(G,y) ⊃x=y]
      definition instance(tree,G) ≡ [  instance(digraph,G)
                                      ∧ ∃x[root(G,x)]
                                      ∧ ∀x[¬successor*(G,x,x)]]
```

Finally **threads** are defined to be specialisation of **trees** such that every node has a unique successor:

```
Data Plan thread
      specialization tree
      roles .nodes(set) .edge(many-to-one)
      properties ∀T instance(thread,T) ⊃
                          [ ∀xy [terminal(T,x) ∧ terminal(T,y) ⊃ x=y]
                           ∧ ∀xyz [successor(T,x,y) ∧ successor(T,z,y) ⊃
                                                                  x=z]  ]
```

It should be noted that both of these have an extra *properties* slot which states useful properties of the data type involved. The relations etc. used in the above definitions are given below:

```
Binrel node : digraph × object → boolean
      definition node(G,x) ≡ (x ∈  G.nodes)
```

Trirel successor : **digraph** × **object** × **object** → **boolean**
definition successor(G,x,y) ≡
[node(G,x) ∧ node(G,y)
∧ binapply(G.edge,x,y)=true]

Trirel successor* : **digraph** × **object** × **object** → **boolean**
definition successor*(G,x,y) ≡ ∃i successorn(i,G,x,y)

Quadrel successorn : **natural** × **digraph** × **object** × **object** → **boolean**
definition successorn(i,G,x,y) ≡
[[i=1 ∧ successor(G,x,y)]
∨ ∃z[successor(G,x,z)
∧ successorn(oneminus(i),G,z,y)]]

Binrel root : **digraph** × **object** → **boolean**
definition root(G,x) ≡ ∀y[(node(G,y) ∧ x≠y) ⊃ successor*(G,x,y)]

Binrel terminal : **digraph** × **object** → **boolean**
definition terminal(G,x) ≡ [node(G,x) ∧ ¬∃y successor(G,x,y)]

Now the primitive operation **internal-thread-find** can be defined. It takes as input a **thread** and a **predicate**, and outputs two nodes (.output and .previous) of the **thread**. The first of these nodes (.output) satisfies the **predicate**, and the other (.previous) is the predecessor of .output in the **thread**:

IOspec **internal-thread-find** / .universe(**thread**) .criterion(**predicate**)
=> .output(**object**) .previous(**object**)
extension **digraph-find**
preconditions ∃x [node(.universe,x) ∧ apply(.criterion,x)=true
∧ ¬root(.universe,x)]
postconditions successor(.universe,.previous,.output)

Internal-thread-find is defined as an *extension* of another primitive action **digraph-find**:

IOspec **digraph-find** / .universe(**digraph**) .criterion(**predicate**) =>
.output(**object**)
preconditions $\exists$x [node(.universe,x) $\wedge$ apply(.criterion,x)=true]
postconditions node(.universe,.output)
$\wedge$ apply(.criterion,.output)=true

The overlay **trailing-generation+search->find** expresses the fact that an instance of the **trailing-generation+search** plan described above can be used to implement (or alternatively, can be viewed as) an **internal-thread-find** operation. It too is expressed in the frame-like notation:

Temporal Overlay **trailing-generation+search->find** :
trailing-generation+search -> **internal-thread-find**
correspondences
generator->digraph(temporal-iterator(trailing-generation+search))
= **internal-thread-find**.universe
$\wedge$ **trailing-generation+search**.exit.if.criterion=
internal-thread-find.criterion
$\wedge$ **trailing-generation+search**.exit.end.output=
internal-thread-find.output
$\wedge$ **trailing-generation+search**.exit.end.two=
internal-thread-find.previous
$\wedge$ **trailing-generation+search**.action.in=**internal-thread-find**.in
$\wedge$ **trailing-generation+search**.exit.out=**internal-thread-find**.out

The new extra slot here is *correspondences*, stating the correspondences between the various inputs to the **internal-thread-find** operation, and the various parts of the **trailing-generation+search** plan. The last five of these correspondences are straightforward, but the first is rather more involved and needs commenting on. It states that:

generator->digraph(temporal-iterator(trailing-generation+search))
= **internal-thread-find**.universe

which makes use of two overlays - the temporal overlay **temporal-iterator**, and the data overlay **generator->digraph**. **Temporal-iterator** (see below for a formal definition) enables one to view an **iterative-generation** plan as an **iterator** data plan, in which the seed of the **iterator** is the initial input to the **iterative generation**, and the function of the **iterator** is the function which is iteratively applied in the **iterative-generation** plan. The data overlay **generator->digraph** enables one to view a **generator** (of which an **iterator** is a special case) as a **digraph** (of which a **thread** is a special case). In this case (that of **iterators**) it reduces to the **iterator->thread** overlay discussed earlier. So this correspondence essentially captures the notion that when we wish to view a **trailing-generation+search** plan as implementing an **internal-thread-find** operation, the **thread** involved is that obtained by viewing the temporal stream of objects produced by the **generation** part of the plan as constituting a **thread**.

Temporal Overlay **temporal-iterator : iterative-generation** -> **iterator**
correspondences **iterative-generation**.action.input=**iterator**.seed
∧ **function->binrel(iterative-generation**.action.op)=**iterator**.op

Finally, for completeness, we give the definitions of the other basic types, relations and functions used in the above plan definitions:

Type **function**
subtype **object**

Type **predicate**
subtype **function**
definition instance(**predicate**,F) ≡
[instance(**function**,F) ∧ range-type(F)=**boolean]**

Test **@predicate** / .criterion(**predicate**) .input(**object**)
condition apply(.criterion,.input)=true

Type **many-to-one**
subtype **binrel**
definition instance(**many-to-one**,R) ≡ [instance(**binrel**,R)
∧ ∀xyz[binapply(R,x,y)=true ∧ binapply(R,x,z)=true
⊃ y=z]]

Data Overlay **function->binrel** : **function** → **many-to-one**
definition R=**function->binrel**(F,s) ≡
∀xy[apply(**function**(F,s),x)=y ↔ binapply(R,x,y)=true]

Type **binfunction**
subtype **object**

Type **binrel**
subtype **binfunction**
definition instance(**binrel**,F) ≡
[instance(**binfunction**,F) ∧ binrange-type(F)=**boolean**]

3.2.2 The Underlying Logic

In the above section Rich's notation for defining plans in the plan calculus has been demonstrated. Hayes [1979] showed how to give a formal semantics to frame-like systems using logic, and indeed Rich's semantics for the above frame-like notation uses a very similar technique. However, rather than using standard first order predicate logic, he uses a situational calculus similar to that of Green [1969] and McCarthy and Hayes [1969]. This is a first order logical language with all the usual paraphernalia associated with these i.e. symbols for functions and relations and constants, the usual logical connectives and quantifiers, and with equality. It also includes set theory (via ∈ and ∉), and integer arithmetic. It could easily be extended to include real arithmetic. Examples of (an abbreviated form of) the language have already been met above in various slots (such as the *preconditions* and *postconditions* slots) of the plan frames.

3.2.3 Semantic Domains and Behaviour Functions

The key to understanding the axiomatisation of the various plans and overlays is the notion of behaviour functions. Because we wish to be able to talk about viewing a data object as the implementation of some other (often more abstract) data object we wish to be able to specify which properties of the object we are currently interested in. Furthermore, since most interesting programming languages enable one to manipulate pointers, and we wish to be able to switch from the point of view in which we regard a pointer (in Lisp say) as pointing to a dotted pair, to the point of view in which we regard the pointer as pointing to a list, again we need to be able to specify which behaviour (or properties) of the pointer we are interested in. Both of these requirements are met by the introduction of behaviour functions.

The basic idea behind this is that the 'real' objects we are interested in are abstract mathematical objects such as integers, sets, sequences, functions, graphs, lists (thought of as a singly recursive data structure without regard to implementation details) and so on. Programs are concerned with manipulating values of one type or another. These values are either constant objects (e.g. the integer 3 appearing as a literal in a program, or the empty list), or are the values of variables, or are values stored in data structures, or are functions of other values. Because programs (in imperative languages) have *state* the behaviour of a program at a given point in its execution (a *situation*) is determined by the behaviour of the objects stored in variables and data-structures in that situation (given that constants always behave the same way). So we wish to have a clear way of talking about the behaviours of objects at different points in time. In order to do this it is necessary to distinguish carefully between the identity of an object and its behaviour in order to enable us to say that the *same* object behaves *differently* at different points in time. So let the set of object identities be **N**, and let **U** be the space of possible behaviours. So **N** consists of such things as:

a) a set of objects ('cells' in memory) for each record type defined in the program. These should be thought of as constant object identities, reflecting the fact that a given record is the <u>same</u> record, even if the values in it change over time.

b) a set of function identities, reflecting the fact that we will sometimes wish to talk about function objects whose behaviour changes over time. As will be seen later, field access functions for record types will come into this category, and if we model arrays as mutable function objects then the array identities (perhaps thought of as locations in memory) come into this category

while **U** consists of such things as:

a) everything in **N**

b) the set of integers

c) the set of sequences (thought of as functions from integers to other values)

d) the set of lists (thought of as abstract recursively defined mathematical entities)

e) a set of functions

f) a set of binary functions

g) a set of records, corresponding to 'filled in' record cells i.e. the set of possible values a cell of the appropriate type could in principle take on.

h) a set of threads as discussed earlier.

i) a set of pointers such that for each record 'cell' there is a pointer to it.

In addition **U** has subsets corresponding to all the data types defined in the plan library (e.g. iterators, directed graphs, etc.). Note that **N** is a subset of **U**.

N should be thought of as a set of 'names' for mutable objects[2]. For example a record 'cell' should be thought of as the name of an actual record. The fact that we can alter the contents of the fields of a record cell then corresponds to the fact that the 'record behaviour' of the cell can change even though the cell itself does not.

Rich[1981] calls the various subdomains of **U** Behaviour Types. Also defined on **U** is a set of type predicates (one for each behaviour type) such as isinteger, isbinfunction etc., which return true on objects of the appropriate type and false otherwise.

Note that the set of functions includes the function '^', which maps a pointer to a record cell to the cell concerned, and the set of binary functions includes such things as '+'. The set of functions also includes the field accessing functions (treated as mutable - see below) for each record type.

[2] Note that our discussion of this differs from that given in Rich[1981] in that we are treating identities as being different from pointers. This is because Rich's ideas were largely developed in the context of Lisp, in which all mutable objects are represented by pointers to the objects i.e. the identity of the object is given by a pointer to it. In Pascal it is possible to have a variable which is a record (say) i.e. the identity of the record is carried by the cell concerned, and this is different to the case where we have a pointer to the cell. So, in our listelement example, in Pascal it is possible to have the following declarations:

```
        var x: listelement;
        var y:  ^listelement
```

and these have to be handled differently. This situation is not possible in Lisp.

This means that a declaration like the following:

```
type listelement = record
                        numb : integer;
                        next : ^listelement;
                     end;
        plist = ^listelement;
```

results in **N** containing a set of listelement cells, and **U** contains the set of pointers to these cells (one pointer for each cell) and the set of listelement records (corresponding to all possible values the record cells can have considered as records).

Let **S** denote a set of situations (times). **S** is totally ordered, since we are dealing with sequential computations. The ordering is given by a primitive relation <u>precedes</u>, which is essentially defined on the actions representing a program by the sequential nature of the program. In order to be able to talk about actions that are never performed we introduced the situation ⊥. ⊥ is a bottom element of this ordering i.e.

$$\forall s[\text{precedes}(s,\bot)]$$

We can now define the binary relation cflow by:

$$\text{cflow}(s,t) \equiv \text{precedes}(s,t) \wedge [s{=}\bot \leftrightarrow t{=}\bot]$$

Now introduce a function:

$$\mathbf{BEHAVIOUR}\text{: } \mathbf{U} \times \mathbf{S} \rightarrow \mathbf{U}$$

which maps objects (and object identities) in a given situation (time) to their behaviours (values) in that situation (time). The first thing to note about this is that:

$$\forall xs[\ [x \in \mathbf{U} \wedge x \notin \mathbf{N}] \supset \mathbf{BEHAVIOUR}(x,s)=x]$$

since elements of **U** which are not 'names' are abstract constant mathematical objects. For instance we cannot alter a pointer - a variable whose value is a pointer can alter, but the pointer itself cannot, since, considered as a 'cell', the object the pointer is pointing to cannot change so that it is a different cell. Similarly an integer is always the same integer, so its behaviour does not change. However, for an object in **N**, **BEHAVIOUR** maps the object to the appropriate object representing its primitive behaviour at the given time. For example, if x is a listelement cell, then **BEHAVIOUR**(x,s) would be the listelement record object (in **U**) corresponding to the cell filled in with whatever values it has in it at time s. This enables us to express the fact that a listelement record (corresponding to a listelement cell x) is different in situations s and t by:

BEHAVIOUR(x,s) ≠ **BEHAVIOUR**(x,t)

Secondly, this enables us to express the fact that a given object x is an integer at some time s by:

isinteger(**BEHAVIOUR**(x,s))=true

Similarly, we can express the fact that an object x is a listelement record at time s by:

islistelement(**BEHAVIOUR**(x,s))=true

and in general we can express the fact that an object x has behaviour T at time s by:

isT(**BEHAVIOUR**(x,s))=true

For each behaviour type T in U (with associated predicate isT) we can now define a *behaviour function* **T**: **U** × **S** → **U** by

$$\forall xys[\ \mathbf{T}(x,s)=y \leftrightarrow$$
$$[[\mathbf{BEHAVIOUR}(x,s)=y \wedge isT(y)] \vee$$
$$\exists z[\mathbf{BEHAVIOUR}(x,s)=z \wedge \neg isT(z) \wedge y=\mathit{undefined}]]]$$

So to express the fact that the behaviour of some object x at time s is an integer it is sufficient to write:

$\mathbf{integer}(x,s) \neq \mathit{undefined}$

and to express the fact that the behaviour of x at time s is a listelement record we can write:

$\mathbf{listelement}(x,s) \neq \mathit{undefined}$

and in general, if we want to express the fact that an object has behaviour T we can write:

instance(x,T)

as a shorthand for

$\exists ys[\mathbf{T}(y,s)=x \wedge x\neq \mathit{undefined}]$

Now a programming language only provides the programmer with a set of facilities for creating and manipulating a fairly small set of data types. All others must be implemented in terms of these basic types. For instance, the subset of Pascal that we are considering provides one with numbers, arrays, records, and pointers, and other types such as lists, directed graphs, threads, and sets must be built up out of these basic ones. Furthermore, all the basic operations of the language only operate on objects of these basic types. So at the surface plan level i.e. before any analysis of the function of the program, only 'these primitive data types can occur. If we wish to view a program as operating on other data types not directly supported by the programming language we need a mechanism for viewing the primitive data objects as implementing the more abstract ones.

In programming languages primitive objects are normally either represented directly in some way (e.g. integers are normally represented as binary integers, a record 'cell' is a collection of contiguous memory locations), or they are represented by a pointer to a direct representation of the object. Now, compound objects (i.e. those with components) and objects pointed to by pointers have the property that they are <u>mutable</u> [Rich 1981]. In the case of pointers for instance, this means that it is possible to change the object pointed at (by side-effect), without changing the pointer itself. Furthermore, as already stated above, if this is done it may not affect all views of the pointer. For instance, suppose P is a pointer to a list constructed out of linked records. Then, if we change the second element of the list by simply updating the appropriate field of the second record in the list, then three points should be noted:

(i) P itself is unchanged

(ii) The <u>record</u> pointed to by P is unchanged

(iii) The <u>list</u> pointed to by P has changed

In order to deal with this we need to specify what view of an object we are interested in, and at what time (in what situation). We also need to be able to specify the connection between the records involved and the list built up out of them. Behaviour functions enable us to formalise all of this. So far we have only discussed the basic behaviour functions. Data overlays correspond to more complex behaviour functions built up out of other behaviour functions.

So consider the following piece of Pascal code:

```
type listelement = record
                        numb : integer;
                        next : ^listelement;
                   end;
     plist = ^listelement;
```

For the moment this can be taken as defining a behaviour function **listelement** which maps objects of type listelement at a given time to themselves, and listelement cells to their record behaviour (at that time), and if at a given time p is a pointer to a listelement (p is of type plist) then the behaviour function **^listelement** maps p to itself (at that time). More formally, the above declaration defines the following (recursively defined) datatypes and behaviour functions:

DataPlan **listelement**
Roles numb(**integer**) next(**^listelement**)

and

DataType **^listelement**
Definition y = **^listelement**(x,s) ↔ [x=nil ∧ y=nil] ∨
∃z [apply(^,y)=z ∧ z ≠ *undefined* ∧ **listelement**(z,s) ≠ *undefined*]

So, in the following piece of code:

```
var x, y :plist;
begin
     new(x); /*value of x is a pointer p1 to a record
                    cell rec1(say)*/
     x^.numb:=1;
     new(y); /*value of y is a pointer p2 to a record
                    cell rec2(say)*/
     y^.numb:=2;
     y^.next:=nil;
     x^.next:=y;         /*situation s1*/
     .
     .
     .
     y^.numb:=3;         /*situation s2*/
     .
     .
     .
end;
```

we see that the following behaviour holds in situation s1:

^listelement(p1 ,s1)= p1 where apply(^,p1)=rec1

^listelement(p2 ,s1)= p2 where apply(^,p2)=rec2

listelement(rec1 , s1) = <listelement 1, p2>

listelement(rec2 , s1) = <listelement 2, nil>

while in situation s2 we have the following:

^listelement(p1 ,s2)= p1 where apply(^,p1)=rec1

^listelement(p2 ,s2)= p2 where apply(^,p2)=rec2

listelement(rec1 , s2) = <listelement 1, p2>

listelement(rec2 , s2) = <listelement 3, nil>

From this it can clearly be seen that x and y themselves (the pointers) have not changed between situations s1 and s2 and that the **listelement** behaviour of the object x is pointing to has not changed. This deals with properties (i) and (ii) above. To deal with property (iii) above we need to introduce an overlay which states the connection between the **listelement** behaviour of an object and the **list** behaviour of the same object. We also need an overlay which describes the connection between **^listelement** behaviours and **list** behaviours. Overlays will be discussed more fully below, but for the moment it is sufficient to observe that this can be formalised by defining overlays **listelement->list** and **^listelement->list** as follows:

Data Overlay **listelement->list** : **listelement** → **list**
    Definition α=**listelement->list**(p,s) ≡
        [[instance(list, α) ∧ head(α)=numb(**listelement**(p,s)) ∧
            tail(α)=**^listelement->list**(next(**listelement**(p,s)),s)]]

and

Data Overlay **^listelement->list** : **^listelement** → **list+nil**
Definition α=**^listelement->list**(p,s) ≡
[[α=nil ∧ p=nil] ∨
 [instance(list, α)
 ∧ head(α)=numb(**listelement**(apply(^, **^listelement**(p,s)),s))
 ∧ tail(α)=**^listelement->list**(next(**listelement**(apply(^,
 ^listelement(p,s)),s)),s)]]

This defines a mapping from objects with **listelement** or **^listelement** behaviour to objects with **list** behaviour. From this it can be seen that:

^listelement->list(p1 ,s1) = w where

head(w)=numb(**listelement**(apply(^, **^listelement**(p1,s1)),s1))

=numb(**listelement**(apply(^, p1),s1))

=numb(**listelement**(rec1,s1))

= 1,

and

tail(w)=**^listelement->list**(
next(**listelement**(apply(^,**^listelement**(p1,s1)),s1)),s1)

= **^listelement->list**(next(**listelement**(apply(^,p1),s1)),s1)

= **^listelement->list**(next(**listelement**(rec1,s1)),s1)

= **^listelement->list**(p2,s1)

= w1

where

head(w1)= numb(**listelement**(apply(^, **^listelement**(p2,s1)),s1))

= numb(**listelement**(apply(^, p2),s1))
(since p2=**^listelement**(p2,s1))

= numb(**listelement**(rec2,s1))

= 2

and

tail(w1)

=^**listelement->list**(
next(**listelement**(apply(^,^**listelement**(p2,s1)),s1)),s1)

= ^**listelement->list**(next(**listelement**(apply(^,p2),s1)),s1)

= ^**listelement->list**(next(**listelement**(rec2,s1)),s1)

= ^**listelement->list**(nil,s1) =nil

which is equivalent to stating that

^**listelement->list**(x ,s1)= [1 2]

in more conventional notation. In a similar manner it can be deduced that

^**listelement->list**(x,s2) = [1 3],

and hence point (iii) above can be dealt with.

A subtle point now arises. The account just given is not how this behaviour will be recognised in practice since it involves defining overlays from user defined data types to the built in standard types. Because we cannot know in advance what data-types users will define, nor how they will use them, this cannot easily be done. Instead we will switch our point of view from one in which we regard the field accessing functions (numb and next in this case) as <u>roles</u> of the <u>behaviours</u> of (mutable) objects, to one in which we regard them as <u>mutable functions</u> operating directly on (now immutable) listelement cell objects. Let us denote the role selecting functions (operating on behaviours) by numb and next, and denote the corresponding mutable functions by NUMB and NEXT respectively. Then the relationship between these two views is given by:

$$\forall ps\ \mathrm{apply}(\mathbf{function}(NUMB,s),p)=\mathrm{numb}(\mathbf{listelement}(p,s))$$

$$\forall ps\ \mathrm{apply}(\mathbf{function}(NEXT,s),p)=\mathrm{next}(\mathbf{listelement}(p,s))$$

This allows us to no longer treat listelement cells as names (in **N**), but as constant immutable objects (in **U**) with fixed primitive **listelementcell** behaviour, with associated type predicate islistelementcell. This amounts to redefining **BEHAVIOUR** so that if **x** is a listelement cell then:

$$\forall s[\ \mathbf{BEHAVIOUR}(x,s)=x]$$

and to defining the behaviour function **listelementcell** by:

$$\begin{aligned}&\forall xys[\ \mathbf{listelementcell}(x,s)=y \leftrightarrow\\ &\quad [[\mathbf{BEHAVIOUR}(x,s)=y \wedge \mathrm{islistelementcell}(y)] \vee\\ &\qquad \exists z[\mathbf{BEHAVIOUR}(x,s)=z \wedge \neg \mathrm{islistelementcell}(z) \wedge y=\mathit{undefined}]]]\end{aligned}$$

This means that the behaviour function **listelement** as we defined it earlier now yields *undefined* on listelement cells. Instead we now need to define an overlay **listelementcell->listelement** by:

Data Overlay **listelementcell->listelement**: **listelementcell**→ **listelement**
Definition α=**listelementcell->listelement**(p,s) ≡
instance(α,listelement)
∧ numb(α)=apply(**function**(NUMB,s), **listelementcell**(p))
∧ next(α)=apply(**function**(NEXT,s), **listelementcell**(p))

giving us:

$$\forall ps\ \mathrm{apply}(\mathbf{function}(NUMB,s),p) = \mathrm{numb}(\mathbf{listelementcell\text{-}\!>\!listelement}(p,s))$$

$$\forall ps\ \mathrm{apply}(\mathbf{function}(NEXT,s),p) = \mathrm{next}(\mathbf{listelementcell\text{-}\!>\!listelement}(p,s))$$

and the whole of our earlier analysis can now be redone in this framework, giving the same results (provided we also redefine the overlays and datatypes given earlier so that they too are based on

listelementcell behaviour). This also means that the only primitive mutable objects are now the mutable access functions and mutable functions corresponding to arrays. Arrays will be discussed further later on. However, it should be noted that if a compound data object depends in any way on mutable objects in its definition, then it too will be mutable.

This approach has several advantages. First of all, as will be discussed in Chapter 6, it enables the translation process to be more accurate in reflecting the true data flows in the presence of compound data objects, since the objects are now not viewed as having changed. Instead record updates can now be modelled as **#newarg** operations with the (mutable) record accessing functions being changed. Secondly, since reasoning about side-effects and structure sharing is in general very difficult (even though the above formalisation enables one to do it) it is intended that common techniques utilising side effects should be captured in the plan library. This turns out to be much easier using the mutable function approach since it avoids the necessity for defining overlays from user defined data types to the standard more abstract ones. This is because, provided the mutable functions and the pattern of accesses and updates in which they are used, match suitable plans in the library, the functions and the pointers often turn out to directly implement abstract data types. So for instance, in order to obtain a **thread** we need a set of objects (the pointers) and two functions (a successor function, and a labelling function). By virtue of the plans used it is often easy (indeed automatic given our parsing approach described in Chapters 5 and 7) to recognise that the functions NEXT and NUMB are playing these roles. Once one has recognised that there is a **thread** in the program, existing overlays (such as **thread->list**) can replace the special purpose overlay introduced above. This subject will be discussed further in Chapter 7.

It should be noted that we now really have two sorts of behaviour functions. These are:

a) primitive behaviour functions defined in terms of **BEHAVIOUR**. Any such behaviour **T** satisfies:

$\forall xy[[\exists s[\mathbf{T}(x,s)=y]] \supset \forall t[\mathbf{T}(y,t)=y]]$

b) non-primitive behaviour functions, defined in terms of the primitive ones. These are all overlays of the form **B->C** where **B** and **C** are primitive behaviour functions. These all satisfy:

$\forall xy[[\exists s[\mathbf{B\text{->}C}(x,s)=y]] \supset \forall t[\mathbf{C}(y,t)=y]]$

One further point needs to be made here. For reasons which will become apparent when we discuss the parsing process, values in **U-N** will be called tie-points. So a tie-point t1 which is an object of type T will satisfy:

$\forall s\ [\mathbf{T}(t1,s)=t1]$

The interpretation of this is that a tie-point is a constant value (in **U**) with behaviour **T** (i.e. is of type T). Note that this does not mean that all views of this tie-point are constant. For instance, in our listelement example above,

^listelementcell(p1,s1) = **^listelementcell**(p1,s2)

and are represented by the same tie-point, but

^listelementcell->list(p1,s1) ≠ **^listelementcell->list**(p1,s2)

are different, not only from each other, but also from **^listelementcell**(y,s1), and should be represented by two new tie-points. So if an action produces some value (primitive behaviour of some object) as its output, this will be a tie-point, and this tie-point will be an input to any other operation which takes that value as input. In the graph representing the surface plan for a program tie-points will be represented as a small filled-in circle on the data-flow arc from the action which produced it. Data flow arcs will go from the tie-point, to the inputs of any operations using it, representing the fact that the relevant value has not changed between the output situation of the 'producing' action, and the input situation of the 'consuming' action. If the recognition process demands that we take another view of a tie-point then new tie-points representing these new behaviours will be introduced as the parsing proceeds. The problem of properly connecting these new tie-points to operations producing and using them will be dealt with by overlays, as explained in Chapter 7.

3.2.4 Axiomatising Operations, Plans, and Overlays

All the various operations, plans, and overlays are axiomatised within the plan calculus. There is an algorithmic procedure for going from the frame notation described above, to sets of axioms for each plan etc. The results of this procedure will simply be illustrated by example here, since full details can be found in Rich [1981].

3.2.4.1 Data Plans

Data plans are axiomatised in two parts. The first of these is an axiom telling us when two instances of a data plan are equal. The second is an axiom telling us under what circumstances we have an instance of the data plan, and when we have one what properties it has. So, for the data type list, whose compact specification is:

Data Plan **list**
    roles .head(**object**) .tail(**list+nil**)

we obtain as the first axiom:

$$\forall\alpha\beta pqs[[\alpha=\text{list}(p,s) \wedge \beta=\text{list}(q,s) \wedge \text{head}(\alpha)=\text{head}(\beta) \wedge \text{list}(\text{tail}(\alpha),s)=\text{list}(\text{tail}(\beta),s)] \supset \alpha=\beta\ ]$$

and for the second axiom we have:

$$\forall xys[\ [\ x\neq\textit{undefined} \wedge [y=\text{nil} \vee \text{list}(y,s)\neq\textit{undefined}]\ ] \leftrightarrow \exists\alpha p[\alpha=\text{list}(p,s) \wedge \alpha\neq\textit{undefined} \wedge \text{head}(\alpha)=x \wedge \text{tail}(\alpha)=y]\ ]$$

3.2.4.2 Data Overlays

Data overlays give rise to two "totality" axioms and a formal definition. The totality axioms guarantee the existence of appropriate data objects, and the formal definition defines the relationships between them. An example is:

Data Overlay **function->binrel : function → many-to-one**

$$\textit{definition}\ R=\textbf{function->binrel}(F,s) \equiv \forall xy[\text{apply}(\textbf{function}(F,s),x)=y \leftrightarrow \text{binapply}(R,x,y)=\text{true}]$$

which enables us to view a function as a binary relation in which two objects are related if the function applied to the first object is the second object. The totality axioms are:

$$\forall xs[\textbf{function}(x,s)\neq\textit{undefined} \supset \exists y[y=\textbf{function->binrel}(x,s)]]$$

$$\forall ys[\textbf{many-to-one}(y,s)\neq\textit{undefined} \supset \exists x[y=\textbf{function->binrel}(x,s)]]$$

and the definition of **function->binrel** is essentially that given in the *definition* slot of the compact notation:

$$R=\textbf{function->binrel}(F,s) \equiv \text{instance}(\text{many-to-one},R) \wedge \forall xy[\text{apply}(\textbf{function}(F,s),x)=y\leftrightarrow\text{binapply}(R,x,y)=\text{true}]$$

3.2.4.3 Basic Operations, Tests, and Joins

Basic operations of the calculus are defined by *IOspec* definitions. An example is the **digraph-find** operation shown again below:

IOspec **digraph-find** / .universe(**digraph**) .criterion(**predicate**) => .output(**object**)

preconditions $\exists x$ [node(.universe,x) $\wedge$ apply(.criterion,x)=true]

postconditions node(.universe,.output) $\wedge$ apply(.criterion,.output)=true

This is converted to the following two axioms:

$$\begin{aligned}
&\forall\alpha\beta\,[[\ \mathbf{digraph\text{-}find}(\alpha) \wedge \mathbf{digraph\text{-}find}(\beta) \wedge \mathrm{in}(\alpha)=\mathrm{in}(\beta)\\
&\qquad \wedge \mathrm{out}(\alpha)=\mathrm{out}(\beta) \wedge \mathrm{universe}(\alpha)=\mathrm{universe}(\beta)\\
&\qquad \wedge \mathrm{criterion}(\alpha)=\mathrm{criterion}(\beta) \wedge \mathrm{output}(\alpha)=\mathrm{output}(\beta)]\\
&\quad \supset \alpha=\beta\ ]
\end{aligned}$$

and

$$\begin{aligned}
&\forall wxuts[\quad [\mathrm{precedes}(s,t) \wedge\\
&\qquad [s\neq\perp \supset\\
&\qquad\quad [\ t\neq\perp \wedge \mathbf{digraph}(w,s)\neq\mathit{undefined}\\
&\qquad\quad \wedge \mathbf{predicate}(x,s)\neq\mathit{undefined}\\
&\qquad\quad \wedge u\neq\mathit{undefined}\\
&\qquad\quad \wedge \exists y[\quad \mathrm{node}(\mathbf{digraph}(w,s),y)\\
&\qquad\qquad \wedge \mathrm{apply}(\mathbf{predicate}(x,s),y)=\mathrm{true}]\\
&\qquad\qquad \wedge \mathrm{node}(\mathbf{digraph}(w,s),u)\\
&\qquad\qquad \wedge \mathrm{apply}(\mathbf{predicate}(x,s),u)=\mathrm{true}]]]\\
&\quad \leftrightarrow\ \exists\alpha[\ \mathbf{digraph\text{-}find}(\alpha) \wedge \mathrm{in}(\alpha)=s \wedge \mathrm{out}(\alpha)=t\\
&\qquad\quad \wedge \mathrm{universe}(\alpha)=w \wedge \mathrm{criterion}(\alpha)=x\\
&\qquad\quad \wedge \mathrm{output}(\alpha)=u\ ]\ ]
\end{aligned}$$

The first of these axioms simply tells us under what conditions we can consider two instances of a **digraph-find** operation to be equal - they are equal iff their input and output situations are equal and if their input roles are equal, and their output roles are equal. The second axiom is more interesting. It tells us that if we have appropriate data objects satisfying all the various constraints, then there exists a

digraph-find operation which relates all these objects in the appropriate way. Alternatively, if we have a **digraph-find** operation in a plan we can use this axiom to deduce the properties of the data objects involved.

The axiomatisation of tests will also be illustrated by example. Consider the test **@predicate**, whose compact definition is:

Test **@predicate** / .criterion(**predicate**) .input(**object**)
    condition apply(.criterion,.input)=true

Tests have three situations associated with them, their input situation(in), their succeed situation(succeed) and their fail situation(fail). The above definition is converted to the following axioms:

$$\begin{array}{l}
\forall\alpha\beta\ [[\ \textbf{@predicate}(\alpha) \wedge \textbf{@predicate}(\beta) \wedge \text{in}(\alpha)=\text{in}(\beta) \\
\qquad \wedge\ \text{fail}(\alpha)=\text{fail}(\beta) \wedge \text{succeed}(\alpha)=\text{succeed}(\beta) \\
\qquad \wedge\ \text{criterion}(\alpha)=\text{criterion}(\beta) \wedge\ \text{input}(\alpha)=\text{input}(\beta)] \\
\qquad \supset \alpha=\beta\ ]
\end{array}$$

and

$$\begin{array}{l}
\forall wxuts[\ [\text{precedes}(s,t) \wedge \text{precedes}(s,u) \wedge [u=\perp \vee t=\perp] \\
\qquad\quad \wedge\ [s\neq\perp \supset [\ [u\neq\perp \vee t\neq\perp] \\
\qquad\qquad\qquad \wedge\ x\neq \textit{undefined} \\
\qquad\qquad\qquad \wedge\ \textbf{predicate}(w,s)\neq \textit{undefined} \\
\qquad\qquad\qquad \wedge\ [t\neq\perp \leftrightarrow\ \text{apply}(\textbf{predicate}(w,s),x)=\text{true}]]]] \\
\qquad \leftrightarrow\quad \exists\alpha[\ \textbf{@predicate}(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{succeed}(\alpha)=t \\
\qquad\qquad\qquad \wedge\ \text{fail}(\alpha)=u \wedge \text{criterion}(\alpha)=w \\
\qquad\qquad\qquad \wedge\ \text{input}(\alpha)=x\ ]\ ]
\end{array}$$

Joins are axiomatised similarly. Joins also have three situations associated with them - two input situations (succeed and fail) and an output situation(out). We will again illustrate the axiomatisation by example. So consider a **join-output** operation whose compact specification is as follows:

Join **join-output** / .succeed-input(**object**) .fail-input(**object**) => .output(**object**)

Postconditions [fail(**join-output**)$\neq\perp\supset$ output=fail-input] $\wedge$ [succeed(**join-output**)$\neq\perp\supset$ output=succeed-input]

This is converted to the following axioms:

$$\begin{aligned}&\forall\alpha\beta[[\,\textbf{join-output}(\alpha)\wedge\textbf{join-output}(\beta)\wedge\text{succeed}(\alpha)=\text{succeed}(\beta)\\&\quad\wedge\text{fail}(\alpha)=\text{fail}(\beta)\wedge\text{out}(\alpha)=\text{out}(\beta)\\&\quad\wedge\text{succeed-input}(\alpha)=\text{succeed-input}(\beta)\\&\quad\wedge\text{fail-input}(\alpha)=\text{fail-input}(\beta)\\&\quad\wedge\text{output}(\alpha)=\text{output}(\beta)]\supset\alpha=\beta\,]\end{aligned}$$

and[3]

$$\begin{aligned}&\forall wxyuts[\;[[u=\perp\vee t=\perp]\wedge[t\neq\perp\supset[s=t\wedge w=x]]\wedge[u\neq\perp\supset[s=u\wedge w=y]]]\\&\quad\leftrightarrow\exists\alpha[\;\textbf{join-output}(\alpha)\wedge\text{out}(\alpha)=s\wedge\text{succeed}(\alpha)=t\\&\qquad\wedge\text{fail}(\alpha)=u\wedge\text{output}(\alpha)=w\wedge\text{succeed-input}(\alpha)=x\\&\qquad\wedge\text{fail-input}(\alpha)=y\;]]\end{aligned}$$

Simple **joins** (i.e. with no outputs) correspond to the same axioms with all mention of succeed-input, fail-input, and output omitted. **Join-2-outputs** is also very similar.

3.2.4.4 Temporal Plans

The conversion of temporal plans to their axiomatic form will be illustrated using the **trailing-generation+search** plan discussed earlier, and whose definition is given again below:

[3] The second axiom we give here differs from that given in Rich [1981] in order to avoid possible contradictions arising when we consider the axiom for deterministic operations discussed in chapter 4. Our axiom still captures the essentials of what a join operation does.

Temporal Plan **trailing-generation+search**
extension **iterative-generation trailing-search**
roles .current(**object**) .previous(**object**) .exit(**cond**)
.action(**@function**) .tail(**trailing-generation+search**)
constraints .current=.action.output ∧ .previous=.action.input

The first axiom again gives us conditions under which two instances of a plan are equal:

$$\begin{aligned}&\forall\alpha\beta[[\ \textbf{trailing-generation+search}(\alpha) \wedge \textbf{trailing-generation+search}(\beta)\\ &\qquad \wedge\ \mathrm{current}(\alpha)=\mathrm{current}(\beta) \wedge \mathrm{previous}(\alpha)=\mathrm{previous}(\beta)\\ &\qquad \wedge\ \mathrm{exit}(\alpha)=\mathrm{exit}(\beta) \wedge \mathrm{action}(\alpha)=\mathrm{action}(\beta) \wedge \mathrm{tail}(\alpha)=\mathrm{tail}(\beta)] \supset \alpha=\beta]\end{aligned}$$

This axiom simply states that two instances of a **trailing-generation+search** plan are equal if their input and output situations are equal and if all their roles are equal.

The second axiom is again the more interesting of the two in that it specifies how the various roles of the plan are related and implicitly captures the inheritance of constraints etc. from the plans of which this one is an *extension*.

$$\begin{aligned}&\forall uvxyzst[\ [\ \exists\alpha[\ \ \textbf{iterative-generation}(\alpha) \wedge \mathrm{action}(\alpha)=x \wedge \mathrm{tail}(\alpha)=y]\\ &\qquad\qquad \wedge \exists\beta\,[\ \ \textbf{trailing-search}(\beta) \wedge \mathrm{current}(\beta)=z\\ &\qquad\qquad\qquad \wedge\ \mathrm{previous}(\beta)=u \wedge \mathrm{exit}(\beta)=v \wedge \mathrm{tail}(\beta)=y]\\ &\qquad\qquad \wedge\ z=\mathrm{output}(x) \wedge u=\mathrm{input}(x)]\\ &\qquad \leftrightarrow\ \exists\delta[\ \ \textbf{trailing-generation+search}(\delta)\\ &\qquad\qquad\quad \wedge\ \mathrm{current}(\delta)=z \wedge \mathrm{previous}(\delta)=u\\ &\qquad\qquad\quad \wedge\ \mathrm{exit}(\delta)=v \wedge \mathrm{action}(\delta)=x \wedge \mathrm{tail}(\delta)=y]]\end{aligned}$$

It is this axiom which underlies plan recognition since it states that if one has a collection of roles satisfying all the right constraints then the existence of a **trailing-generation+search** plan is guaranteed.

3.2.4.5 Temporal Overlays

These are axiomatised in a similar fashion to data overlays by means of two "totality" axioms, and a formal definition. For example:

Temporal Overlay **trailing-generation+search->find** :
trailing-generation+search -> **internal-thread-find**
correspondences
generator->digraph(temporal-iterator(trailing-generation+search))
=internal-thread-find.universe
∧ **trailing-generation+search**.exit.if.criterion=
internal-thread-find.criterion
∧ **trailing-generation+search**.exit.end.output=
internal-thread-find.output
∧ **trailing-generation+search**.exit.end.two=
internal-thread-find.previous
∧ **trailing-generation+search**.action.in=**internal-thread-find**.in
∧ **trailing-generation+search**.exit.out=**internal-thread-find**.out

corresponds to the two axioms:

$$\begin{aligned}&\forall\alpha[\textbf{trailing-generation+search}(\alpha) \supset\\ &\qquad\exists\beta[\ \textbf{internal-thread-find}(\beta)\\ &\qquad\quad\wedge\ \beta=\textbf{trailing-generation+search->find}(\alpha)]]\end{aligned}$$

$$\begin{aligned}&\forall\beta[\textbf{internal-thread-find}(\beta) \supset\\ &\qquad\exists\alpha[\ \textbf{trailing-generation+search}(\alpha)\\ &\qquad\quad\wedge\ \beta=\textbf{trailing-generation+search->find}(\alpha)]]\end{aligned}$$

and has the formal definition:

$$\begin{aligned}&\beta=\textbf{trailing-generation+search->find}(\alpha) \equiv\\ &\quad[\ \textbf{internal-thread-find}(\beta)\\ &\qquad\wedge\ \textbf{generator->digraph(temporal-iterator}(\alpha))\\ &\qquad\qquad=\text{thread}(\text{universe}(\beta))\\ &\qquad\wedge\ \text{predicate}(\text{criterion}(\text{if}(\text{exit}(\alpha))))=\text{predicate}(\text{criterion}(\beta))\\ &\qquad\wedge\ \text{output}(\text{end}(\text{exit}(\alpha)))=\text{output}(\beta)\\ &\qquad\wedge\ \text{two}(\text{end}(\text{exit}(\alpha)))=\text{previous}(\beta)\\ &\qquad\wedge\ \text{in}(\text{action}(\alpha))=\text{in}(\beta)\\ &\qquad\wedge\ \text{out}(\text{exit}(\alpha))=\text{out}(\beta)\ ]\end{aligned}$$

3.3 Plan Recognition and Inference in the Plan Calculus

As alluded to earlier, the system will use a process of plan recognition whereby instances of plans (thought of as graphs) which occur in the surface plan of programs will be explicitly represented by a single node representing the plan. These, in turn can form parts of higher level plans which will also be represented by single nodes. In this way the system will build up a high level description of the program. Now the graphs representing plans can be thought of as forming the rules for some kind of graph grammar (a flowgraph grammar) and the recognition process then becomes one of parsing the surface plan according to the grammar. Chapters 5 and 7 will describe this process in some detail. However, in order to be sure that this process has some kind of theoretical basis, it would be nice if the steps of the parsing process could be viewed as some kind of inference process, and this is exactly what the semantics just described give us. On the one hand we have the intuitive notion of programs represented as control and data flow graphs, and the recognition process is one of finding subgraphs corresponding to implementations of programming clichés. On the other hand we can view this process as inference in the plan calculus. A simple (rather artificial) example will make this clearer. Suppose we have the partial surface plan shown in Figure 3.6, and the plans and overlays shown[4] in Figure 3.7. The surface plan can be expressed as the following (partial) statement in the situational logic:

[4] Note that when we are thinking of temporal plans as *rules* of a graph grammar we will show the graph for the plan on the right-hand side of the arrow in the rule, and the left-hand side will be a "box" representing the whole plan. Overlays will be represented by rules in which the right-hand side of the rule represents the plan or operation occurring to the left of the arrow in the overlay name, and the left-hand side represents the plan or operation to the right of the arrow in the overlay name (i.e. the arrow directions are reversed in the diagrams). This is to facilitate thinking of the rules (either plans or overlays) in the normal way that one thinks of production rules in a grammar. Furthermore, we will often use the words "plan" and "rule" interchangeably.

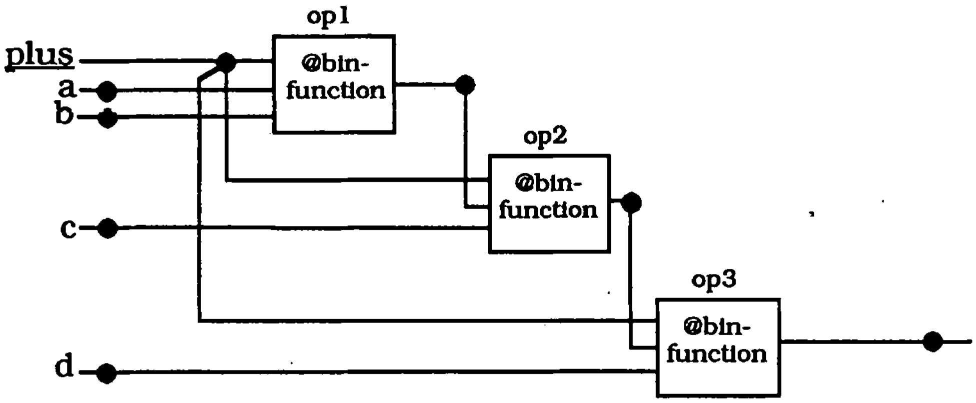
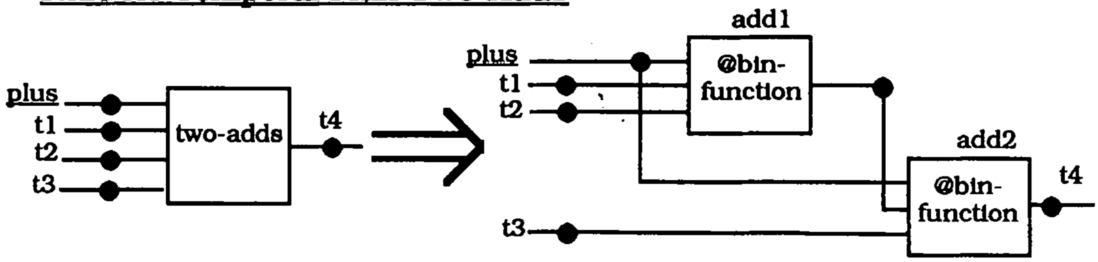
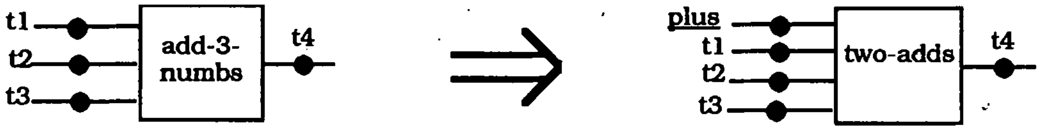


Figure 3.6 Surface Plan For Add-4-Numbs Example

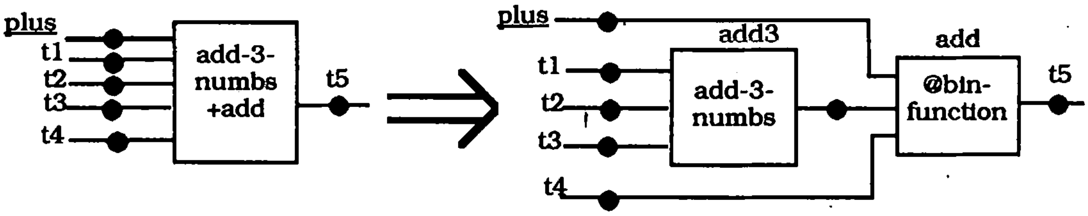
Rule for Temporal Plan **Two-Adds**



Rule for Temporal Overlay **Two-Adds->Add-3-Numbs**



Rule for Temporal Plan **Add-3-Numbs+Add**



Rule for Temporal Overlay **Add-3-Numbs+add->Add-4-Numbs**

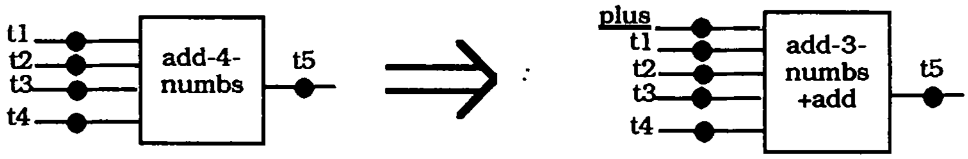


Figure 3.7 Rules and Overlays for Add-4-Numbs Example

⋮

precedes(in(op1),out(op1) ∧ ....(1)

precedes(in(op2),out(op2) ∧ ....(2)

precedes(in(op3),out(op3) ∧ ....(3)

cflow(out(op1),in(op2)) ∧ ....(4)

cflow(out(op2),in(op3)) ∧ ....(5)

@binfunction(op1) ∧..(6)

@binfunction(op2) ∧..(7)

@binfunction(op3) ∧..(8)

op(op1)=plus ∧...(9)

op(op2)=plus ∧...(10)

op(op3)=plus ∧...(11)

input1(op1)=a ∧..(12)

input2(op1)=b ∧..(13)

input1(op2)=output(op1) ∧..(14)

input2(op2)=c ∧..(15)

input1(op3)=output(op2) ∧..(16)

input2(op3)=d ∧..(17)

⋮

where a, b, c, and d have been used to denote whatever (integer) values the inputs have i.e.

a=**integer**(a,in(op1)) ∧ a≠*undefined* ∧...........................(18)

b=**integer**(b,in(op1)) ∧ b≠*undefined* ∧...........................(19)

c=**integer**(c,in(op2)) ∧ c≠*undefined* ∧...........................(20)

d=**integer**(d,in(op3)) ∧ d≠*undefined* ....(21)

We can assume (18)-(21) have been proven to follow from the behaviour of the rest of the surface plan, or can be assumed as properties of any values input to the program.

The plans, overlays, and basic actions, and their corresponding axioms are given by:

IOspec **add-4-numbs** / .input1(**integer**) .input2(**integer**)
.input3(**integer**) .input4(**integer**)
=> .output(**integer**)

Postconditions

.output=binapply(plus, binapply(plus, binapply(plus, .input1, .input2), .input3), .input4)

Axioms for add-4-numbs

$$\begin{aligned}
&\forall\alpha\beta\ [[\mathbf{add\text{-}4\text{-}numbs}(\alpha) \wedge \mathbf{add\text{-}4\text{-}numbs}(\beta) \wedge \mathrm{in}(\alpha)=\mathrm{in}(\beta) \wedge \mathrm{out}(\alpha)=\mathrm{out}(\beta)\\
&\qquad \wedge \mathrm{input1}(\alpha)=\mathrm{input1}(\beta) \wedge \mathrm{input2}(\alpha)=\mathrm{input2}(\beta)\\
&\qquad \wedge \mathrm{input3}(\alpha)=\mathrm{input3}(\beta) \wedge \mathrm{input4}(\alpha)=\mathrm{input4}(\beta)\\
&\qquad \wedge \mathrm{output}(\alpha)=\mathrm{output}(\beta)] \supset \alpha=\beta\ ]
\end{aligned}$$

$$\begin{aligned}
&\forall uwxyzts[[\mathrm{precedes}(s,t) \wedge [s\neq\bot \supset\\
&\qquad\qquad [t\neq\bot \wedge \mathbf{integer}(w,s)\neq \mathit{undefined} \wedge \mathbf{integer}(x,s)\neq \mathit{undefined}\\
&\qquad\qquad \wedge \mathbf{integer}(y,s)\neq \mathit{undefined} \wedge \mathbf{integer}(z,s)\neq \mathit{undefined}\\
&\qquad\qquad \wedge \mathbf{integer}(u,t)\neq \mathit{undefined}\\
&\qquad\qquad \wedge u=\mathrm{binapply}(\mathrm{plus},\mathrm{binapply}(\mathrm{plus},\mathrm{binapply}(\mathrm{plus},w,x),y,z)]]]\\
&\qquad\quad \leftrightarrow \exists\alpha[\mathbf{add\text{-}4\text{-}numbs}(\alpha) \wedge \mathrm{in}(\alpha)=s \wedge \mathrm{out}(\alpha)=t \wedge \mathrm{input1}(\alpha)=w\\
&\qquad\qquad\quad \wedge \mathrm{input2}(\alpha)=x \wedge \mathrm{input3}(\alpha)=y \wedge \mathrm{input4}(\alpha)=z\\
&\qquad\qquad\quad \wedge \mathrm{output}(\alpha)=u\ ]]
\end{aligned}$$

IOspec **add-3-numbs** / .input1(**integer**) .input2(**integer**)
.input3(**integer**) => .output(**integer**)

Postconditions

.output=binapply(plus, binapply(plus, .input1, .input2), .input3)

Axioms for add-3-numbs

$\forall\alpha\beta\ [[\textbf{add-3-numbs}(\alpha) \wedge \textbf{add-3-numbs}(\beta) \wedge \text{in}(\alpha)=\text{in}(\beta) \wedge \text{out}(\alpha)=\text{out}(\beta)$
$\quad \wedge\ \text{input1}(\alpha)=\text{input1}(\beta) \wedge \text{input2}(\alpha)=\text{input2}(\beta)$
$\quad \wedge\ \text{input3}(\alpha)=\text{input3}(\beta) \wedge \text{output}(\alpha)=\text{output}(\beta)] \supset \alpha=\beta\ ]$

$\forall\text{uxyzts}[[\text{precedes}(s,t) \wedge [s\neq\bot \supset$
$\quad [t\neq\bot \wedge \textbf{integer}(x,s)\neq \textit{undefined}$
$\quad \wedge\ \textbf{integer}(y,s)\neq \textit{undefined} \wedge \textbf{integer}(z,s)\neq \textit{undefined}$
$\quad \wedge\ \textbf{integer}(u,t)\neq \textit{undefined}$
$\quad \wedge\ u=\text{binapply}(\text{plus},\text{binapply}(\text{plus},x,y),z)]]]$
$\quad \leftrightarrow \exists\alpha[\textbf{add-3-numbs}(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{out}(\alpha)=t \wedge \text{input1}(\alpha)=x$
$\quad \wedge\ \text{input2}(\alpha)=y \wedge \text{input3}(\alpha)=z \wedge \text{output}(\alpha)=u\ ]]$

Temporal Plan **two-adds**
roles .add1(**@binfunction**) .add2(**@binfunction**)
constraints $.\text{add1.op}=\text{plus} \wedge .\text{add2.op}=\text{plus}$
$\wedge\ .\text{add2.input1}=.\text{add1.output}$
$\wedge\ \text{cflow}(.\text{add1.out},\ .\text{add2.in})$

Axioms for Two-Adds

$\forall\alpha\beta[[\textbf{two-adds}(\alpha) \wedge \textbf{two-adds}(\beta) \wedge \text{add1}(\alpha)=\text{add1}(\beta) \wedge \text{add2}(\alpha)=\text{add2}(\beta)]$
$\quad \supset \alpha=\beta]$

$\forall\alpha\beta[\quad [\textbf{@binfunction}(\alpha) \wedge \textbf{@binfunction}(\beta) \wedge \text{op}(\alpha)=\text{plus} \wedge \text{op}(\beta)=\text{plus}$
$\quad \wedge\ \text{input1}(\beta)=\text{output}(\alpha) \wedge \text{cflow}(\text{out}(\alpha),\ \text{in}(\beta))]$
$\quad \leftrightarrow \exists\delta[\textbf{two-adds}(\delta) \wedge \text{add1}(\delta)=\alpha \wedge \text{add2}(\delta)=\beta\ ]]$

Temporal Overlay **two-adds->add-3-numbs** : **two-adds** -> **add-3-numbs**
correspondences **add-3-numbs**.input1=**two-adds**.add1.input1
∧ **add-3-numbs**.input2=**two-adds**.add1.input2
∧ **add-3-numbs**.input3=**two-adds**.add2.input2
∧ **add-3-numbs**.output=**two-adds**.add2.output
∧ **add-3-numbs**.in=**two-adds**.add1.in
∧ **add-3-numbs**.out=**two-adds**.add2.out

Axioms and Definition for Temporal Overlay Two-Adds->Add-3-Numbs

$$\forall\alpha[\textbf{two-adds}(\alpha) \supset \exists\beta[\textbf{add-3-numbs}(\beta) \wedge \beta=\textbf{two-adds->add-3-numbs}(\alpha)]]$$

$$\forall\beta[\textbf{add-3-numbs}(\beta) \supset \exists\alpha[\textbf{two-adds}(\alpha) \wedge \beta=\textbf{two-adds->add-3-numbs}(\alpha)]]$$

Definition

$$\begin{aligned}\beta=\textbf{two-adds->add-3-numbs}(\alpha) \equiv [\quad & \textbf{add-3-numbs}(\beta)\\ & \wedge\ \text{input1}(\beta)=\text{input1}(\text{add1}(\alpha))\\ & \wedge\ \text{input2}(\beta)=\text{input2}(\text{add1}(\alpha))\\ & \wedge\ \text{input3}(\beta)=\text{input2}(\text{add2}(\alpha))\\ & \wedge\ \text{output}(\beta)=\text{output}(\text{add2}(\alpha))\\ & \wedge\ \text{in}(\beta)=\text{in}(\text{add1}(\alpha))\\ & \wedge\ \text{out}(\beta)=\text{out}(\text{add2}(\alpha))]\end{aligned}$$

Temporal Plan **add-3-numbs+add**

roles .add3(**add-3-numbs**) .add(**@binfunction**)

constraints .add.op=plus

∧ .add.input1=.add3.output

∧ cflow(.add3.out, .add.out)

Axioms for Add-3-Numbs+Add

$$\begin{aligned}&\forall\alpha\beta[[\textbf{add-3-numbs+add}(\alpha) \wedge \textbf{add-3-numbs+add}(\beta) \wedge \text{add3}(\alpha)=\text{add3}(\beta)\\ &\quad \wedge \text{add}(\alpha)=\text{add}(\beta)] \quad \supset \alpha=\beta]\end{aligned}$$

$$\begin{aligned}\forall\alpha\beta[\quad & [\textbf{add-3-numbs}(\alpha) \wedge \textbf{@binfunction}(\beta) \wedge \text{op}(\beta)=\text{plus}\\ & \quad \wedge \text{input1}(\beta)=\text{output}(\alpha) \wedge \text{cflow}(\text{out}(\alpha), \text{in}(\beta))]\\ & \leftrightarrow \exists\delta[\textbf{add-3-numbs+add}(\delta) \wedge \text{add3}(\delta)=\alpha \wedge \text{add}(\delta)=\beta\ ]]\end{aligned}$$

Temporal Overlay **add-3-numbs+add->add-4-numbs** :
add-3-numbs+add -> **add-4-numbs**
correspondences
add-4-numbs.input1=**add-3-numbs+add**.add3.input1
∧ **add-4-numbs**.input2=**add-3-numbs+add**.add3.input2
∧ **add-4-numbs**.input3=**add-3-numbs+add**.add3.input3
∧ **add-4-numbs**.input4=**add-3-numbs+add**.add.input2
∧ **add-4-numbs**.output=**add-3-numbs+add**.add.output
∧ **add-4-numbs**.in=**add-3-numbs+add**.add3.in
∧ **add-4-numbs**.out=**add-3-numbs+add**.add.out

Axioms and Definition for
Temporal Overlay Add-3-Numbs+Add->Add-4-Numbs

∀α[**add-3-numbs+add**(α) ⊃ ∃β[**add-4-numbs**(β)
∧ β=**add-3-numbs+add->add-4-numbs**(α)]]

∀β[**add-4-numbs**(β) ⊃ ∃α[**add-3-numbs+add**(α)
∧ β=**add-3-numbs+add->add-4-numbs**(α)]]

Definition
β=**add-3-numbs+add->add-4-numbs**(α) ≡ [**add-4-numbs**(β)
∧ input1(β)=input1(add3(α))
∧ input2(β)=input2(add3(α))
∧ input3(β)=input3(add3(α))
∧ input4(β)=input2(add(α))
∧ output(β)=output(add(α))
∧ in(β)=in(add3(α))
∧ out(β)=out(add(α))]

The definition of **@binfunction** is:

IOSpec **@binfunction** / .op(**binfunction**) .input1(**object**) .input2(**object**)
=> .output(**object**)
Preconditions binapply(.op, .input1, .input2)≠*undefined*
Postconditions binapply(.op, .input1, .input2)=.output

The axioms for this are:

Axioms for @binfunction

$$\begin{aligned}&\forall\alpha\beta\ [[\textbf{@binfunction}(\alpha) \wedge \textbf{@binfunction}(\beta) \wedge \mathrm{in}(\alpha)=\mathrm{in}(\beta) \wedge \mathrm{out}(\alpha)=\mathrm{out}(\beta)\\ &\qquad \wedge \mathrm{input1}(\alpha)=\mathrm{input1}(\beta) \wedge \mathrm{input2}(\alpha)=\mathrm{input2}(\beta) \wedge \mathrm{op}(\alpha)=\mathrm{op}(\beta)\\ &\qquad \wedge \mathrm{output}(\alpha)=\mathrm{output}(\beta)] \supset \alpha=\beta\ ]\end{aligned}$$

$$\begin{aligned}&\forall uxyfts[[\mathrm{precedes}(s,t) \wedge [s\neq\bot \supset\\ &\qquad\qquad [t\neq\bot \wedge x\neq \mathit{undefined}\\ &\qquad\qquad \wedge y\neq \mathit{undefined}\\ &\qquad\qquad \wedge u\neq \mathit{undefined}\\ &\qquad\qquad \wedge \textbf{binfunction}(f,s)\neq \mathit{undefined}\\ &\qquad\qquad \wedge \mathrm{binapply}(\textbf{binfunction}(f,s),x,y)\neq \mathit{undefined}\\ &\qquad\qquad \wedge u=\mathrm{binapply}(\textbf{binfunction}(f,s),x,y)]]]\\ &\qquad \leftrightarrow \exists\alpha[\textbf{@binfunction}(\alpha) \wedge \mathrm{in}(\alpha)=s \wedge \mathrm{out}(\alpha)=t \wedge \mathrm{input1}(\alpha)=x\\ &\qquad\qquad \wedge \mathrm{input2}(\alpha)=y \wedge \mathrm{op}(\alpha)=f \wedge \mathrm{output}(\alpha)=u\ ]]\end{aligned}$$

Finally we need the definition of plus itself:

Binfunction **plus** : **integer** × **integer** → **integer**
Properties instance(aggregative-binfunction, plus) ∧ identity(plus)=0

The details of the definition of an aggregative-binfunction are irrelevant here, and will be omitted for clarity, but it should be noted that the first line if this definition tells us that:

$$\forall xy[\ [\mathrm{instance}(x,\mathrm{integer}) \wedge \mathrm{instance}(y,\mathrm{integer})] \supset \mathrm{instance}(\mathrm{binapply}(\mathrm{plus},x,y),\ \mathrm{integer})]$$

The first thing to note is that we can show that this piece of surface plan performs an **add-4-numbs** operation on a,b,c, and d without recourse to anything other than the properties of plus, **@binfunction**, **add-4-numbs**, and the axiomatic formulation of the surface plan itself (i.e. without using any of the plans and overlays above). A brief (and simplified) account of how this is done is as follows:

First note that there are two cases we have to deal with - the case when in(op1)≠⊥ and the case when in(op1)=⊥. Assume that the first case holds i.e.:

in(op1)≠⊥ ....(22)

From statements 22, 1, 12, 13, 18, 19, the definition of plus, and the axioms for **@binfunction** we can deduce that

out(op1)≠⊥ ....(23)

∧ a≠*undefined* ....(24)

∧ b≠*undefined* ....(25)

∧ output(op1)≠*undefined* ....(26)

∧ plus≠*undefined* ....(27)

∧ binapply(plus, a, b)≠*undefined* ....(28)

∧ output(op1)=binapply(plus, a, b) ....(29)

∧ instance(output(op1), integer) ....(30)

From 23, and 4 we can deduce that

precedes(out(op1),in(op2)) ....(31)

in(op2)≠⊥ ....(32)

from which (in a very similar fashion) we can deduce

out(op2)≠⊥ ....(33)

∧ input1(op2)≠*undefined* ....(34)

∧ c≠*undefined* ....(35)

∧ output(op2)≠*undefined* ....(36)

∧ plus≠*undefined* ....(37)

∧ binapply(plus, input1(op2), c)≠*undefined* ....(38)

∧ output(op2)=binapply(plus, input1(op2), c) ....(39)

∧ instance(output(op2), integer) ....(40)

From 33, and 5 we can deduce that

precedes(out(op2),in(op3))..(41)

in(op3)≠⊥..(42)

from which we can deduce

out(op3)≠⊥...(43)
∧ input1(op3)≠*undefined*..(44)
∧ d≠*undefined*..(45)
∧ output(op3)≠*undefined*...(46)
∧ plus≠*undefined*..(47)
∧ binapply(plus, input1(op3), d)≠*undefined*................(48)
∧ output(op3)=binapply(plus, input1(op3), d)..............(49)
∧ instance(output(op3), integer)................................(50)

Now from 49, 39, and 29 we can deduce that

output(op3)=binapply(plus, binapply(plus,
binapply(plus, a, b), c), d)........(51)

It then follows from the axiom for add-4-numbs,the transitivity of the precedes relation, 18, 19, 20, 21, 34, and 35 that there exists an add-4-numbs operation, taking a, b, c, and d as inputs and whose output satisfies 51.

We then have to deal with the second case i.e. when in(op1)=⊥. This is rather simpler, in that the definition of precedes, ⊥, and the transitivity of precedes tells us that out(op3)=⊥. From this and the axiom for **add-4-numbs** we can deduce the existence of an **add-4-numbs** operation taking a, b, c, and d as inputs.

Note that in the above we have made a lot of implicit use of the fact that:

$\forall xy[[\exists s[\mathbf{T}(x,s)=y]] \supset \forall t[\mathbf{T}(y,t)=y]]$

for primitive behaviours **T**, of which (of course) **integer** is one. A theorem prover would of course have had to do this explicitly. Also note that we essentially had to compute (a large part of) the transitive closure of the precedes relation. If this surface plan had been much larger, possibly with intermediate actions between the actions we have shown, then the direct control flow links between the operations might not have existed, and so computing (a large part of) the transitive closure of precedes would have involved a lot more effort.

Even for these rather simple operations this is a lot of work, especially for a forward-chaining theorem prover (corresponding to bottom-up analysis) which might have thousands of axioms to deal with if the plan library and/or the surface plan were large.

On the other hand, use of the plans and overlays can greatly simplify this deductive process. Indeed, one view of the plans and overlays is that they are pre-proved lemmas which the theorem prover can then use. Using the plans and overlays, the recognition process goes like this:

From the definition of the two-adds plans, and 4, 6, 7, 9, 10, and 14 we can deduce the existence of a **two-adds** plan, consisting of op1 and op2. If we call this plan two-adds1 then we have immediately:

two-adds(two-adds1) ....(23′)

∧ add1(two-adds1)=op1 ....(24′)

∧ add2(two-adds1)=op2 ....(25′)

Using the axioms for **two-adds->add-3-numbs** we can deduce the existence of an **add-3-numbs** operation (let us call it add-3-numbs1) satisfying the following:

add-3-numbs(add-3-numbs1) ....(26′)

∧ input1(add-3-numbs1)=a...(27′)

∧ input2(add-3-numbs1)=b...(28′)

∧ input3(add-3-numbs1)=c...(29′)

∧ output(add-3-numbs1)=output(op2).........................(30′)

∧ in(add-3-numbs1)=in(op1) ....(31′)

∧ out(add-3-numbs1)=out(op2)...................................(32′)

From the axiom for the plan **add-3-numbs+add** we can now deduce the existence of an **add-3-numbs+add** plan (add-3-numbs+add1 say) satisfying:

add-3-numbs+add(add-3-numbs+add1).......................(33′)

∧ add3(add-3-numbs+add1)=add-3-numbs1 ....(34′)

∧ add(add-3-numbs1)=op3 ....(35′)

From this and the axioms for the overlay **add-3-numbs+add->add-4-numbs** we can deduce the existence of an **add-4-numbs** operation (add-4-numbs1 say), satisfying:

add-4-numbs(add-4-numbs1).......................................(36′)

∧ input1(add-4-numbs1)=a..(37′)

∧ input2(add-4-numbs1)=b..(38′)

∧ input3(add-4-numbs1)=c...(39′)

∧ input4(add-4-numbs1)=d..(40′)

∧ output(add-4-numbs1)=output(op3)..........................(41′)

∧ in(add-4-numbs1)=in(op1) ....(42′)

∧ out(add-4-numbs1)=out(op3)....................................(43′)

Note that if we then assume (or prove from other parts of the plan) that in(op1)$\neq\perp$ then we can immediately deduce from the existence of this plan that:

precedes(in(op1),out(op3))
$\wedge$ out(op3)$\neq\perp$
$\wedge$ output(op3)=binapply(plus,
binapply(plus, binapply(plus, a, b), c), d)

without having to compute the transitive closure of precedes.

It should be clear from the above that the plan calculus provides an extremely powerful technique for reasoning about programs, both from first principles, and from recognition of plans (clichés) occurring in them. Indeed the example just analysed, although simple, illustrates the ability it provides to recognise what a program does even if the programmer has used plans not in the library. For if the plans and overlays used had not been there we have showed how a plan calculus based system, using a theorem prover, could still recognise the function of the surface plan. Equally well, if the surface plan being analysed had corresponded to:

2*((a+b)/2+(c+d)/2) (say)

instead of

a+b+c+d

it could still have recognised the **add-4-numbs** operation being performed from the axiomatisations of plus, times, and divides. On the other hand we have shown how, when the programmer does use standard plans (and it is hoped that eventually the library will contain enough plans to be applicable to most programs!) this deductive process can be greatly simplified.

Although we have shown how the deductive process is simplified by the use of plans and overlays, it is still a lot of work for a standard theorem prover to do this. This is because such a theorem prover (being general purpose) cannot make use of the inherent structure present in surface plans, and the axioms corresponding to plans and overlays. This structure is made explicit in the graphical representation. So if we could write a special purpose theorem prover with knowledge of this structure (perhaps indexing and grouping the axioms and derived theorems in clever ways) then this deduction process could be made much more efficient in that it would avoid blind alleys and unnecessary work. In the deductive process above we showed none of the additional work a theorem prover might have done, but only the direct proof of the existence of the **add-4-numbs** operation. In chapters 5 and 7 we will present a graph parser, capable of recognising the occurrences of plans in the surface plan treated as a graph, by treating plans and overlays as rules of a graph grammar. Because this parser makes use of the graph-like structure of the rules and surface plan, it can be thought of as just such a special purpose theorem prover.

Complete grammatical entities found by the parser correspond to theorems that have been proved from the axioms for the plans and the "facts" describing the initial surface plan. The partial grammatical entities that are found can be viewed as partial proofs, where the information in a partial entity specifying what would need to be found in the graph in order to complete the entity corresponds to what still needs to be shown in order to complete the proof. In this case near miss recognition and repair can be viewed as being similar to Murray's [1986] TALUS system which repairs bugs by repairing a proof of the equivalence of the program to a known program, except that in this case we are repairing the proof of equivalence to a known plan.

Chapter 4.

Extensions and Modifications of the Theory.

When trying to recognise plans using a graph parser there are various situations which can cause problems. The first of these is to do with problems caused by non-standard (i.e. not in accordance with the plan library) control-flow. This can often result in code for which the graph is apparently unrecognisable as the cliché it actually represents. This is particularly so when combined with another problem that can occur. This is associated with optimisations - if a programmer has either

(i) made unforeseen optimisations involving sharing parts of two or more plans in their code, or

(ii) failed to make optimisations assumed by the plan library i.e. a plan involves some action feeding its output(s) to more than one place, but the programmer has duplicated the action instead,

then this can give rise to problems for a graph based parser as in either case the surface plan that is being analysed differs from what one would get from pure rule rewriting (treating the plan library as productions in a grammar).

4.1 Control-Flow Environments

In order to discuss the problems caused by non-standard control-flow, and the problems caused by optimisation (or the lack of it), we need to look at control flow in a bit more detail. The first thing to remember is that <u>precedes</u> is a total ordering on the situations in a program, and that $\perp$ (used to denote situations that are never reached) is a bottom element of this ordering i.e.

$$\forall s[\text{precedes}(s,\perp)]$$

Of course, which situations are not reached will vary between instances of actual computations (runs of the program) represented by the graph. This means that for most situations in the graph, we cannot say definitely that they are equal to $\perp$, or that they are not, although we can state the conditions under which they will be $\perp$. However, we can often say that a situation s will be $\perp$ if and only if some other situation t is $\perp$. For instance, all the situations occurring on the succeed side of a cond will be $\perp$ if the test fails i.e. if the succeed situation of the test is $\perp$. So we can define an equivalence relation $\approx$ on the set of situations in a program by:

$$s \approx t \equiv [s \neq \perp \leftrightarrow t \neq \perp\,]$$

We will refer to the equivalence classes under $\approx$ as control-flow environments, and say that s and t are in the <u>same control-flow environment</u> if $s \approx t$. We will write env(s) to denote the control-flow environment in which situation s occurs. Note that the form of the axiomatisation of operations:

$$\forall \ldots st[\ \ [\text{precedes}(s,t) \wedge [s \neq \perp \supset [t \neq \perp \wedge \ldots]]$$
$$\leftrightarrow \exists \alpha\,[\ \ \text{op}(\alpha) \wedge \text{in}(\alpha)=s \wedge \text{out}(\alpha)=t \wedge \ldots]\ ]$$

implies that the input and output situations of operations must always be in the same control flow environment. This means that for any operation op1, we can refer to the control flow environment env_{op1} of the operation.

If S and T denote control-flow environments, then we can define the relation subenv(S,T) by:

$$\text{subenv}(S, T) \equiv \exists\alpha[\text{test}(\alpha) \wedge \text{in}(\alpha)\in S \wedge$$
$$[\text{env}(\text{fail}(\alpha))=T \vee \text{env}(\text{succeed}(\alpha))=T]]$$

We can now define a relation <u>encloses</u> on control-flow environments by:

$$\forall ST\ [\text{encloses}(S,T) \leftrightarrow [\text{subenv}(S,T) \vee \exists C[\ \text{subenv}(S,C)$$
$$\wedge\ \text{encloses}(C,T)]]$$

and then define a relation $\leq_{CF}$ on control-flow environments by:

$$S{\leq_{CF}}T \equiv S=T \vee \text{encloses}(S,T)$$

which defines a partial ordering on control-flow environments. We will call two control-flow environments S and T <u>comparable</u> iff either $S{\leq_{CF}}T$ or $T{\leq_{CF}}S$ holds, and we will say S is the outermost of two comparable control-flow environments S and T iff $S{\leq_{CF}}T$, otherwise we will say it is the innermost. We will also denote the control-flow environment containing $\perp$ by $\perp$. It should also be noted that, as a result of the axiomatisation of tests, the above definitions imply that:

$$[\ \text{subenv}(S,T) \wedge S=\perp] \supset T=\perp$$

This in turn implies that:

$$[\ \text{encloses}(S,T) \wedge S=\perp] \supset T=\perp$$

It should also be noted that in a consistent program, if we assume that the initial situation s_0 (the start of the program) is not equal to $\perp$, and if we assume that any external files[1] used by the program contain suitable sequences of objects, then it follows from the axiomatisations of all the operations occurring in a surface plan that the only way in which a situation can equal $\perp$ is by being in a branch of a cond. As a result every control-flow environment T (apart from s_0) has a test

[1] We include terminal input and output as a type of file.

Test(T) associated with it. The form of the axiomatisation of tests also tells us that every test X (say) has some condition C(X)[2] associated with it such that

$$[\text{succeed}(X)\neq\bot \leftrightarrow [\text{in}(X)\neq\bot \wedge C(X)]]$$
$$\wedge\ [\text{fail}(X)\neq\bot \leftrightarrow [\text{in}(X)\neq\bot \wedge \neg C(X)]]$$

which is equivalent to:

$$[\text{env}(\text{succeed}(X))\neq\bot\leftrightarrow\ [\text{env}(\text{in}(X))\neq\bot\wedge\ C(X)]]$$
$$\wedge\ [\text{env}(\text{fail}(X))\neq\bot\leftrightarrow\ [\text{env}(\text{in}(X))\neq\bot\wedge \neg C(X)]]$$

We will denote env(succeed(X)) by succ(X) and env(fail(X)) by fail(X). This means that we can associate a condition Cond(T) with every control-flow environment T. If we call the initial control-flow environment (the one which contains the initial situation s_0 of the program) E_0, then we define:

$$\text{Cond}(E_0)=\text{true}$$

The condition for other control-flow environments is defined recursively by:

$$\text{Cond}(E)=\begin{cases}\text{Cond}(\text{env}(\text{in}(\text{Test}(E)))) \wedge C(\text{Test}(E)) \text{ if } E \text{ is succ}(\text{Test}(E))\\ \text{Cond}(\text{env}(\text{in}(\text{Test}(E)))) \wedge \neg C(\text{Test}(E)) \text{ if } E \text{ is fail}(\text{Test}(E))\end{cases}$$

We now note that the following relationship holds between a control-flow environment E and its condition Cond(E):

$$E\neq\bot\leftrightarrow\ \text{Cond}(E)$$

Since any operation op in the surface plan occurs in a control-flow environment this tells us that:

$$\text{env}_{op}\neq\bot\leftrightarrow\ \text{Cond}(\text{env}_{op})$$

[2] Note that C(X) is the actual condition tested for by the test, not the entire sequence of conditions which must be true for this part of the program to be reached.

We will call Cond(env_{op}) the controlling condition for the action op, and the above simply states that any operation is executed if and only if its controlling condition is true. We can now rewrite the axioms for an operation in terms of the controlling operations for the situations involved, as follows:

$$\forall \ldots st[\ \ [precedes(s,t) \wedge [Cond(s) \leftrightarrow Cond(t)] \wedge [Cond(s) \supset \ldots]]$$
$$\leftrightarrow \exists\alpha\ [\ \ op(\alpha) \wedge in(\alpha)=s \wedge out(\alpha)=t \wedge \ldots]\ ]$$

4.2 Generalised Control-Flow Environments

Up until now we have been treating control-flow environments as the objects which define their associated conditions. We will now switch viewpoint and regard the conditions as defining the control flow environments. For a condition X we define:

$$Env(X) = \{s: s \neq \perp \leftrightarrow X\}$$

This is a much more general notion since the condition X can be any condition, not just one associated with an actual sequence of tests in a program. We will refer to such environments as generalised control-flow environments, and the associated conditions will be called generalised controlling conditions. For operations in the program this switch of viewpoint also entails regarding their controlling conditions as fundamental, rather than the control-flow environments. For actions occurring in the surface plan for a program the controlling conditions are simply the ones associated with the (non-generalised) control-flow environments discussed earlier. However, it will turn out that allowing operations and plans found by the recognition process to be associated with generalised controlling conditions rather than just with the actual conditions associated with tests in the program, has far-reaching and important implications, and enables us to deal with the first two of the problems discussed above.

So, rather than associating control flow or control-flow environments with each operation we will associate a (generalised) controlling condition. Tests will get three such expressions, corresponding to their in, succeed, and fail controlling conditions, and joins will similarly get three controlling conditions. These controlling conditions are simply those associated with the appropriate (non-generalised) control flow environments. Each such condition will be a symbolic expression. Each test X, with condition C(X), is given a unique variable (e.g. Z), which denotes C(X), and so, if the symbolic expression denoting the controlling condition for in(X) is P (say), then succeed(X) and any actions and tests in the same control flow environment will get $(P \wedge Z)$, and fail(X) and any operations and tests in the same environment will get symbolic expression $(P \wedge \neg Z)$. The actions in the initial environment simply get true. This can all be done by the translation program as described in Chapter 6. The result of this process is illustrated in Figure 4.1. Note that the length of these expressions depends on the level of nesting of conditionals that occurs in the program. In most programs this is not very large, so the expressions will not be terribly long. It should also be noted that such expressions can be used in two ways - the first is to reason with them using propositional logic as will be shown later, and the second is to use them as *names* of environments. In most cases actions and so on recognised by the recognition system will actually have control conditions corresponding to the original control environments in the program (i.e. those defined by sequences of tests). In this case the relation encloses(A,B) defined earlier can be checked for by simply checking that the expression for A is a prefix of the expression for B e.g. if A has expression $X \wedge Y$ as its controlling expression, and B has

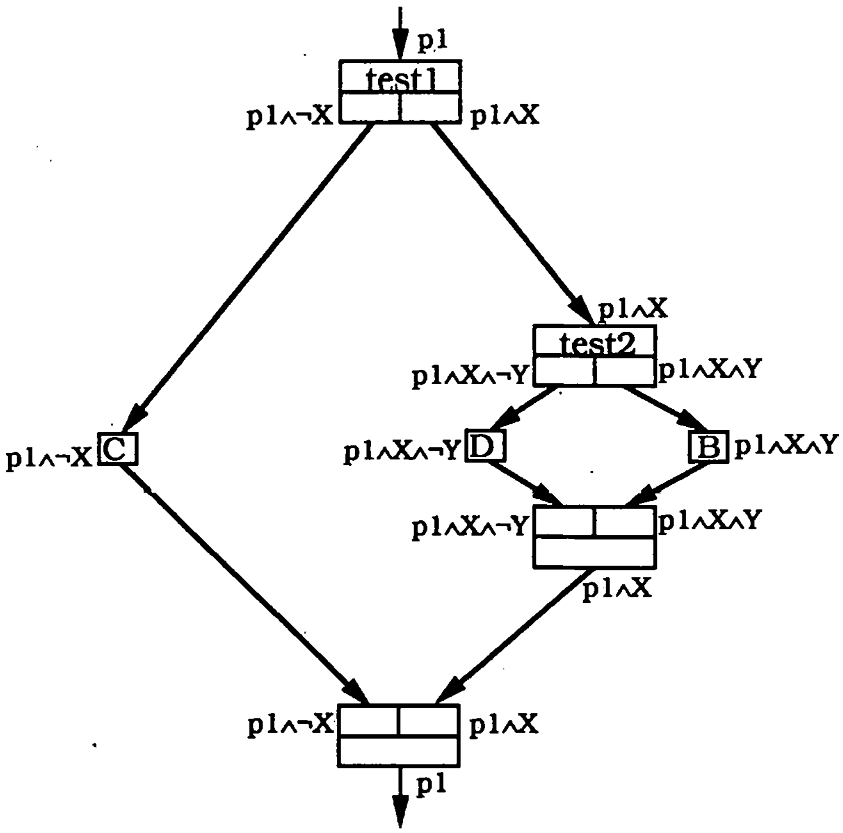


Figure 4.1 Surface Plan Annotated with Controlling Conditions

expression $X \wedge Y \wedge Z$ then it follows that encloses(A,B) holds. Note also that $X \wedge Y \wedge Z \supset X \wedge Y$ holds, and in general

$$\text{encloses}(A,B) \leftrightarrow \text{Cond}(B) \supset \text{Cond}(A)$$

holds. From now on we will treat this as the definition of encloses.

4.3 Plan Conditions and Control-Flow Environments for Plans (Simple Plans)

Now we will consider how plans are handled. We note first of all that the constraints fall into two categories. The first of these is simply the collection of data flow constraints and type information on *operations*. These are what the parser described in Chapters 5 and 7 actually checks. The second is a set of control flow constraints. These can be rewritten in the form of relationships between the control-flow environments (and hence the controlling conditions) of the actions occurring in the plan. There is actually a third class of constraints implicit in thc compact notation for plans which is made explicit in the axiomatic form of the plans - this class consists of a set of precedes relations between various situations occurring in the plan. These relationships actually come from the cflow constraints, and will be separated out as described below. Finally there is another set of constraints consisting of conditions that the data objects must satisfy (e.g. type information). Both of these last two types of constraint will be assumed to be checked for by some other mechanism (e.g. when the parser recognises a plan, the plan will be passed to a theorem prover to check these other constraints). So from now on we will assume when discussing plan recognition that the plan does not have any such constraints or, equivalently that they are all true. In Chapter 9 we will return to the subject of these other constraints.

The general form of the defining axiom for a plan P with roles $role_1 ... role_n$ is as follows:

$$\forall R_1 ... R_n \, [\, [T(R_1 ... R_n) \wedge D(R_1 ... R_n) \wedge Reqs(R_1 ... R_n\ )] \\ \leftrightarrow \exists\alpha[P(\alpha) \wedge role_1(\alpha)=R_1 \wedge ... \wedge role_n(\alpha)=R_n]]$$

where $T(R_1 ... R_n)$ represents the set of precedes information and type constraints on data objects, $D(R_1 ... R_n)$ represents the set of constraints on the types of the actions and on the data flows between them, and $Reqs(R_1 ... R_n\ )$ is the set of control flow constraints. Now what the parser actually finds is a set of operations $A_1 ... A_n$ such that $D(R_1 ... R_n)$ holds, and assuming $T(A_1 ... A_n)$ is satisfied, it now has to check that the control flow constraints $Reqs(A_1 ... A_n)$ are satisfied. Now the roles $R_1 ... R_n$ have an associated set of controlling conditions $c_1 ... c_m$[3], and it will be shown below how $Reqs(R_1 ... R_n)$ can be rewritten in terms of $c_1 ... c_m$. Now the plan instance (i.e. $A_1 ... A_n$) has an associated set of control-flow environments $env_1 ... env_m$, with associated controlling conditions $X_1 ... X_m$, which satisfy the following:

$$[env_1 \neq \perp \leftrightarrow X_1] \wedge ... \wedge [env_m \neq \perp \leftrightarrow X_m]$$

Now a plan is only valid if the following holds for it:

$$Reqs(A_1 ... A_n) \wedge [env_1 \neq \perp \leftrightarrow X_1] \wedge ... \wedge [env_m \neq \perp \leftrightarrow X_m]$$

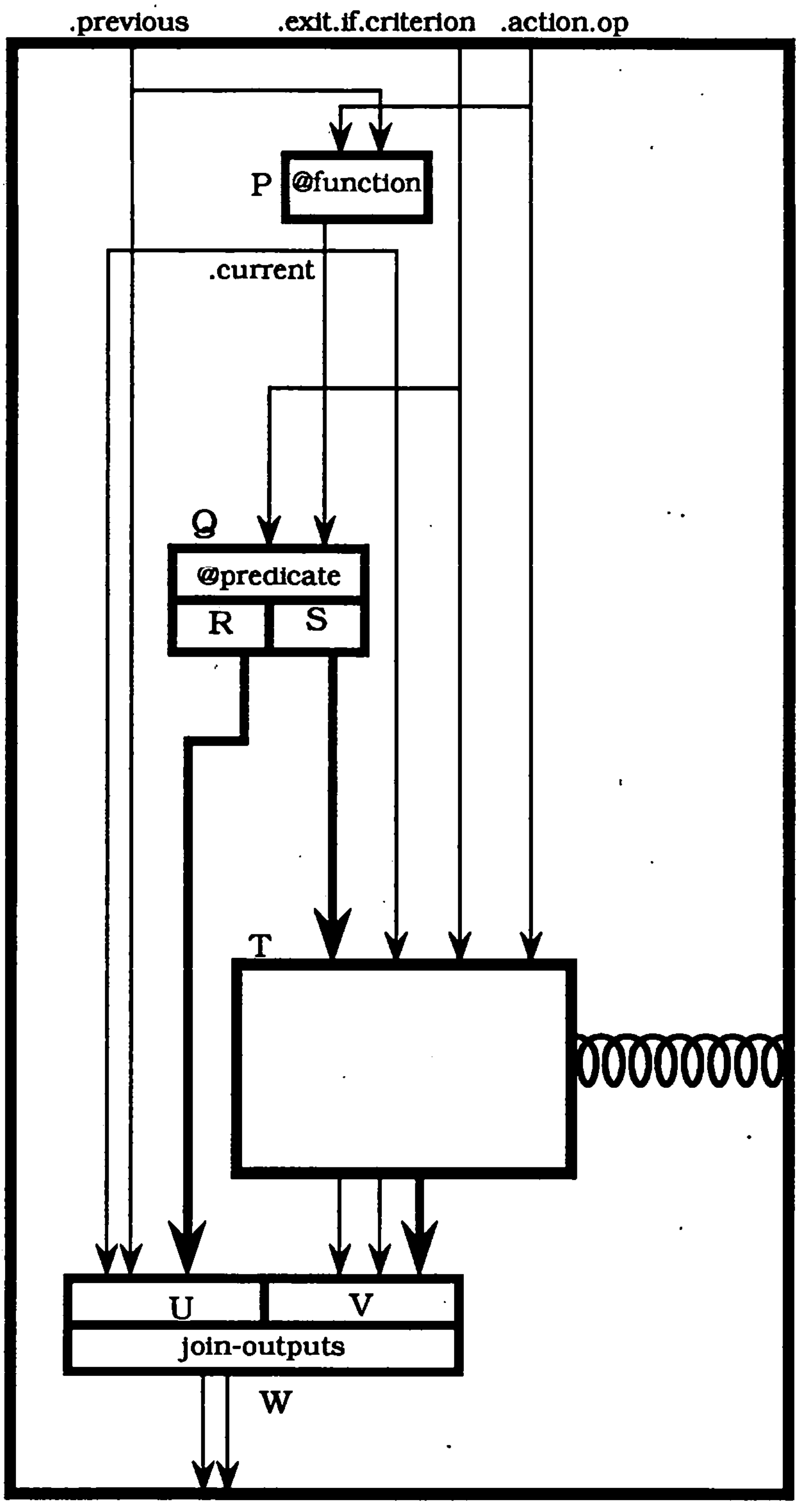
which simply states that the roles of a valid plan both execute when they do (as implied by their control-flow environments) and must satisfy the requirements of the plan. We will refer to this requirement as the consistency requirement for the plan.

[3] Note that there may be more c_i than R_i since some of the roles may be tests or joins, and these have more than one control flow environment associated with them.

So the parser must check that the consistency requirement holds for any plan it has tentatively found. It will do this by checking that a condition, known as the plan condition, is satisfiable for the plan. If the plan condition is simply true, then the plan is valid and satisfies the consistency requirement. If the plan condition is not satisfiable (i.e. the plan condition is always false for $A_1...A_n$) then $A_1...A_n$ cannot be the roles of a valid plan. If the plan condition is neither simply true or false, then we have recognised a plan conditionally. If the plan is 'executed' under circumstances when the plan condition is not true then, although (some of) the actions have been carried out, this execution instance cannot be regarded as an execution of the plan, but merely as the execution of (some of) its components. If on the other hand it is 'executed' under circumstances which make the plan condition true, then under these circumstances we can regard it as the execution of a valid plan.

For the moment we will assume that the plans we are considering are simple i.e. they have at most a single test/join (**cond**) in them. More complex plans will be discussed later. The plan condition for a simple plan is very simple to construct. Suppose we have a plan with action roles $R_1...R_n$ with corresponding control conditions $c_1...c_m$, and control flow requirements Reqs($R_1...R_n$). Note that the $c_1...c_m$ represent *variables* which denote whatever the actual controlling conditions may be when a plan is recognised in the program. Figure 4.2 illustrates this convention for **trailing-generation-and-search**. Now Reqs($R_1...R_n$) always consists of the conjunction of a set of control flow requirements. These are all of the form (although later we will introduce other forms):

cflow(s,t)



P, Q, R, S, T, U, V, and W are variables denoting actual controlling conditions when the action roles are matched against actions in a program

Figure 4.2 Trailing Generation and Search Rule Illustrating Variables Denoting Controlling Conditions

where s and t are input or output situations of some of $R_1 \ldots R_n$. This means:

$$\text{precedes}(s,t) \wedge [s \neq \perp \leftrightarrow t \neq \perp]$$

As discussed above precedes(s,t) is bundled in with $T(R_1 \ldots R_n)$, so we are left with:

$$[s \neq \perp \leftrightarrow t \neq \perp]$$

From our definition of control-flow environments and controlling conditions, we know that $[s \neq \perp \supset t \neq \perp]$ if and only if $\text{Cond}(s) \supset \text{Cond}(t)$, and as a result $[s \neq \perp \leftrightarrow t \neq \perp]$ if and only if $\text{Cond}(s) \leftrightarrow \text{Cond}(t)$. This enables us to express all the control flow constraints in terms of controlling conditions.

So all the constraints are of the form:

$$c_i \leftrightarrow c_j$$

and the plan condition is simply the conjunction of all of these coming from $\text{Reqs}(R_1 \ldots R_n)$. Now when an actual plan is found by the parser, we get actual roles $A_1 \ldots A_n$ matched against the roles $R_1 \ldots R_n$ of the plan definition. These roles have actual control conditions $X_1 \ldots X_m$ matched against $c_1 \ldots c_m$ in the rule. The plan condition for this actual plan instance is therefore the plan condition for the rule, with $c_1 \ldots c_m$ substituted for by $X_1 \ldots X_m$. It is this substitution which guarantees that if a role executes in a plan when the plan condition is true then the requirements for the role are satisfied, and that when the requirements are satisfied then the role will execute. We will now illustrate this by example.

Consider the **abs** plan[4] given below:

```
Temporal Plan abs
        Roles .if(@binrel) .end(join-output) .action(@function)
        Constraints .if.op=less  .if.two=0  .action.op=neg
                    .action.input=.if.one
                    .end.fail-input=.if.one
                    .end.succeed-input=.action.output
                     cflow(.if.succeed,.action.in)
                     cflow(.action.out, .end.succeed)
                     cflow(.if.fail,.end.fail)
```

In the axiomatic formulation of the above plan the control flow constraints turn into:

$$\text{precedes(.if.succeed,.action.in)} \wedge [\text{.if.succeed} \neq \bot \leftrightarrow \text{.action.in} \neq \bot]$$

$$\text{precedes(.action.out, .end.succeed)} \wedge [\text{.action.out} \neq \bot \leftrightarrow \text{.end.succeed} \neq \bot]$$

$$\text{precedes(.if.fail,.end.fail)} \wedge [\text{.if.fail} \neq \bot \leftrightarrow \text{.end.fail} \neq \bot]$$

As mentioned earlier, the precedes part of these will be dealt with separately, and the remainder are equivalent to assertions about the controlling conditions for the associated situations i.e. if the controlling condition for .if.succeed, .if.fail, .end.succeed, .end.fail, and .action, are $C_{succeed}$, C_{fail}, $J_{succeed}$, J_{fail} and A respectively, then the above turns into:

$$[C_{succeed} \leftrightarrow A] \wedge [A \leftrightarrow J_{succeed}] \wedge [C_{fail} \leftrightarrow J_{fail}]$$

which is the plan condition for this plan.

The associated rule (viewed as a graph) is shown in Figure 4.3. Now consider the surface plan shown in Figure 4.4. Assuming the parser has matched all the roles of the plan rule against the surface

[4] This is essentially the plan as given in Rich[1981]. Later on we will show that the control flow constraints are actually much stronger than is necessary since they prevent us from recognising valid variants of the plan. This is true for most of Rich's plans, and we will discuss the conditions under which they can and should be weakened.

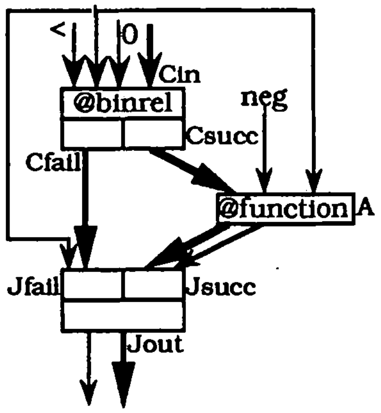


Figure 4.3
Abs Rule

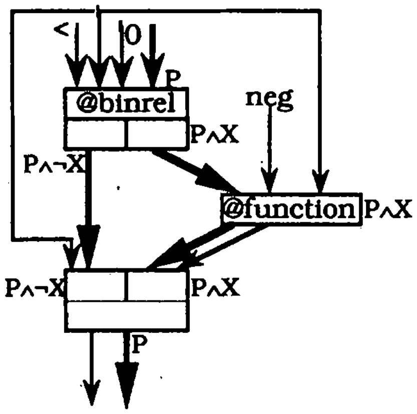


Figure 4.4
Abs Surface Plan

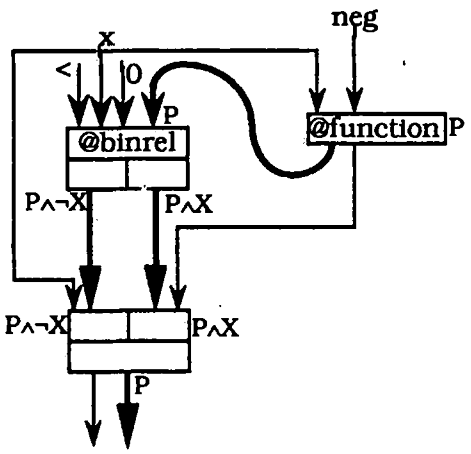


Figure 4.5
Abs Surface Plan
With Neg Before
Conditional

plan and checked that all the data flows are satisfactory, then we have the following values for the controlling conditions in the plan:

A	$P \wedge X$
$C_{succeed}$	$P \wedge X$
C_{fail}	$P \wedge \neg X$
$J_{succeed}$	$P \wedge X$
J_{fail}	$P \wedge \neg X$
C_{in}	P
J_{out}	P

Substituting these into the plan condition gives us the following:

$$[(P \wedge X) \leftrightarrow (P \wedge X)] \wedge [(P \wedge X) \leftrightarrow (P \wedge X)] \wedge [(P \wedge \neg X) \leftrightarrow (P \wedge \neg X)]$$

which reduces to true. This means that the plan is completely satisfactory. Now suppose the program being analysed had the following piece of code in it:

```
y:= -x;
.
.
.
if x<0 then
        z:=y
else
        z:=x;
```

corresponding to the graph shown in Figure 4.5. The parser will tentatively recognise this as an instance of the **abs** plan on the basis of the data flows, and then will proceed to evaluate the plan condition. This time we get the following values for the controlling conditions in the plan:

A	P
$C_{succeed}$	$P \wedge X$
C_{fail}	$P \wedge \neg X$
$J_{succeed}$	$P \wedge X$
J_{fail}	$P \wedge \neg X$
C_{in}	P
J_{out}	P

and substituting these into the plan condition we get:

$$[(P \wedge X) \leftrightarrow P] \wedge [(P \wedge X) \leftrightarrow P] \wedge [(P \wedge \neg X) \leftrightarrow (P \wedge \neg X)]$$

which after simplification reduces to:

$$P \supset X$$

Up until now we have been talking about plans 'executing' as if it was the plans we were interested in. Of course this is not quite accurate - it is the operations implemented by plans that we are really interested in. The switch of viewpoint from plans to operations is done by overlays. So suppose we had an overlay **abs->absop** (say), which took us from the **abs** plan to some operation which returns the absolute value of its input. These could have been defined as follows:

IOSpec **absop** /.input(**integer**) => .output(**integer**)

Postconditions .output=absolute-value(.input)

TemporalOverlay **abs->absop**

Correspondences **absop**.input=**abs**.if.one ∧ **absop**.output=**abs**.end.output

∧ **absop**.in=**abs**.if.in ∧ **absop**.out=**abs**.end.out

This tells us what the input and output situations of the **absop** operation are in terms of situations in the **abs** plan. In controlling condition terms it tells us that the controlling condition for the **absop** operation is the same as that of the test and join in the **abs** plan, in the case of valid plans.

However, in this case discussed above we have a conditional plan. What are its input and output situations, and what is its controlling condition? The general answer to this is as follows:

Suppose we have a plan instance **P**, and an overlay **P->Q** with the input situation of **P** given by s_P, and output situation given by t_P, and the controlling condition of **Q** given by Cond(env(s_P)) in the case of a valid plan. Then, for a plan instance **P** with condition C, we define:

$$\text{in}(\mathbf{Q}) = \begin{cases} s_P \text{ if Cond(env(}s_P\text{))}\wedge\text{C holds} \\ \perp \text{ otherwise} \end{cases}$$

and similarly for out(**Q**). In other words we regard the input situation of **Q** as being equal to $\perp$ if either the input situation given by the plan is $\perp$, or the plan condition does not hold. Note that for valid plans i.e. those whose plan condition is true, this just simply reduces to the original definition.

It should also be clear from the above that the controlling condition for **Q** is simply given by:

Cond(env(s_P))$\wedge$C

This is interpreted as meaning that a conditional plan executes in a generalised control flow environment, whereas a valid plan executes in a normal (non-generalised) control flow environment.

Now, returning to the **abs** example where the plan condition was $P \supset X$, the controlling condition for the **absop** operation is therefore given by $P\wedge(P \supset X)$. This reduces to:

$P\wedge X$

This implies that we have recognised the plan provided it executes in an environment in which both P (whatever it is), and $\overset{\frown}{X}$ (i.e. $x<0$) are true i.e. we have recognised the plan but only think it works when x is negative. Clearly, we would have hoped to recognise this as an **abs** plan in environment P, since this is actually what the code does. The problem here, however, is not the generalised control flow machinery we have set up. For suppose we had a similar plan involving a square root operation, and the plan specified that this happens on the $x>0$ side of the test. If the programmer had actually written code in which the square root operation appeared before the test then this code would clearly only work conditionally (i.e. provided $x >0$). So the recogniser is actually only doing what it should. The problem in the **abs** case is that the **abs** plan is actually over-specified. It insists that the negation operation happen inside the conditional, whereas this is not essential since the test is not actually setting up a precondition of the operation, unlike the square root case. The problem is that in Rich's [1981] plan library **cflow** is used for all control flow constraints, and this is too strong a constraint.

4.4 Generalised Cflows

In order to enable us to express weaker control flow constraints than just cflow, we define the following:

$$\text{scflow}(s,t) \equiv [s \neq \perp \supset t \neq \perp]$$

and

$$\text{ncflow}(s,t) \equiv [t \neq \perp \supset s \neq \perp]$$

and

$$\text{pcflow}(s,t) \equiv \text{precedes}(s,t) \wedge \text{ncflow}(s,t)$$

The interpretation of these is that an scflow constraint between two situations means that s is sufficient for t i.e. if s is reached then t must be reached (or have been reached) as well, but places no constraints on whether or not s is actually reached if t is reached. On the other hand, an ncflow constraint tells us that s is necessary for t i.e. t cannot occur unless s occurs, but places no constraints on whether or not t is actually reached if s is reached. A pcflow constraint between two situations s and t tells us that s is necessary for t, and that s must occur before t.

We note first of all that cflow(s,t) is equivalent to

$$\text{scflow}(s,t) \wedge \text{pcflow}(s,t)$$

and that Rich uses cflow constraints everywhere in his plans when there is a data flow between two actions in the plan, or when an action occurs on one or other side of a test. Introducing the weaker constraints scflow and ncflow enables us to be less restrictive when expressing plans. When there is a data flow between two actions A and B such that A produces a value used by B, <u>and A and B are expected to be in the same control flow environment</u>, then we should use a cflow constraint as Rich does. However, if there is a data flow between two actions A and B such that A produces a value used by B, <u>but B could be in a different control flow environment from A</u>, (as is usually the case with an action feeding its output to a join) then a pcflow constraint should be used, since we want to express the fact that for B to execute, A must execute first. If we wish to express that if some condition is satisfied then we want to perform some action, then an scflow constraint should be used, and if we wish to express the fact that some action should be performed <u>if and only if</u> some condition is satisfied then we should use both an ncflow and an scflow constraint. So, in the square root example above we need an ncflow and an scflow constraint, but in the abs example we only need an scflow constraint. It should be noted that this means that all constraints can be written in one of the two forms:

$A \supset B$ or $A \leftrightarrow B$

So, suppose the **abs** plan is redefined as follows:

```
Temporal Plan abs
    Roles .if(@binrel) .end(join-output) .action(@function)
    Constraints .if.op=less  .if.two=0  .action.op=neg
                .action.input=.if.one
                .end.fail-input=.if.one
                .end.succeed-input=.action.output
                 scflow(.if.succeed,.action.in)
                 cflow(.if.succeed, .end.succeed)
                 pcflow(.action.out,.end.succeed)
                cflow(.if.fail,.end.fail)
```

then we get the following plan condition:

$$[C_{succeed} \supset A] \wedge [J_{succeed} \supset A] \wedge [J_{fail} \leftrightarrow C_{fail}] \wedge [C_{succeed} \leftrightarrow J_{succeed}]$$

Now when matching against the graph in Figure 4.5 we still get the same values for the control conditions in the plan i.e.

A	P
$C_{succeed}$	$P \wedge X$
C_{fail}	$P \wedge \neg X$
$J_{succeed}$	$P \wedge X$
J_{fail}	$P \wedge \neg X$
C_{in}	P
J_{out}	P

$[C_{succeed} \supset A] \wedge [J_{succeed} \supset A] \wedge [J_{fail} \leftrightarrow C_{fail}] \wedge [C_{succeed} \leftrightarrow J_{succeed}]$but when we substitute these into the plan condition we get:

$$[(P \wedge X) \supset P] \wedge [(P \wedge X) \supset P] \wedge [(P \wedge \neg X) \leftrightarrow (P \wedge \neg X)]$$
$$\wedge [(P \wedge \neg X) \leftrightarrow (P \wedge \neg X)]$$

which simplifies to true. So we have correctly recognised the plan as being valid (and unconditional).

4.5 More Complex Plans and their Plan Conditions

So far we have only discussed plans with at most one cond inside them. Constructing the plan condition for a more complex plan, i.e. one with more than one cond, is similar, but more complicated. Consider the rule shown in Figure 4.6, and suppose its control flow constraints are given by:

$$
\begin{array}{l}
P \supset R \wedge \\
Q \supset S \wedge \\
T \supset V \wedge \\
U \supset M \wedge \\
K \supset V \wedge \\
L \supset M \wedge \\
T \leftrightarrow K \wedge \\
U \leftrightarrow L \wedge \\
P \leftrightarrow J \wedge \\
J \supset R
\end{array}
$$

Now we analyse this set of constraints, and break it up into disjoint subsets such that each subset is a maximal set of mutually interdependent constraints. By this is meant that each subset S satisfies the following two properties:

(1) For each constraint C in S, all other constraints involving either of the two variables in C are also in S, and

(2) S cannot be broken down any further into subsets satisfying property (1).

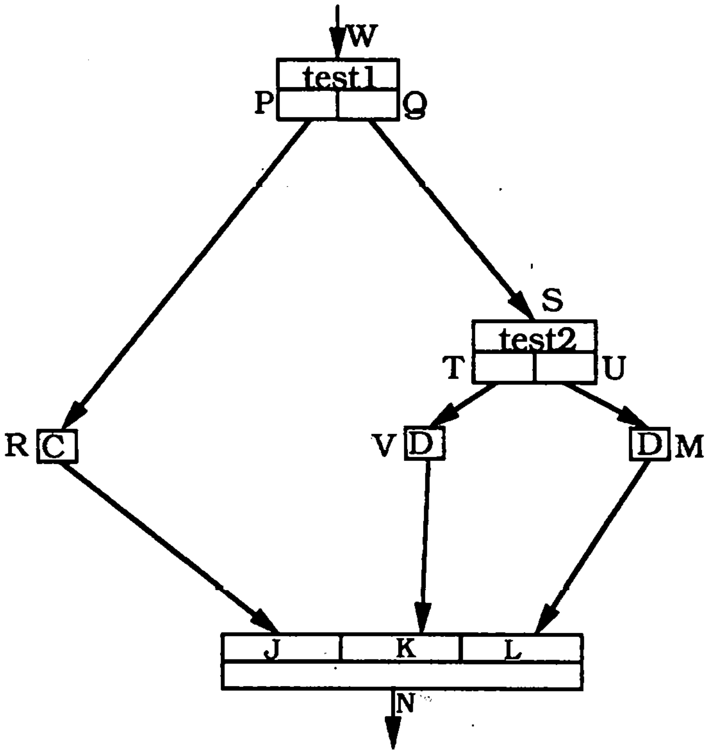
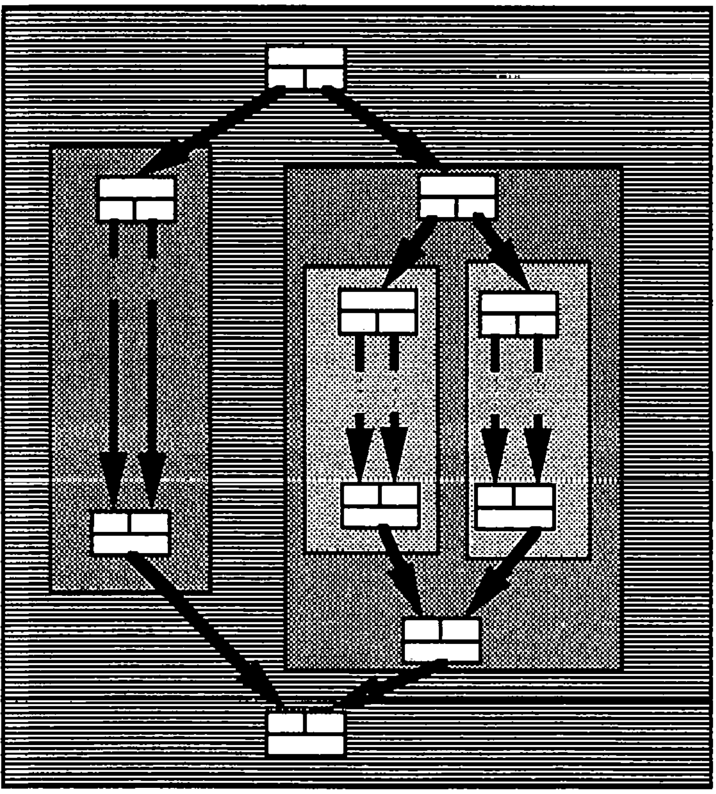


Figure 4.6
A Multiple Cond
Rule



The nested "sub-plans" shown contribute to the plan condition for the rule as described in the text.

Figure 4.7
Rule With Nested Sub-Plans

So, for the above rule, the maximal mutually inter-dependent subsets are:

$\{P \supset R,\ J \supset R,\ P \leftrightarrow J\}\ \{Q \supset S\}\ \{T \supset V,\ K \supset V,\ T \leftrightarrow K\}\ \{U \supset M,\ L \supset M, U \leftrightarrow L\}$

These subsets essentially correspond to different control flow environments within the rule. They correspond to constraints which must be satisfied in each environment. Now although these subsets are independent in terms of the variables involved, they are not independent in the sense that the environment of a test is "responsible" for setting up conditions so that the constraints in each sub-branch of the test are satisfied. So, in this rule (Figure 4.6), there is a condition imposed on test2, namely that:

$$S \wedge (T \leftrightarrow V \wedge V \leftrightarrow K \wedge T \leftrightarrow K) \wedge (U \leftrightarrow M \wedge M \leftrightarrow L \wedge U \leftrightarrow L)$$

should hold. This is equivalent to regarding the test test2 as being part of a sub-plan with its own plan condition, and the above expression gives the true generalised controlling condition for this test. Note how the extra parts of this condition come from the subsets of the constraints corresponding to T and U, i.e. the fail and succeed environments of the test. If this condition does not hold at the test then we cannot be sure that the constraints for the branches of the test will be satisfied when the plan executes.

This means that the constraint $Q \supset S$ should really be:

$$Q \supset [S \wedge (T \leftrightarrow V \wedge V \leftrightarrow K \wedge T \leftrightarrow K) \wedge (U \leftrightarrow M \wedge M \leftrightarrow L \wedge U \leftrightarrow L)]$$

So conditions in plans should really be propagated up the plan as implied in Figure 4.7. This means that the plan condition for the rule above is:

$$[P \supset R \wedge J \supset R \wedge P \leftrightarrow J]$$

$$\wedge\ [Q \supset [S \wedge (T \leftrightarrow V \wedge V \leftrightarrow K \wedge T \leftrightarrow K) \wedge (U \leftrightarrow M \wedge M \leftrightarrow L \wedge U \leftrightarrow L)]]$$

We will return to this example towards the end of the chapter.

4.5 The Collapsing Operation

When a programmer has made unforeseen optimisations involving sharing parts of two or more plans in their code, the resulting surface plan can be different from what one might expect. An example is shown in Figure 4.8, where the programmer has decided to implement a plan given by rule a) in the figure, using rules b) and c) to implement the subplans, which should give rise to the plan d) as a result. However the programmer has realised that this involves redundant computation and has implemented the plan as graph e). A theorem proving system based on the formal semantics of the plan calculus would have no problems with this, since there is nothing in the axioms to prevent such sharing (we would just end up deducing that the two sub-plans have the same input situations), but a graph recognition process might. Discussion of how this is dealt with will be delayed until the next chapter where a parsing algorithm for flowgraph grammars will be presented which explicitly allows such sharing. Indeed the parser would have to go to quite a lot of trouble (by implementing what we have called the no-sharing check) to prevent this from happening. The main point to note here is that this sharing of sub-parts of plans is allowed by the underlying plan calculus, thus justifying our use of a parser which also allows it.

On the other hand, the case where the programmer has failed to make optimisations assumed by the plan library (i.e. a plan involves some action feeding its output(s) to more than one place, but the programmer has duplicated the action instead) is more problematical.

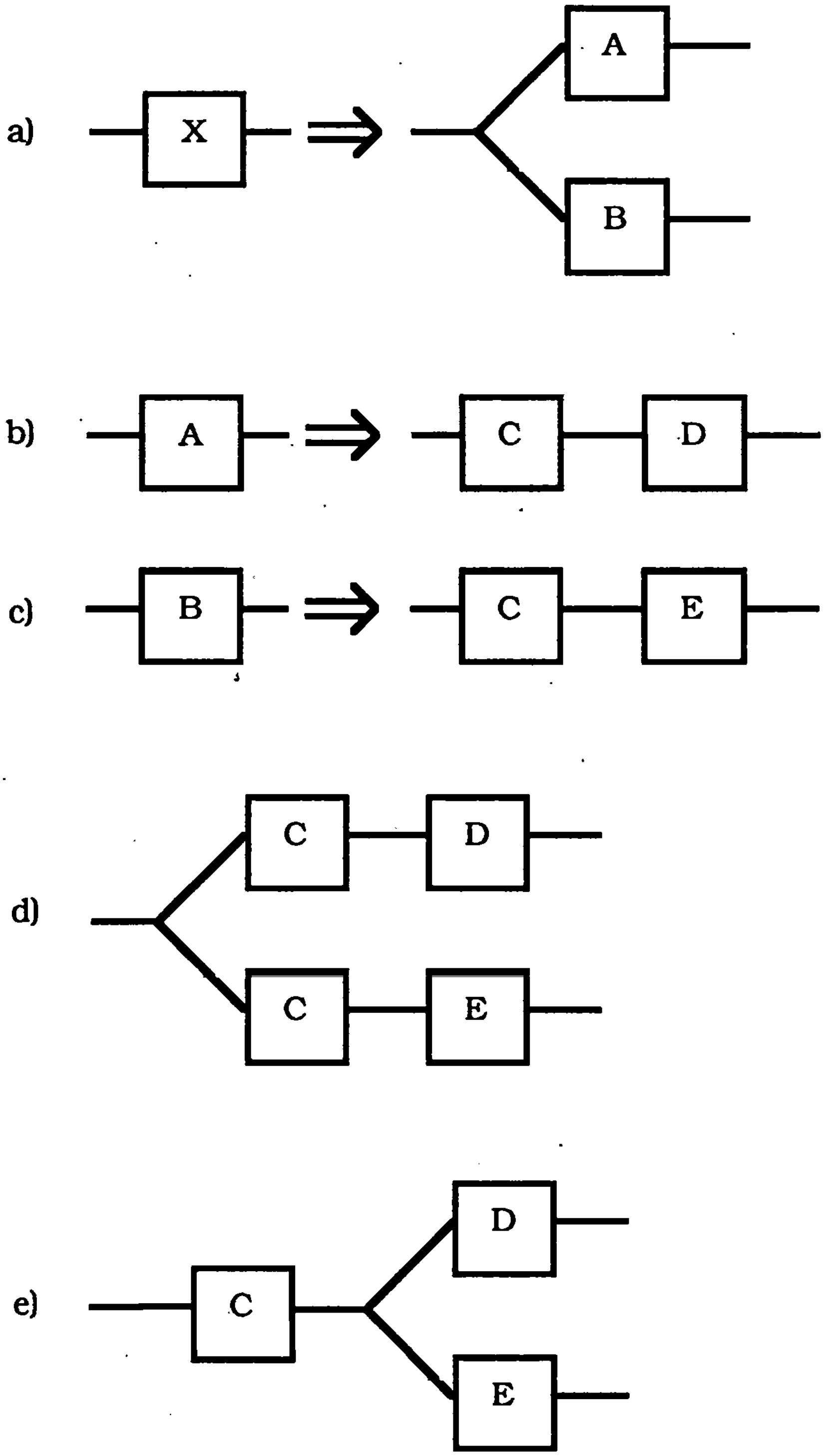


Figure 4.8
Structure Sharing

An example is shown in Figure 4.9(a). This shows the case where a programmer has performed some action twice in the program, when they could have performed it only once, as shown in Figure 4.9(b). An example in code might be:

```
y:=f(a)+c;
.
.
.
z:=f(a)+d
```

We want first of all to recognise that an action has been repeated, and secondly to realise that this is equivalent to:

```
w:=f(a); /*f has no side effects!!*/
y:=w+c;
.
.
.
z:=w+d
```

What this amounts to is the need to recognise graphs like those in Figure 4.9 as equivalent. Doing this really amount to identifying the duplicated actions and their outputs, and operations for which this can be done will be called collapsible.

Now it might be wondered why this is an issue. Why can't all operations with the same inputs be collapsed? The issue is not even that we have specified that f has no side effects, for if it did it would have as input some of the mutable functions, and would output new values for some of these. In this case the second occurrence of f would have different inputs, so the comment was really for clarity of exposition. It turns out that in fact most operations with the same inputs can be collapsed, but not all. The problem can be seen by considering Figure 4.10. The point here is that even though the two

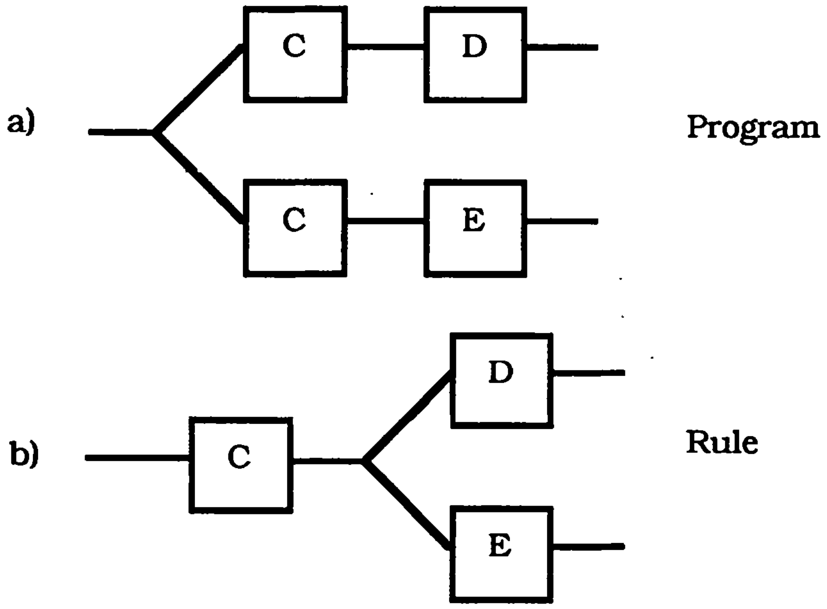


Figure 4.9
Program With Unnecessarily
Duplicated Actions

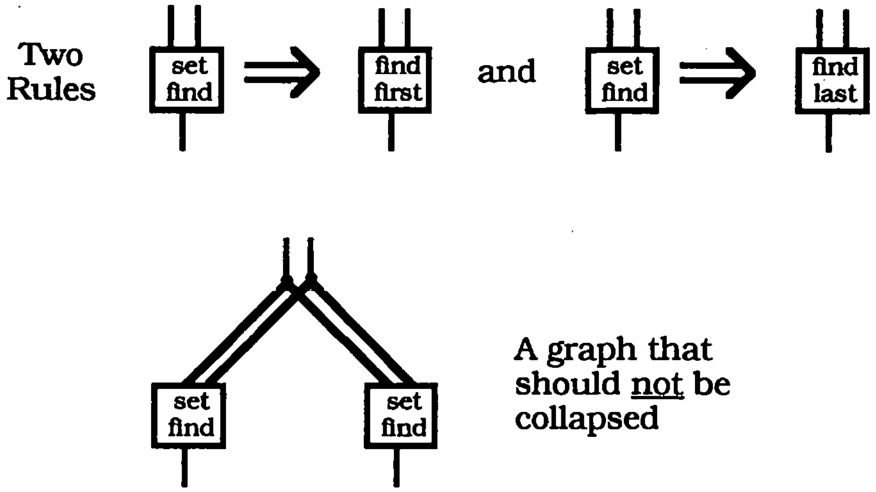


Figure 4.10
The set-find example

set-find operations[5] have the same inputs, their outputs are not the same value, since one has been implemented by a **find-first** operation, and the other by a **find-last** operation[6]. The point here is not that we have to worry about whether or not two operations that we wish to collapse have been implemented by the same underlying plan or not, since this would be unnecessarily restrictive. For instance it would prevent us from recognising that the outputs from a bubble-sort procedure and a merge- sort procedure were the same if the two procedures received the same inputs. The real problem with **set-find** is that it is not deterministic i.e. its definition tells us what *properties* its output satisfies, and guarantees that such an object exists, but does not tell us which object it actually is, since there could be many. Hence one possible implementation, using a list to represent the set, could find the first element of the list satisfying the relevant property, and another implementation, could find the last element satisfying the property. Since these objects are different, we cannot identify the tie-points representing them.

It follows from this that the only operations which are collapsible are the deterministic ones i.e. those which, given a set of inputs to the operation, have uniquely determined outputs. To make this precise consider an operation **op**, whose specification and corresponding axioms (rewritten in terms of controlling conditions) are of the form:

IOSpec **op** $\mathrm{in}_1(\mathbf{B_1})\ldots\mathrm{in}_n(\mathbf{B_n}) \Rightarrow \mathrm{out}_1(\mathbf{C_1})\ldots\mathrm{out}_m(\mathbf{C_m})$

Preconditions $\mathrm{le}_{pre}(\mathrm{in}_1,\ldots,\mathrm{in}_n)$

Postconditions $\mathrm{le}_{post}(\mathrm{in}_1,\ldots,\mathrm{in}_n,\mathrm{out}_1,\ldots,\mathrm{out}_n)$

and

[5] We are treating sets here as if they were bags i.e. they are allowed to contain repeated elements.

[6] **find-first** and **find-last** are not plans currently in the library. They are fictional plans with suggestive names to illustrate the problem.

$$\begin{array}{l}
\forall\alpha\beta\ [\ \mathbf{op}(\alpha) \wedge \mathbf{op}(\beta) \wedge in_1(\alpha)=in_1(\beta) \wedge \ldots \wedge in_n(\alpha)=in_n(\beta) \\
\qquad \wedge\ out_1(\alpha)=out_1(\beta) \wedge \ldots \wedge out_m(\alpha)=out_m(\beta) \\
\qquad \wedge\ in(\alpha)=in(\beta) \wedge out(\alpha)=out(\beta)\ ] \supset \alpha=\beta\ ]
\end{array}$$

$$\begin{array}{l}
\forall x_1 \ldots x_n y_1 \ldots y_m st[\ [precedes(s,t) \wedge [Cond(s) \leftrightarrow Cond(t)] \\
\qquad\qquad\qquad \wedge\ [Cond(s) \supset \\
\qquad\qquad\qquad\quad [t \neq \bot \wedge \mathbf{B_1}(x_1,s) \neq \mathit{undefined} \\
\qquad\qquad\qquad\quad \wedge \ldots \wedge \mathbf{B_n}(x_n,s) \neq \mathit{undefined} \\
\qquad\qquad\qquad\quad \wedge\ \mathbf{C_1}(y_1,t) \neq \mathit{undefined} \\
\qquad\qquad\qquad\quad \wedge \ldots \wedge \mathbf{C_m}(y_m,t) \neq \mathit{undefined} \\
\qquad\qquad\qquad\quad \wedge\ le_{pre}(\mathbf{B_1}(x_1,s),\ldots,\mathbf{B_n}(x_n,s)) \\
\qquad\qquad\qquad\quad \wedge\ le_{post}(\mathbf{B_1}(x_1,s),\ldots,\mathbf{B_n}(x_n,s),\mathbf{C_1}(y_1,t),\ldots,\mathbf{C_m}(y_m,t))\ ]]] \\
\qquad\qquad\qquad \leftrightarrow \exists\alpha\ [\ \ \mathbf{op}(\alpha) \wedge in(\alpha)=s \wedge out(\alpha)=t \\
\qquad\qquad\qquad\qquad \wedge\ in_1(\alpha)=x_1 \wedge \ldots \wedge in_n(\alpha)=x_n \\
\qquad\qquad\qquad\qquad \wedge\ out_1(\alpha)=y_1 \wedge \ldots \wedge out_m(\alpha)=y_m]\ ]
\end{array}$$

where le_{pre} and le_{post} represent the logical expressions making up the pre- and post-conditions of the operation. We say that this operation is deterministic if and only if the following condition holds:

There exist functions $f_1,\ldots,f_m$ such that

$$\begin{array}{l}
\forall\alpha x_1 \ldots x_n y_1 \ldots y_m st[\ [\ \ \mathbf{op}(\alpha) \wedge in(\alpha)=s \wedge out(\alpha)=t \\
\qquad\qquad\qquad\qquad \wedge\ in_1(\alpha)=x_1 \wedge \ldots \wedge in_n(\alpha)=x_n \\
\qquad\qquad\qquad\qquad \wedge\ out_1(\alpha)=y_1 \wedge \ldots \wedge out_m(\alpha)=y_m] \supset \\
\qquad\qquad\qquad\quad [\ \ \mathbf{C_1}(y_1,t)=f_1(\mathbf{B_1}(x_1,s),\ldots,\mathbf{B_n}(x_n,s)) \\
\qquad\qquad\qquad\qquad \wedge \ldots \wedge \mathbf{C_m}(y_m,t)=f_m(\mathbf{B_1}(x_1,s),\ldots,\mathbf{B_n}(x_n,s))\ ]
\end{array}$$

In other words, there is a pure functional relationship between the inputs and the outputs to an operation. It should be noted that this functional relationship can be either implicit or explicit in the operation definition. Obviously, those where it is explicit are easier to recognise, but even where it is implicit it is still a logical consequence of the definition. Accordingly, we can pre-analyse the plan library and work out which operations are deterministic. To show that the collapsing operation is justified under the plan calculus for deterministic operations, consider Figure 4.11, which shows two

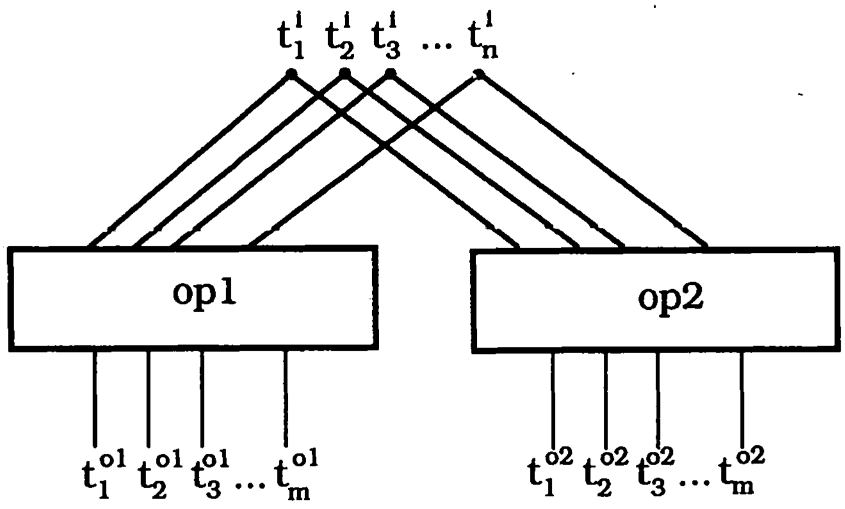

t_1^i t_2^i t_3^i ... t_n^i
op1
op2
t_1^{o1} t_2^{o1} t_3^{o1} ... t_m^{o1}
t_1^{o2} t_2^{o2} t_3^{o2} ... t_m^{o2}

Figure 4.11
Two Instances of op operation

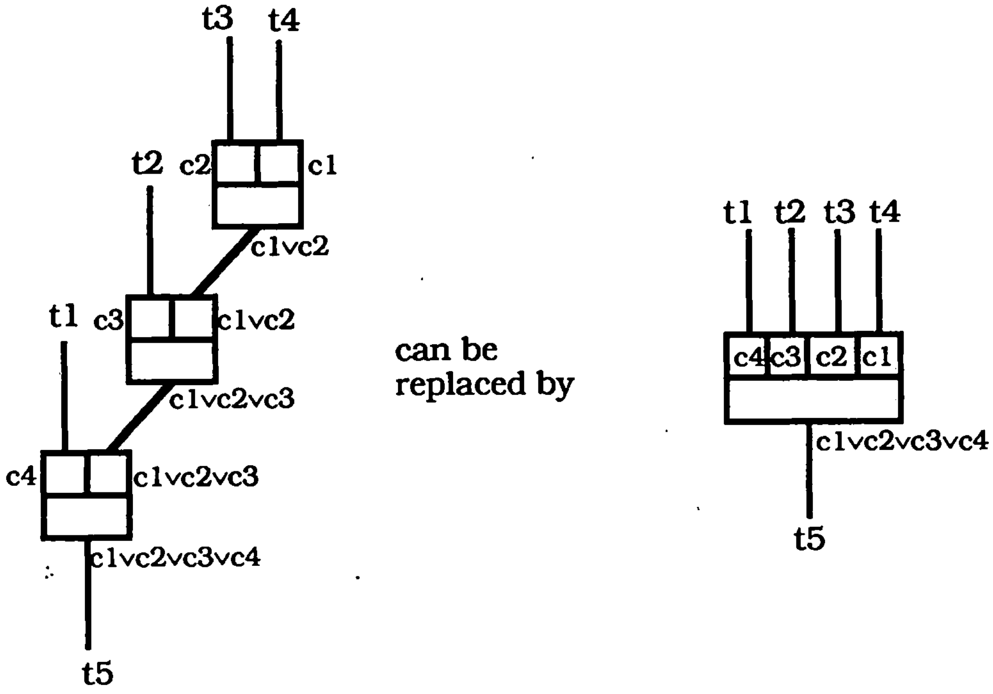


Figure 4.12
Cascades of Joins

instances op1 and op2 of an **op** operation, with common input tie-points. Bearing in mind (from the earlier discussion of tie-points) that (refer to Figure 4.11 for the notation)

$$\begin{array}{l}
t^{i}{}_{1} = \mathbf{B}_{1}(\mathrm{in}_{1}(\mathrm{op1}),\mathrm{in}(\mathrm{op1})) = \mathbf{B}_{1}(\mathrm{in}_{1}(\mathrm{op2}),\mathrm{in}(\mathrm{op2})) \\
\wedge \ldots \wedge t^{i}{}_{n} = \mathbf{B}_{n}(\mathrm{in}_{n}(\mathrm{op1}),\mathrm{in}(\mathrm{op1})) = \mathbf{B}_{n}(\mathrm{in}_{n}(\mathrm{op2}),\mathrm{in}(\mathrm{op2})) \\
\wedge\ t^{o1}{}_{1} = \mathbf{C}_{1}(\mathrm{out}_{1}(\mathrm{op1}),\mathrm{out}(\mathrm{op1})) \\
\wedge \ldots \wedge t^{o1}{}_{m} = \mathbf{C}_{m}(\mathrm{out}_{m}(\mathrm{op1}),\mathrm{out}(\mathrm{op1})) \\
\wedge\ t^{o2}{}_{1} = \mathbf{C}_{1}(\mathrm{out}_{1}(\mathrm{op2}),\mathrm{out}(\mathrm{op2})) \\
\wedge \ldots \wedge t^{o2}{}_{m} = \mathbf{C}_{m}(\mathrm{out}_{m}(\mathrm{op2}),\mathrm{out}(\mathrm{op2}))
\end{array}$$

we can deduce that

$$\begin{array}{l}
t^{o1}{}_{1}=t^{o2}{}_{1}=f_{1}(t^{i}{}_{1},\ldots,t^{i}{}_{n}) \wedge \\
t^{o1}{}_{2}=t^{o2}{}_{2}=f_{2}(t^{i}{}_{1},\ldots,t^{i}{}_{n}) \wedge \\
\cdot \\
\cdot \\
\cdot \\
t^{o1}{}_{m}=t^{o2}{}_{m}=f_{m}(t^{i}{}_{1},\ldots,t^{i}{}_{n})
\end{array}$$

so the output tie-points of the two operations can be identified. Using $t^{o1}{}_{i}$ to stand for both $t^{o1}{}_{i}$ and $t^{o2}{}_{i}$, we can go further than this, and actually replace the two operations by a single one. Since we are dealing with actual instances of an **op** operation we can deduce :

$$\begin{array}{l}
[\mathrm{Cond}(\mathrm{in}(\mathrm{op1})) \supset \\
\quad [\ t^{i}{}_{1} \neq \mathit{undefined} \wedge \ldots \wedge t^{i}{}_{n} \neq \mathit{undefined} \\
\quad \wedge\ t^{o1}{}_{1} \neq \mathit{undefined} \wedge \ldots \wedge t^{o1}{}_{m} \neq \mathit{undefined} \\
\quad \wedge\ \mathrm{le}_{\mathrm{pre}}(t^{i}{}_{1},\ldots,t^{i}{}_{n}) \\
\quad \wedge\ \mathrm{le}_{\mathrm{post}}(t^{i}{}_{1},\ldots,t^{i}{}_{n},t^{o1}{}_{1},\ldots,t^{o1}{}_{m})\ ]]
\end{array}$$

and

$$\begin{array}{l}
[\mathrm{Cond}(\mathrm{in}(\mathrm{op2})) \supset \\
\quad [\ t^{i}{}_{1} \neq \mathit{undefined} \wedge \ldots \wedge t^{i}{}_{n} \neq \mathit{undefined} \\
\quad \wedge\ t^{o1}{}_{1} \neq \mathit{undefined} \wedge \ldots \wedge t^{o1}{}_{m} \neq \mathit{undefined} \\
\quad \wedge\ \mathrm{le}_{\mathrm{pre}}(t^{i}{}_{1},\ldots,t^{i}{}_{n}) \\
\quad \wedge\ \mathrm{le}_{\mathrm{post}}(t^{i}{}_{1},\ldots,t^{i}{}_{n},t^{o1}{}_{1},\ldots,t^{o1}{}_{m})\ ]]
\end{array}$$

from which it follows that[7]:

$$\begin{array}{l}
[\ [\mathrm{Cond}(op1) \vee \mathrm{Cond}(op2)] \supset \\
\qquad [\ \ t^{i}{}_{1} \neq \mathit{undefined} \wedge \ldots \wedge t^{i}{}_{n} \neq \mathit{undefined} \\
\qquad \wedge\ t^{o1}{}_{1} \neq \mathit{undefined} \wedge \ldots \wedge t^{o1}{}_{m} \neq \mathit{undefined} \\
\qquad \wedge\ le_{pre}(t^{i}{}_{1},\ldots,t^{i}{}_{n}) \\
\qquad \wedge\ le_{post}(t^{i}{}_{1},\ldots,t^{i}{}_{n},t^{o1}{}_{1},\ldots,t^{o1}{}_{m})\ ]]
\end{array}$$

Now suppose that we define two new situations s1 and s2 as follows:

$$s1=\min_{precedes}(in(op1),in(op2))$$

$$s2=\min_{precedes}(out(op1),out(op2))$$

where $\min_{precedes}(s,t)$ is simply the earliest (according to precedes) of s and t (or either of them if they are equal). Clearly,

$$s1\neq\bot \leftrightarrow [\mathrm{Cond}(in(op1)) \vee \mathrm{Cond}(in(op2))]$$

$$s2\neq\bot \leftrightarrow [\mathrm{Cond}(out(op1)) \vee \mathrm{Cond}(out(op2))]$$

from which it follows that the controlling condition for s1 and s2 is [Cond(in(op1)) v Cond(in(op2))]. In other words s1 and s2 occur in the generalised control flow environment determined by [Cond(in(op1)) v Cond(in(op2))]. Additionally, the following is true of s1 and s2:

$$precedes(s1,s2)$$

Accordingly, we can deduce (from the axioms for **op**) that there exists an instance of the **op** operation with input situation s1, output situation s2, inputs and outputs the same as op1 and op2, and controlling condition given by [Cond(in(op1)) v Cond(in(op2))]. Furthermore, this new instance actually implies the existence of the original two, and so can replace them. Since the 'true' interpretation of the graph for a program is really the set of axioms for the operations and control and

[7] We are using the identity $[A \supset C] \wedge [B \supset C] \equiv [(A \vee B) \supset C]$

data flows between them, and the axioms obey the usual rule of substitutivity, we see that we can therefore use either of t^{o1}_i or t^{o2}_i (for any $1 \leq i \leq m$) anywhere they occur in the axioms. In particular this means that we can connect anything that was connected to any of the outputs from op2, to the corresponding outputs from op1 instead, without changing the semantics of the graph. So whenever we have two operations of the same type with the same inputs we can replace them by a single operation with the same inputs, the same outputs, and controlling condition equal to the disjunction of the controlling conditions for the original two operations. Thus, the collapsing operation performed by the parser described in the next two chapters has a sound formal justification, rather than just being a "hack".

4.6 Generalised Joins

Before we can demonstrate how all of this is put to use, we need to deal with a problem caused by joins. Joins are rather strange entities since they represent no real computation, but are merely present to rejoin diverging control flows and to give us a single tie-point in data-flow graphs to represent the several values that an object could have depending on which way the computation went. Thus they are really an artifact of the representation we are using. In principle we could take any two disjoint situations (i.e. two situations s and t such that $[s \neq \perp \leftrightarrow t = \perp]$), and tie-points representing values in those situations and add a join with these tie-points as inputs, and a new output tie-point representing the value of the inputs in both situations. Note that adding such a join simply adds new axioms to the axiomatic representation of the program - it does not involve changing any of the existing axioms or situations in any way. It turns out that we will often need to add such joins.

It actually turns out to be useful to define a more general notion of **join**. Let us define an **n-join-output** by the following schema, modelled on the definition for **join-output** given in Chapter 3.

Join **n-join-output** / .input$_1$(**object**) ... .input$_n$(**object**) =>.output(**object**)
Postconditions [succeed$_1$(**n-join-output**)$\neq\perp \supset$ .output=.input$_1$]
$\wedge \ldots$
$\wedge$ [succeed$_n$(**n-join-output**)$\neq\perp \supset$ .output=.input$_n$]

In this schema succeed$_1$...succeed$_n$ represent n input situations. This is converted to the following axiom schema:

$$\begin{array}{l}
\forall\alpha\beta[[\ \mathbf{n\text{-}join\text{-}output}(\alpha) \wedge \mathbf{n\text{-}join\text{-}output}(\beta) \\
\qquad \wedge\ \mathrm{succeed}_1(\alpha)=\mathrm{succeed}_1(\beta) \\
\qquad \wedge \ldots \\
\qquad \wedge\ \mathrm{succeed}_n(\alpha)=\mathrm{succeed}_n(\beta) \\
\qquad \wedge\ \mathrm{out}(\alpha)=\mathrm{out}(\beta) \\
\qquad \wedge\ \mathrm{input}_1(\alpha)=\mathrm{input}_1(\beta) \\
\qquad \wedge \ldots \\
\qquad \wedge\ \mathrm{input}_n(\alpha)=\mathrm{input}_n(\beta) \\
\qquad \wedge\ \mathrm{output}(\alpha)=\mathrm{output}(\beta)] \supset \alpha=\beta\ ]
\end{array}$$

and

$$\begin{array}{l}
\forall w x_1 \ldots x_n s t_1 \ldots t_n[\quad [\ [\mathrm{none}_1(t_1 \ldots t_n) \vee \ldots \vee \mathrm{none}_n(t_1 \ldots t_n)] \\
\qquad\qquad\qquad \wedge\ [t_1 \neq \perp \supset [s=t_1 \wedge w=x_1]] \\
\qquad\qquad\qquad \wedge \ldots \\
\qquad\qquad\qquad \wedge\ [t_n \neq \perp \supset [s=t_n \wedge w=x_n]]\ ] \\
\qquad\qquad \leftrightarrow \exists\alpha[\ \mathbf{n\text{-}join\text{-}output}(\alpha) \wedge \mathrm{out}(\alpha)=s \\
\qquad\qquad\qquad \wedge\ \mathrm{succeed}_1(\alpha)=t_1 \\
\qquad\qquad\qquad \wedge \ldots \\
\qquad\qquad\qquad \wedge\ \mathrm{succeed}_n(\alpha)=u \\
\qquad\qquad\qquad \wedge\ \mathrm{output}(\alpha)=w \\
\qquad\qquad\qquad \wedge\ \mathrm{input}_1(\alpha)=x_1 \\
\qquad\qquad\qquad \wedge \ldots \\
\qquad\qquad\qquad \wedge\ \mathrm{input}_1(\alpha)=x_n\ ]]
\end{array}$$

where $\text{none}_i(t_1...t_n)$ is defined by[8]:

$$\text{none}_i(t_1...t_n) \equiv t_1=\bot \wedge ... \wedge t_{i-1}=\bot \wedge t_{i+1}=\bot \wedge ... \wedge t_n=\bot$$

For an **n-join-output** the input controlling conditions $c_1...c_n$ are simply those of the corresponding input situations, and the controlling condition of its output situation is given by:

$$c_1 \vee ... \vee c_n$$

Now in the surface plan for a program, as well as in the rules in the library, "cascades" of **joins** like those shown in Figure 4.12 will be replaced by single **n-joins** as shown in the figure. Furthermore, if the parser is looking for a **join** with known inputs and known input controlling situations, and it cannot find a **join** with these inputs and these controlling situations then it will simply introduce one as this has no effect on the interpretation of the program. To do this it will create a new tie-point representing the output of the **join** (i.e. representing the 'merged' data values). It will also sometimes turn out that the parser will be looking for a **join** like that shown in Figure 4.13. In this case it can deduce that the output tie-point is actually the same tie-point as the inputs.

One last point needs to be made here. There is a special form of the collapsing operation applicable to **joins**. The situation in which this arises is shown in Figure 4.14 where the i^{th} input to an **n-join** is the same as the j^{th} input to an **m-join**. Then, provided that the i^{th} controlling condition c_i of the first **join** and the j^{th} controlling condition d_j of the second **join** satisfy:

$$c_i \leftrightarrow d_j$$

[8] Strictly speaking we ought to worry about the end cases of i=1 and i=n in this definition, but given that the meaning is clear, it will be left as it is.

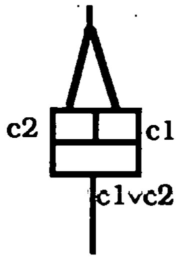


Figure 4.13 A Join With Identical Inputs

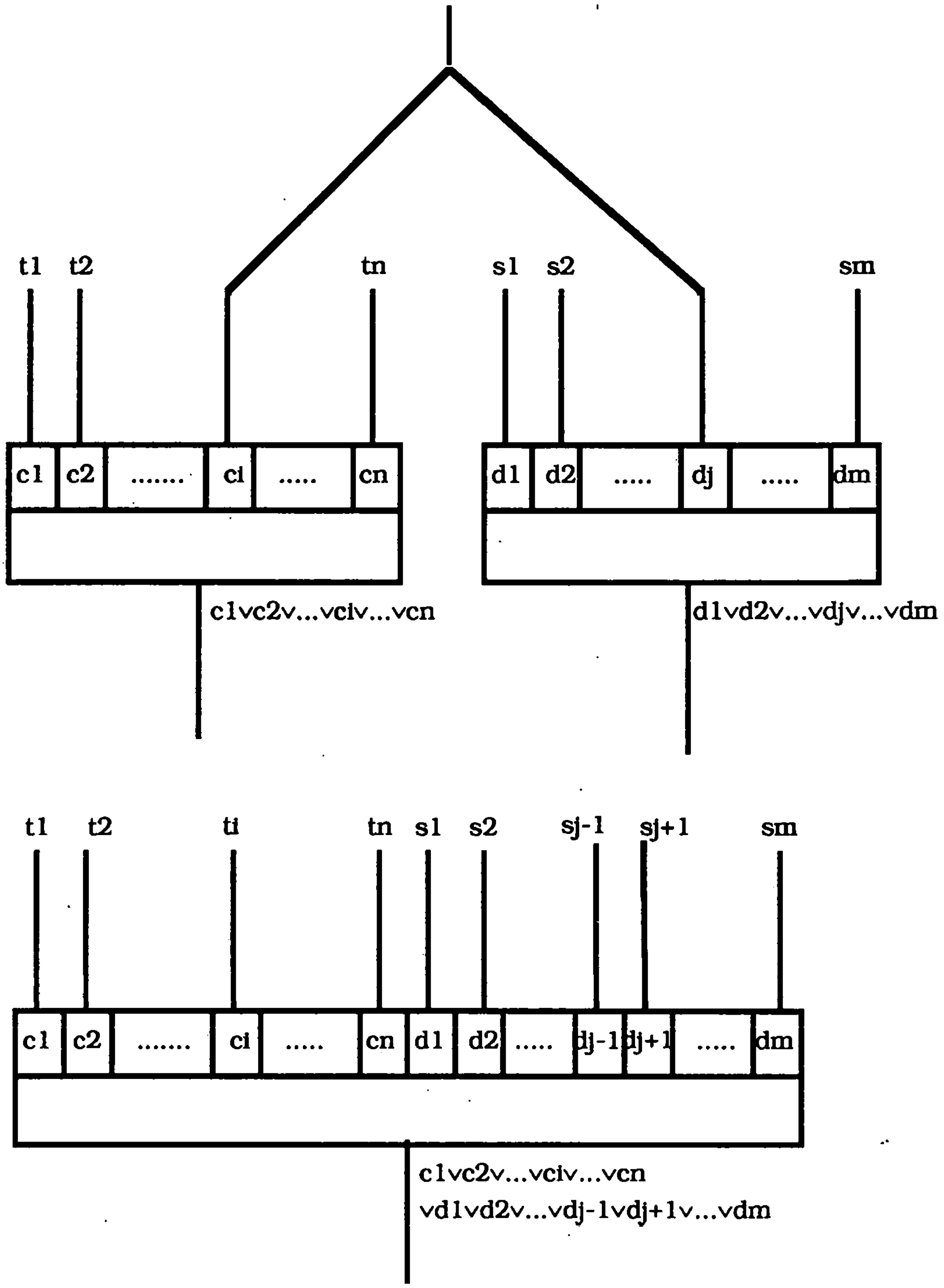


Figure 4.14 Join Collapsing

then we can collapse the **join** into an **(m+n-1)-join** as shown in the figure, provided that:

$$(c_1 \vee \ldots \vee c_{i-1} \vee c_{i+1} \vee \ldots \vee c_n) \leftrightarrow \neg(d_1 \vee \ldots \vee d_{j-1} \vee c_{j+1} \vee \ldots \vee d_m)$$

i.e. provided they are all disjoint.

4.7 Complex Examples

Now we are in a position to demonstrate the power of all this machinery for control flow condition manipulation, and structure sharing and collapsing (and in the next two chapters we will describe a parser that can do both of these).

Consider the rules shown in Figure 4.15 (we have omitted data flow arcs for clarity). These correspond to clichés which could have been coded (schematically) as follows:

```
F    if test1 then B else C
G    if test1 then D else C
Z    if test2 then F else G
```

resulting in code for Z:

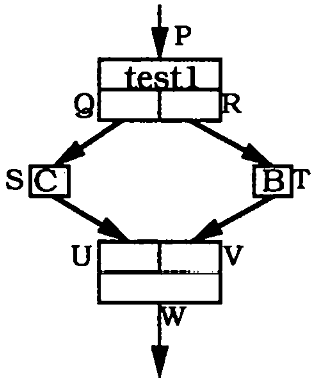
```
if test2 then
    if test1 then B else C
else
    if test1 then D else C
```

This would have resulted in the graph shown in Figure 4.16, and this would have been easily recognised. However suppose the programmer had optimised their code into the following:

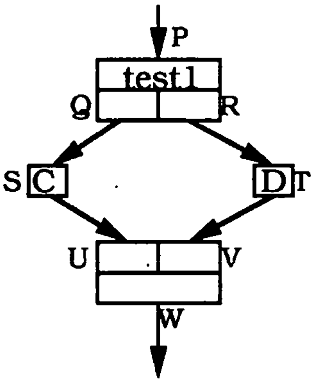
```
if test1 then
    if test2 then B else D
else
    C
```

Plan Condition for Both X and Y

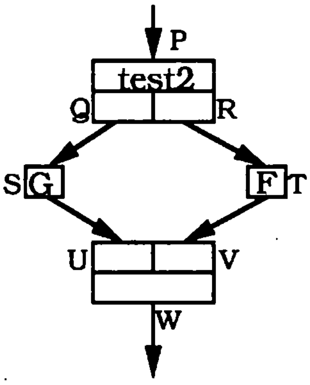
$(U\leftrightarrow Q)\wedge(V\leftrightarrow R)\wedge(Q\supset S)\wedge(U\supset S)\wedge(R\supset T)\wedge(V\supset T)$



Rule for F



Rule for G



Rule for Z

Plan Condition for Z

$(U\leftrightarrow Q)\wedge(V\leftrightarrow R)\wedge(Q\supset S)\wedge(U\supset S)\wedge(R\supset T)\wedge(V\supset T)$

Figure 4.15
Rules for F, G, and Z

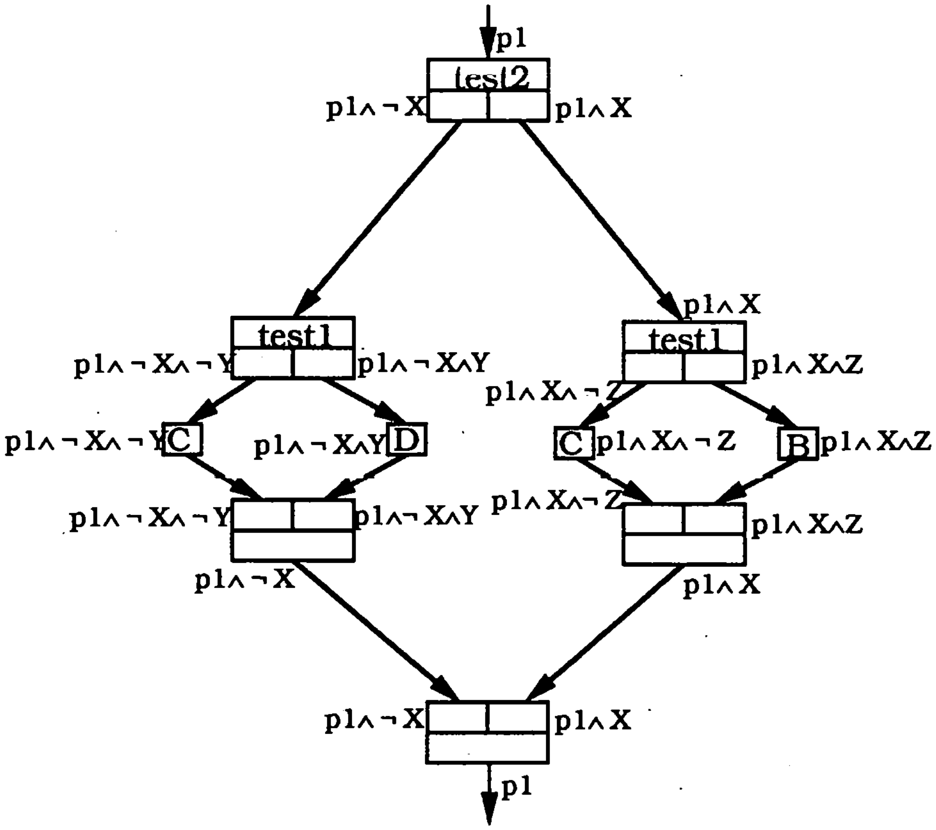


Figure 4.16
Unoptimised Graph

Even people have to think quite hard to recognise this as equivalent to the earlier code, and this is reflected in the difficulty of seeing that the surface plan for this code, shown in Figure 4.17, is not only equivalent to that shown in Figure 4.16, but has been derived from the same clichés. We will now show how the notion of generalised control-flow environments enables this.

Figure 4.18 shows the situation after the parser has found suitable operations satisfying the data flow constraints (which we have not shown). It then evaluates the plan condition, using the following substitutions:

P	$P1$
Q	$P1\wedge\neg X$
R	$P1\wedge X$
S	$P1\wedge\neg X$
T	$P1\wedge X\wedge Y$
U	$P1\wedge\neg X$
V	$P1\wedge X$
W	$P1$

The plan condition for the rule for F is:

$$(U\leftrightarrow Q)\wedge(V\leftrightarrow R)\wedge(Q\supset S)\wedge(U\supset S)\wedge(R\supset T)\wedge(V\supset T)$$

After substituting the above values and simplifying we get:

$$(P1\wedge X)\supset Y)$$

which gives us (assuming the input situation of F is given by the input to the test):

$$P1\wedge(\neg X\vee Y)$$

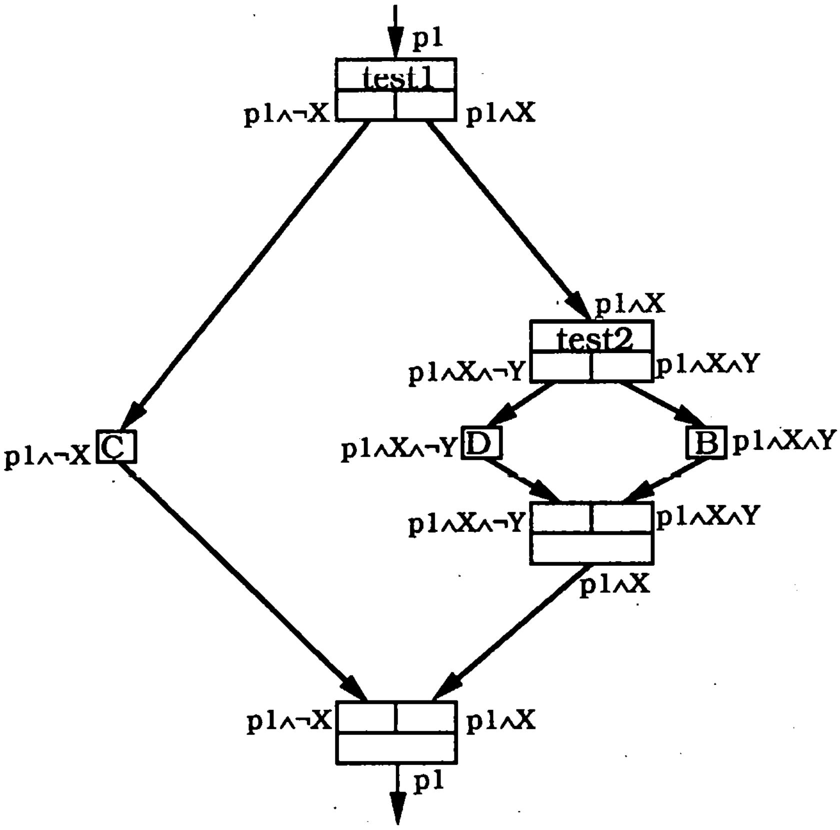


Figure 4.17
Optimised Graph

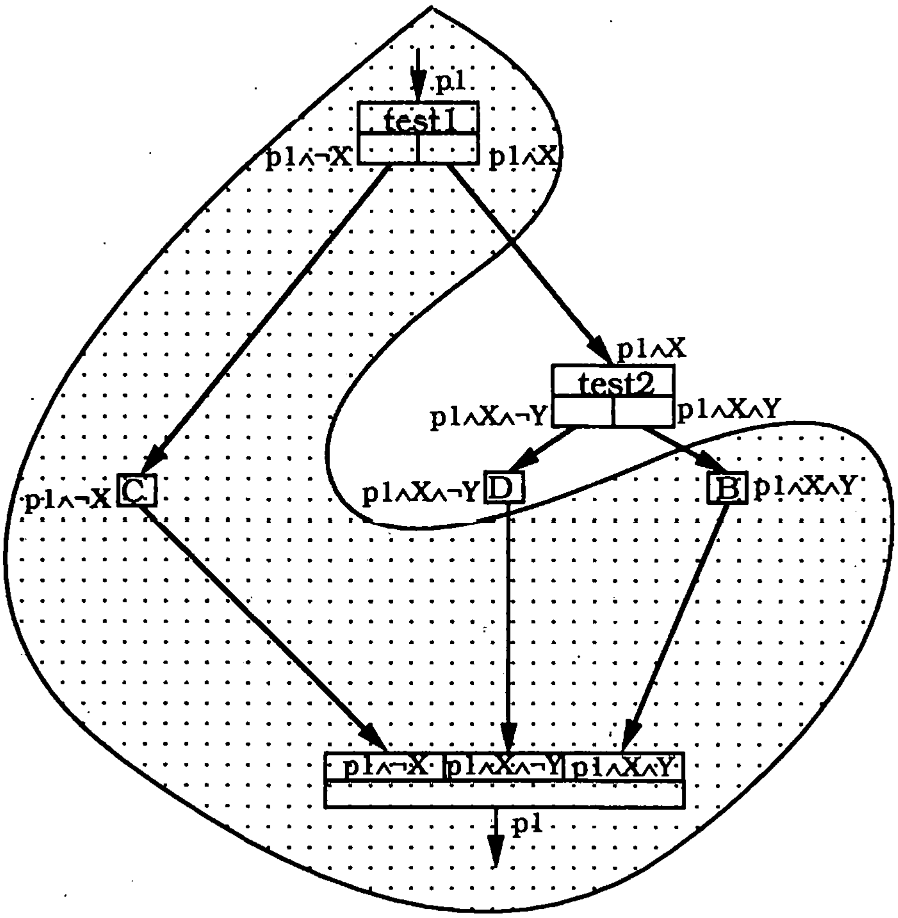


Figure 4.18
F Found In Optimised Graph

for its controlling condition. Notice how this confirms the intuition that this plan works if test1 fails($\neg X$), or if test2 succeeds(Y), but not otherwise.

In a similar fashion a candidate G plan is found (Figure 4.19), and this time its controlling condition evaluates to:

$$P1 \wedge (\neg X \vee \neg Y)$$

Now the parser is essentially in the position shown in Figure 4.20. Suppose it has found that test2 and the F and G plans just found satisfy the data flow constraints needed. Then it will be in the position of looking for an appropriate join (as required by the rule in Figure 4.15). However, there is no such join in the program, since the data outputs (which we have not shown) of both the F and G plans are the data outputs from the 3-join in Figures 4.18 and 4.19. However, we are effectively looking for a join like that shown in Figure 4.20, where the inputs on both sides of the join are the same. The plan condition for this rule for Z is also:

$$(U \leftrightarrow Q) \wedge (V \leftrightarrow R) \wedge (Q \supset S) \wedge (U \supset S) \wedge (R \supset T) \wedge (V \supset T)$$

and so far the substitutions are:

P	$P1 \wedge X$
Q	$P1 \wedge X \wedge \neg Y$
R	$P1 \wedge X \wedge Y$
S	$P1 \wedge (\neg X \vee \neg Y)$
T	$P1 \wedge (\neg X \vee Y)$
U	??
V	??
W	??

In this situation, where the parser is looking for a join to join the outputs from two (or more) operations, and cannot find one in the program it can just add one with input conditions equal to the input

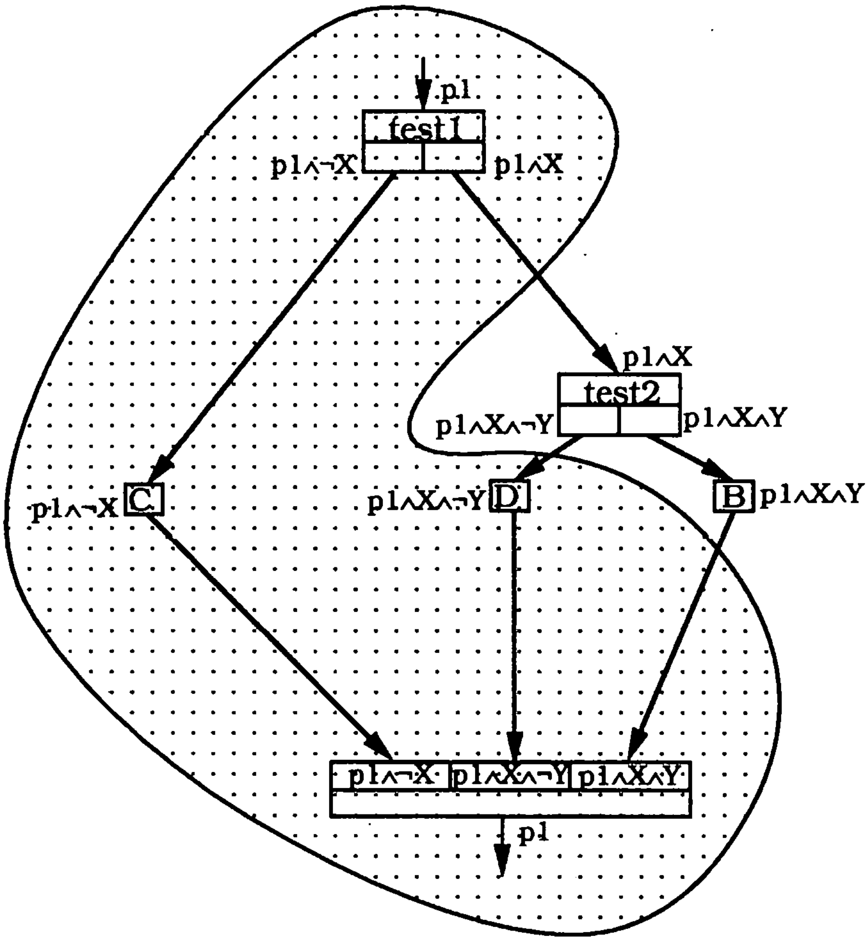
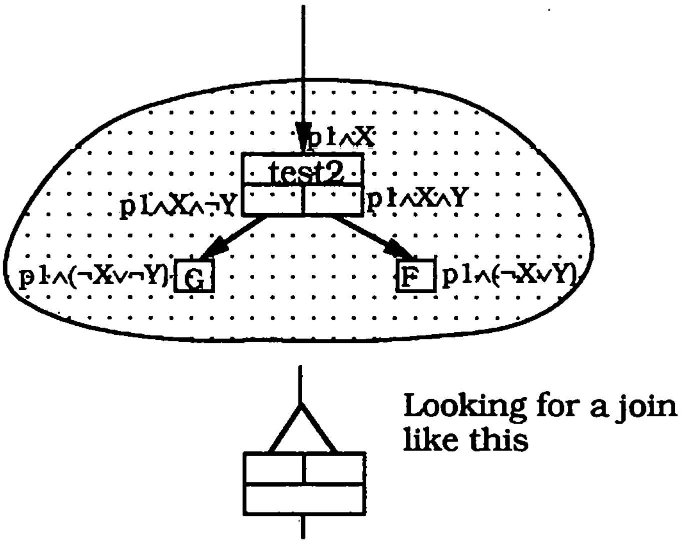


Figure 4.19
G Found In Optimised Graph



Looking for a join like this

Figure 4.20
Trying to Find a Z

conditions of the corresponding branches of the tests in the plan. As stated earlier this cannot alter the behaviour of the graph. In this case this would result in:

U	$P1 \wedge X \wedge \neg Y$
V	$P1 \wedge X \wedge Y$
W	$(P1 \wedge X \wedge \neg Y) \vee (P1 \wedge X \wedge Y) = P1 \wedge X$

Before continuing with the analysis, it should be noted that the input data objects to this join are identical, and hence the output is the same as the output from both of the two sub-plans.

If we make these substitutions into the plan condition we get:

$$((P1 \wedge X \wedge \neg Y) \leftrightarrow (P1 \wedge X \wedge \neg Y)) \wedge ((P1 \wedge X \wedge Y) \leftrightarrow (P1 \wedge X \wedge Y))$$
$$\wedge ((P1 \wedge X \wedge \neg Y) \supset (P1 \wedge (\neg X \vee \neg Y))) \wedge ((P1 \wedge X \wedge \neg Y) \supset (P1 \wedge (\neg X \vee \neg Y)))$$
$$\wedge ((P1 \wedge X \wedge Y) \supset (P1 \wedge (\neg X \vee Y))) \wedge ((P1 \wedge X \wedge Y) \supset (P1 \wedge (\neg X \vee Y)))$$

which simplifies to:

true

giving us:

$P1 \wedge X$

for the controlling condition of the Z operation (since $P1 \wedge X$ is the controlling condition of its input situation which is the input situation of the Y test). First of all note that although the components of this were conditional, the environment in which it is finally recognised ($P1 \wedge X$) together with the test involved, actually imply the conditions of all the sub-plans, so it has ended up unconditional and can 'execute' within a normal non-generalised control flow environment.

This is good, but not quite what was hoped for, since although we have recognized that there is a Z in the program, it is within the conditional (X test) rather than being at "top-level". However, we can do better than this. Suppose the input tie-points (which we have not shown) to the Y-test come from before the X-test in the original surface plan. The test Y is really testing whether some condition holds of various data objects, so it is really checking whether some condition $C(\mathbf{B}_1(p_1,in(Y)),\ldots,\mathbf{B}_n(p_n,in(Y)))$ holds or not, where $p_1,\ldots,p_n$ represent the data objects involved and in(Y) represents the input situation to the Y test, and $\mathbf{B}_1,\ldots,\mathbf{B}_n$ represent the relevant behaviours of the objects $p_1,\ldots,p_n$. Saying that input tie-points come from before the test (say from situation in(X)) means that:

$$\mathbf{B}_1(p_1,in(Y))=\mathbf{B}_1(p_1,in(X)) \wedge \ldots \wedge \mathbf{B}_n(p_n,in(Y))=\mathbf{B}_n(p_n,in(X))$$

This means that the condition C satisfies:

$$C(\mathbf{B}_1(p_1,in(X)),\ldots,\mathbf{B}_n(p_n,in(X)))=C(\mathbf{B}_1(p_1,in(Y)),\ldots,\mathbf{B}_n(p_n,in(Y)))$$

Now suppose we define a situation $S_{succeed}$ by:

$$S_{succeed}=\begin{cases} in(X) \text{ if } C(\mathbf{B}_1(p_1,in(X)),\ldots,\mathbf{B}_n(p_n,in(X))) \text{ holds} \\ \bot \text{ otherwise} \end{cases}$$

and define another situation S_{fail} by:

$$S_{fail}=\begin{cases} in(X) \text{ if } \neg(C(\mathbf{B}_1(p_1,in(X)),\ldots,\mathbf{B}_n(p_n,in(X)))) \text{ holds} \\ \bot \text{ otherwise} \end{cases}$$

Now the situations in(X), $S_{succeed}$, and S_{fail}, and the condition $C(\mathbf{B}_1(p_1,in(X)),\ldots,\mathbf{B}_n(p_n,in(X)))$ satisfy the axioms for tests, from which we can deduce that there exists a test with input situation in(X), and succeed and fail situations given by $S_{succeed}$, and S_{fail}, and with condition $C(\mathbf{B}_1(p_1,in(X)),\ldots,\mathbf{B}_n(p_n,in(X)))=Y$. So we have deduced the existence of a Y test, with controlling condition P1 (the same as the X test). Note that this test has the same inputs as the original test, so will still satisfy the data flow constraints that the original test satisfied.

Under most circumstances this is not a particularly useful deduction to make, since plans involving this new test are usually just "pulled back" into where they would have been had we not made the deduction by the controlling conditions of the other actions in the plan. However, in this case, when we use this test, with the F and G plans found earlier we get the substitutions:

P	$P1$
Q	$P1\wedge\neg Y$
R	$P1\wedge Y$
S	$P1\wedge(\neg X\vee\neg Y)$
T	$P1\wedge(\neg X\vee Y)$
U	$P1\wedge\neg Y$
V	$P1\wedge Y$
W	$(P1\wedge\neg Y)\vee(P1\wedge Y)=P1$

giving us the plan condition:

$$((P1\wedge\neg Y)\leftrightarrow(P1\wedge\neg Y))$$

$$\wedge(P1\wedge Y)\leftrightarrow(P1\wedge Y))$$

$$\wedge((P1\wedge\neg Y)\supset(P1\wedge(\neg X\vee\neg Y)))$$

$$\wedge((P1\wedge\neg Y)\supset(P1\wedge(\neg X\vee\neg Y)))$$

$$\wedge((P1\wedge Y)\supset(P1\wedge(\neg X\vee Y)))$$

$$\wedge((P1\wedge(\neg X\vee Y))\supset(P1\wedge(\neg X\vee Y)))$$

which also reduces to:

true

However, this time the environment of the resulting Z is P1, so we have recognised the same plan as we would have recognised had the surface plan been that of Figure 4.16.

Note that this analysis depended on the inputs to the Y-test coming from before the X-test. This can be checked by the parser. However, rather than having the parser do this every time it may want find more general plans, we adopt the following strategy. Before parsing surface plans, all tests are analysed and given the least strong controlling condition that is compatible with all their inputs. In practice this means that we look at all the input tie-points $t_1 \ldots t_n$ to each test, and look at the controlling conditions $c_1 \ldots c_n$ for the actions which produced these inputs. We then take as the controlling condition of the test the condition $c_1 \wedge c_2 \wedge \ldots \wedge c_n$. Since all the inputs to an action must come from environments which enclose the action, this will simplify down to the innermost environment of those producing the inputs to the test. Doing this would mean that the parser would have found the correct analysis of the surface plan above without need to deduce the existence of the test in a different environment from that in which it occurs, since this would have effectively been done before the parsing started.

Now we will consider an example in which both collapsing and the generalised control flow machinery are needed in order to recognise the plan. Consider the rule discussed earlier, shown in Figure 4.6, with plan condition:

$$[P \supset R \wedge J \supset R \wedge P \leftrightarrow J]$$
$$\wedge [Q \supset [S \wedge (T \leftrightarrow V \wedge V \leftrightarrow K \wedge T \leftrightarrow K) \wedge (U \leftrightarrow M \wedge M \leftrightarrow L \wedge U \leftrightarrow L)]]$$

and suppose the programmer has code resulting in the surface plan shown in Figure 4.21. First of all, note that the tests have already been treated as having the outermost controlling condition possible, as described in the previous example. Now note that we can collapse both instances of test1, and both instances of C, resulting in the graph shown in Figure 4.22. Now assuming the data flows (still not shown) are satisfactory the parser will identify the obvious nodes as being part of a Z plan, albeit rather a strange one. It will then attempt to evaluate the plan condition. The appropriate substitutions (given by the

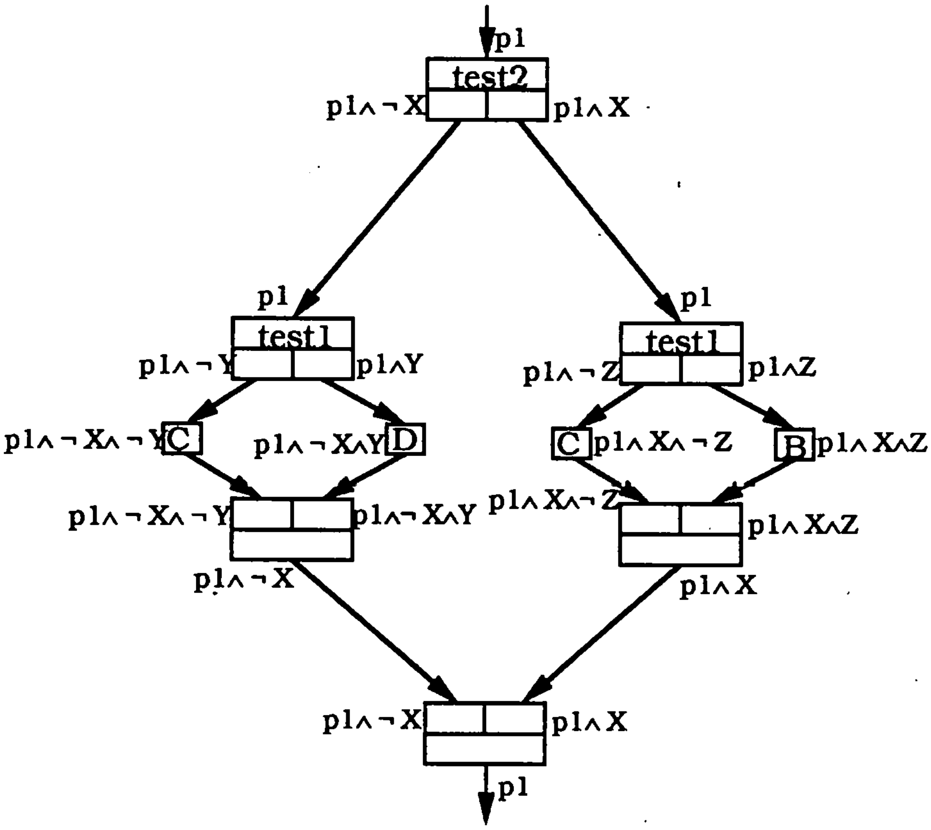


Figure 4.21
Surface Plan of 4.16, with Controlling Conditions of Tests Made as Global as Possible

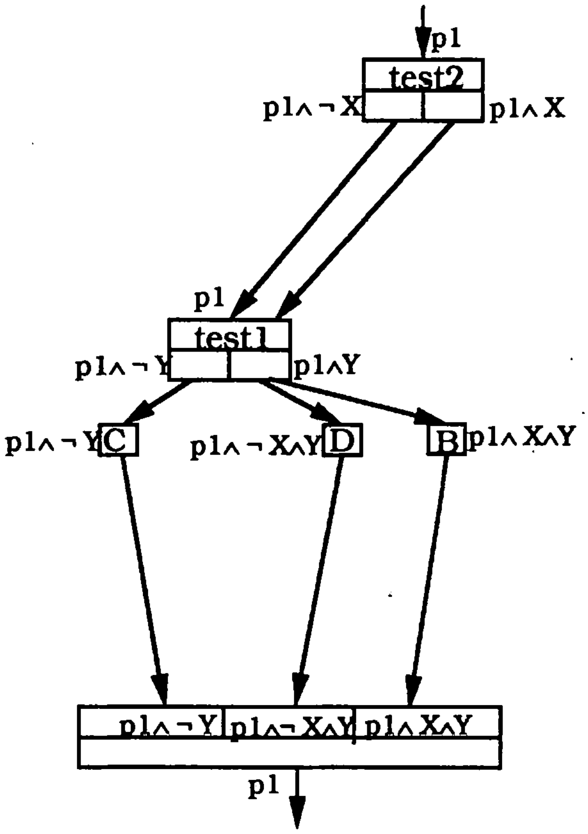


Figure 4.22
Collapsed 4.21

matching of nodes in the graph against nodes in the rule done during the data flow parsing) are:

W	$P1$
P	$P1\wedge\neg Y$
Q	$P1\wedge Y$
R	$P1\wedge\neg Y$
S	$P1$
T	$P1\wedge\neg X$
U	$P1\wedge X$
V	$P1\wedge\neg X\wedge Y$
M	$P1\wedge X\wedge Y$
J	$P1\wedge\neg Y$
K	$P1\wedge\neg X\wedge Y$
L	$P1\wedge X\wedge Y$
N	$P1$

Substituting into the plan condition gives us

$$\begin{aligned}
&[((P1\wedge\neg Y)\supset(P1\wedge\neg Y)) \wedge ((P1\wedge\neg Y)\supset(P1\wedge\neg Y)) \wedge ((P1\wedge\neg Y)\leftrightarrow(P1\wedge\neg Y))]\\
&\wedge [(P1\wedge Y)\supset\\
&\qquad [P1 \wedge [((P1\wedge\neg X)\leftrightarrow(P1\wedge\neg X\wedge Y)) \wedge ((P1\wedge\neg X\wedge Y)\leftrightarrow(P1\wedge\neg X\wedge Y))\\
&\qquad\qquad \wedge ((P1\wedge\neg X)\leftrightarrow(P1\wedge\neg X\wedge Y))]\\
&\qquad\quad \wedge [((P1\wedge X)\leftrightarrow(P1\wedge X\wedge Y)) \wedge ((P1\wedge X\wedge Y)\leftrightarrow(P1\wedge X\wedge Y))\\
&\qquad\qquad \wedge ((P1\wedge X)\leftrightarrow(P1\wedge X\wedge Y))]]]
\end{aligned}$$

which simplifies to *true*. So this is an unconditional plan and its controlling condition is given by P1. In other words, despite the lack of optimisation done by the programmer, we have managed to recognise the standard plan. This illustrates very nicely the power of collapsing and generalised control flow environments.

Now it may be thought that this involves manipulating rather nasty propositional formulae. However, there are two points to note. Firstly, the plan condition for rules can be worked out in advance of parsing, so this is a one-off expense. Secondly, although in principle doing this kind of simplification to propositional formulae is NPcomplete (it involves satisfiability as a special case) there are various redeeming factors which make it not too bad in practice. The first of these is that in most cases plans do not turn up in these bizarre ways, and when they do the "contortions" seem to be fairly localised. This means there are not usually too many variables involved, so the formulae are usually quite small (or at least all have common prefixes, which can be lumped together into a single variable like the variable P1 that we have been using in the examples, which means we can in practice deal with small formulae). The second is that plan conditions normally break down into well defined sub-expressions corresponding to the maximal sets of interdependent variables discussed above, and these sub-expressions can be simplified independently, before evaluating the entire plan condition. The third redeeming feature is that, as stated already, and as can be seen in the worked examples above, a great many of the expressions involved are either trivial to verify, or can be checked syntactically, by seeing if one expression is a prefix of another.

Chapter 5.

Chart Parsing of Flowgraphs

5.1 Introduction and Motivation.

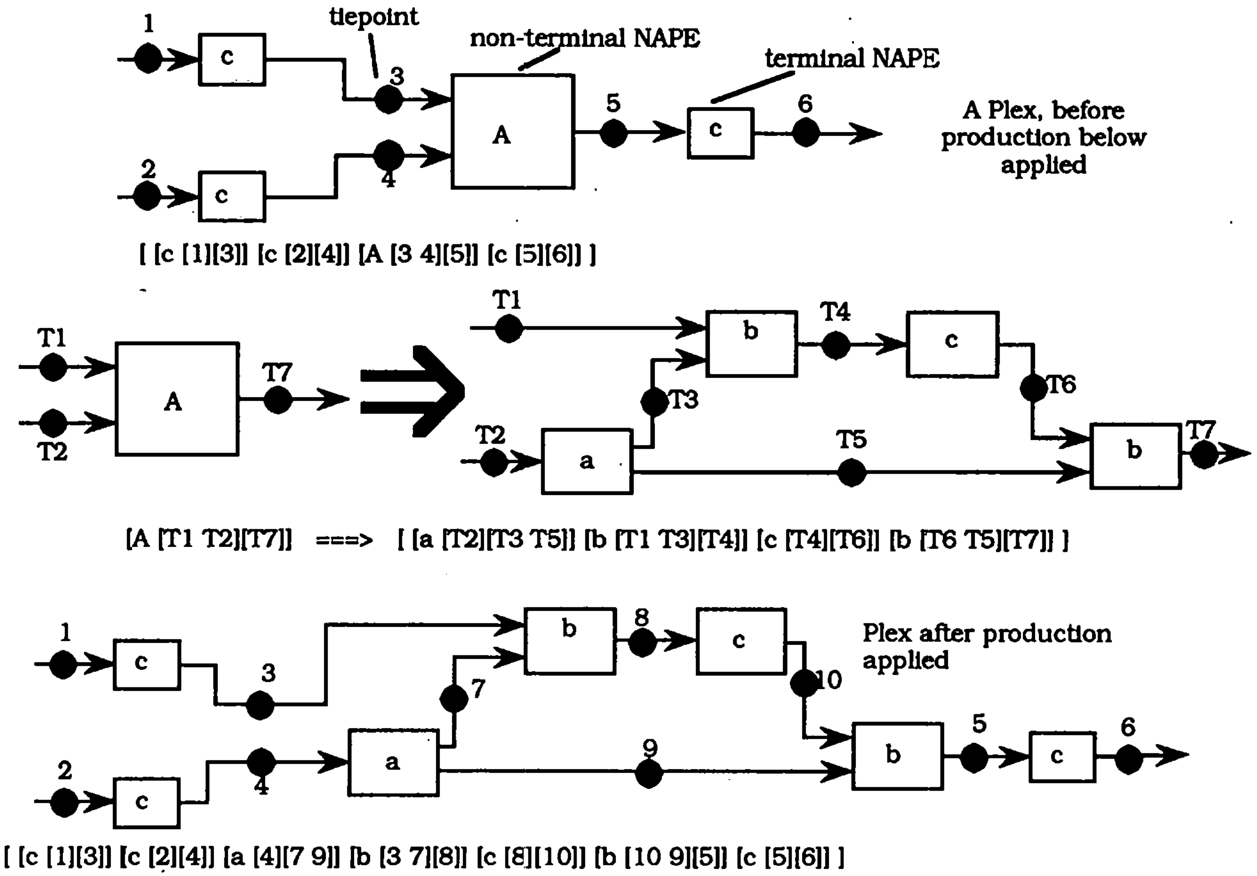
Many applications make use of diagrams to represent complex objects. Examples are electrical circuit diagrams, as well as the surface plans we are considering here. In such applications it is often necessary to systematically recognise how some diagram has been pieced together from other diagrams. This is analogous to the parsing problem for strings, and this chapter will present a generalisation of chart parsing [Thompson and Ritchie, 1984] able to cope with the case where the object being parsed is some kind of diagram (a flowgraph) and the grammar is an appropriate type of graph grammar (a flowgraph grammar). Often the various components of the diagrams can be regarded as producers of values which are fed as inputs to other components which in turn produce values to be passed on elsewhere. A feature that often occurs is structure sharing, when one component feeds one or more of its results to more than one other component (fan-out). In this situation the source component can be viewed as playing more than one role in the whole structure, and could have been duplicated so that separate copies of the component were responsible for each of these roles. This leads to no change in functionality, although there may be a loss in efficiency as measured by the number of components (electrical circuit case), or computational effort and code size (plan diagram case). This chapter also discusses the problem of diagram recognition in the case where structure sharing is allowed, noting that we want to permit structure sharing, but not enforce it.

The symmetric case of structure sharing arising through fan-in, rather than fan-out is not dealt with explicitly in this chapter.

However, the parsing algorithm is easily modified to cope with it, the necessary modifications to the algorithm being similar to those needed for fan-out.

5.2 Notation and Definitions.

Flowgraphs and flow grammars will be defined as special cases of plex languages and plex grammars first studied by Feder [1971]. Much of the terminology used will be borrowed from that for conventional string languages and grammars, and readers unfamiliar with this are referred to [Aho and Ullman, 1977]. A plex is a structure consisting of labelled nodes having an arbitrary number, n, of distinct attaching points, used to join nodes together. A node of this kind is called an n-attaching point entity (NAPE). Attaching points of NAPEs are not connected directly together, but are connected via intermediate points known as tie-points. A single tie-point may be responsible for connecting together two or more attaching points. If the direction of the connections is important then the plex is known as a directed plex. Many types of graph structure (e.g. webs [Pfaltz and Rosenfeld, 1969, Rosenfeld and Milgram, 1972], directed graphs, and indeed, strings) can be regarded as special cases of directed plexes. We will only consider the special case of directed plexes in which each NAPE's attaching points (from now on called ports) are subdivided into two mutually exclusive groups, known as input ports (restricted to only have incoming connections) and output ports (restricted to only have outgoing connections). We will further restrict ourselves to the special case in which each port of a NAPE is only connected to a single tie-point. This type of plex will be called a flowgraph and is a generalisation of Brotsky's use [1984] of the term. See Figure 5.1 (top) for an example of a simple flowgraph.



[[c [1][3]] [c [2][4]] [A [3 4][5]] [c [5][6]]]

[A [T1 T2][T7]] ===> [[a [T2][T3 T5]] [b [T1 T3][T4]] [c [T4][T6]] [b [T6 T5][T7]]]

[[c [1][3]] [c [2][4]] [a [4][7 9]] [b [3 7][8]] [c [8][10]] [b [10 9][5]] [c [5][6]]]

Figure 5.1
Simple Flowgraph and Rules

Just as a set of strings constitutes a language, so a set of plexes constitutes a plex language, and it is possible to define a plex grammar and the plex language generated by a plex grammar. Similar remarks apply to flowgraphs, webs, and graphs etc.

A production in a string grammar specifies how one string may be replaced by another, either in producing strings or in recognising them. In plex grammars the same is true but we encounter a difficulty (due to the 2-dimensional nature of plexes) not apparent in the string case. In the string case a production like

A ==> aXYb

applied to a string

....dAe..... (say)

results in the string

....daXYbe.....

and the question of how the replacement string is to be embedded in the host string in place of **A** never arises because there is a single obvious choice i.e. whatever is to the left of **A** in the original string is to the left of the replacing string, and similarly on the right. In the graph case we no longer have this simple left-right ordering on the NAPEs and this question of embedding becomes much more complicated. Most of the discussion of this topic is in the web and graph grammar literature (e.g. [Pfaltz and Rosenfeld, 1969, Rosenfeld and Milgram, 1972]), but most of it applies (with some slight modifications) to the flowgraph case as well. The approach taken here is to specify with each production which tie-points on the left hand side correspond to which tie-points on the right and then connect everything connecting to one of these left hand tie-points (from the surrounding subgraph) to its corresponding right-hand tie-point.

We define a flowgraph grammar **G** to be a 4-tuple **(N,T,P,S)** where:

N is a finite non-empty set of NAPEs known as nonterminals.

T is a finite non-empty set of NAPEs known as terminals.

P is a finite set of productions.

S is a special member of **N** known as the initial (or start) NAPE

and the intersection of **N** and **T** must be empty.

If we arbitrarily order the input and output ports of a NAPE then each NAPE in a flowgraph can be represented in the form of a triple

(NAPE-label, input list, output list)

where NAPE-label is the label on the NAPE, and input list is a list in which the i^{th} entry is the tie-point to which the i^{th} input port is connected. Similarly the output list specifies to which tie-point each of the output ports is connected. Using this convention a complete flowgraph **G** can be represented as a set $\mathbf{G}^{\mathbf{c}}$ (known as the component set) of such triples.

With the above conventions the productions in a flowgraph grammar have the general form

$\mathbf{A\ L_i\ L_o \Longrightarrow C\ R_i\ R_o}$

where

A is known as the left-side structure, represented as a component set

C is known as the right-side structure, represented as a component set

$\mathbf{L_i}$ is the left-side input tie-point list

$\mathbf{R_i}$ is the right-side input tie-point list

$\mathbf{L_o}$ is the left-side output tie-point list

and $\mathbf{R_o}$ is the right-side output tie-point list.

$\mathbf{L_i}$ and $\mathbf{R_i}$ must be of the same length, as must $\mathbf{L_o}$ and $\mathbf{R_o}$, and specify how an instance of the right-side structure is to be embedded into a structure **W** containing an instance of the left-side structure which is being rewritten according to the production. We define the *arity* of the left side of the rule to be the ordered pair $(|\mathbf{L_i}|, |\mathbf{L_o}|)$ and the arity of the right side of the rule to be the ordered pair $(|\mathbf{R_i}|, |\mathbf{R_o}|)$. So this requirement simply states that the left- and right-side arities must be the same. The rewriting and embedding is done as follows:

The instance of the left-side structure is removed from **W** and replaced by an instance of the right-side structure. Now, for each tie-point **X** in $\mathbf{L_i}$ any previous connections from NAPEs in **W** to **X** are replaced by connections from the same attaching points of the same NAPEs to the corresponding tie-point in $\mathbf{R_i}$. The same is done for tie-points in $\mathbf{L_o}$ and $\mathbf{R_o}$. Note that one can eliminate the need for explicit storing of $\mathbf{R_i}$ and $\mathbf{R_o}$ by simply using the same variable names on the left and right hand sides of the production to denote corresponding tie-points.

Just as in the string case, by considering various restrictions on the form of **X** and **Y** in a production of the form:

X ==> Y

one can arrive at the notions of context-sensitive, context-free, and regular languages [Ehrig, 1979]. In particular, restricting the productions to have a single NAPE in their left-side structure gives us the flowgraph equivalent of context-free string languages, and we will only concern ourselves with these from now on. In this case we no longer need to store $\mathbf{L_i}$ and $\mathbf{L_o}$ since the input and output lists of the single triple on the left of the production already specify this information. See Figure 5.1 for an example of the notation and of the rewriting process.

5.3 Chart Parsing of Context-free Flowgraphs.

In a chart parser, assertions about what has been found by the parsing algorithm are kept in a "database" known as thc chart. Such assertions will be called covering patches (or simply patches), and are of two kinds - complete patches and partial patches. A complete patch is a statement that a complete grammatical entity (corresponding to some terminal or non-terminal symbol of the grammar) has been found. Partial patches are assertions that part of some grammatical entity has been found, and about what would need to be found in order to complete the grammatical entity concerned. One can think of a patch as being a closed loop drawn round some subgraph of the flowgraph, indicating that this subgraph corresponds to all or part of some grammatical entity as defined by the grammar. If we regard the right-side structures of rules as uninstantiated templates, then complete patches with non-terminal labels correspond to the occurrence of an instantiation of the right-side structure of some rule, thus forming an occurrence of the left-side structure of the rule. Partial patches correspond to partially instantiated instances of the right-side structure of some rule, and thus to partially recognised instances of the left-side structure of the rule. Each patch A contains the following information:

1) label(A) - the name of the grammatical entity corresponding to the patch, and is always one of the terminal or non-terminal symbols of the grammar.

2) inputs(A) - a set of input tie-points for the patch.

3) outputs(A) - a set of output tie-points for the patch.

4) components(A) - a list of the other patches involved in making up this patch i.e. what other patches have been used to recognise this patch.

5) needed(A) - a description of what else needs to be found to complete the patch. In the case of a complete patch this will be empty, and for partial patches will be a flowgraph structure, represented as a list of triples.

For a partial patch, the input and output tie-points (i.e. those by which the patch connects to the surrounding flowgraph) are each subdivided into two categories - the set of active tie-points where the patch itself is still seeking other components to attach to these tie-points, and the set of inactive tie-points which are those which would be inputs or outputs of the patch were it complete. A NAPE needed by a partial patch will be called immediately needed if any of its tie-points are active. The components entry of a patch lists (instantiated versions of) those NAPEs in the right-side structure of the rule which have been completely instantiated, and the needed entry lists uninstantiated (as yet) parts of the rule. Note that some of the tie-points in the needed entry may be instantiated. These are where the needed NAPEs connect to the ones already found. We will say that a partial patch A is extendible by a complete patch B (or that B can extend A) in the case where A immediately needs a patch of the same type as B and the instantiated tie-points in this needed patch do not conflict with any instantiations actually occurring in B.

The essence of the chart parsing strategy can then be stated as follows:

Every time a complete patch is added to the chart a search is made for any partial patches immediately needing a patch of the sort just added at the appropriate place. For each of these partial patches a new patch is made extending it by the complete one, and this new patch is then added to an agenda of patches to be processed at some appropriate time. Similarly, every time a partial patch is added to the chart a search is made for any complete patches which could be used to extend the partial patch just added, and if any are found new patches are made which extend the partial one, and these are added to the agenda to be processed when appropriate. Note that patches are only ever added to the chart. They are never removed, thus avoiding the need to redo work that has been done before.

It should be clear from this that the basic operation of the algorithm is that of joining a complete patch to a partial patch to make a new enlarged patch. Figure 5.2 shows a partial patch being joined to a complete patch to make a new patch (the enclosing box). The resulting patch has the same items in its components entry as the original partial patch plus the complete patch. Its needed entry is equal to that

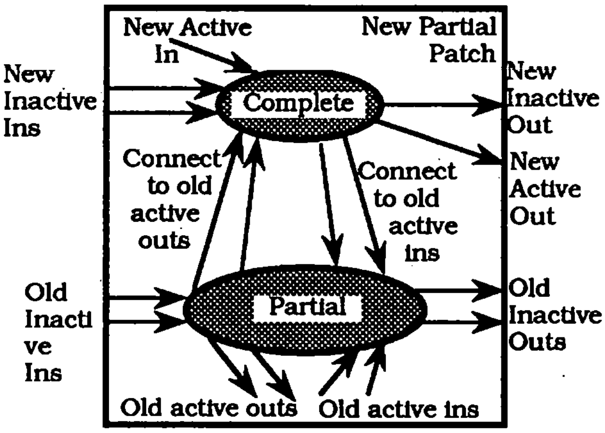
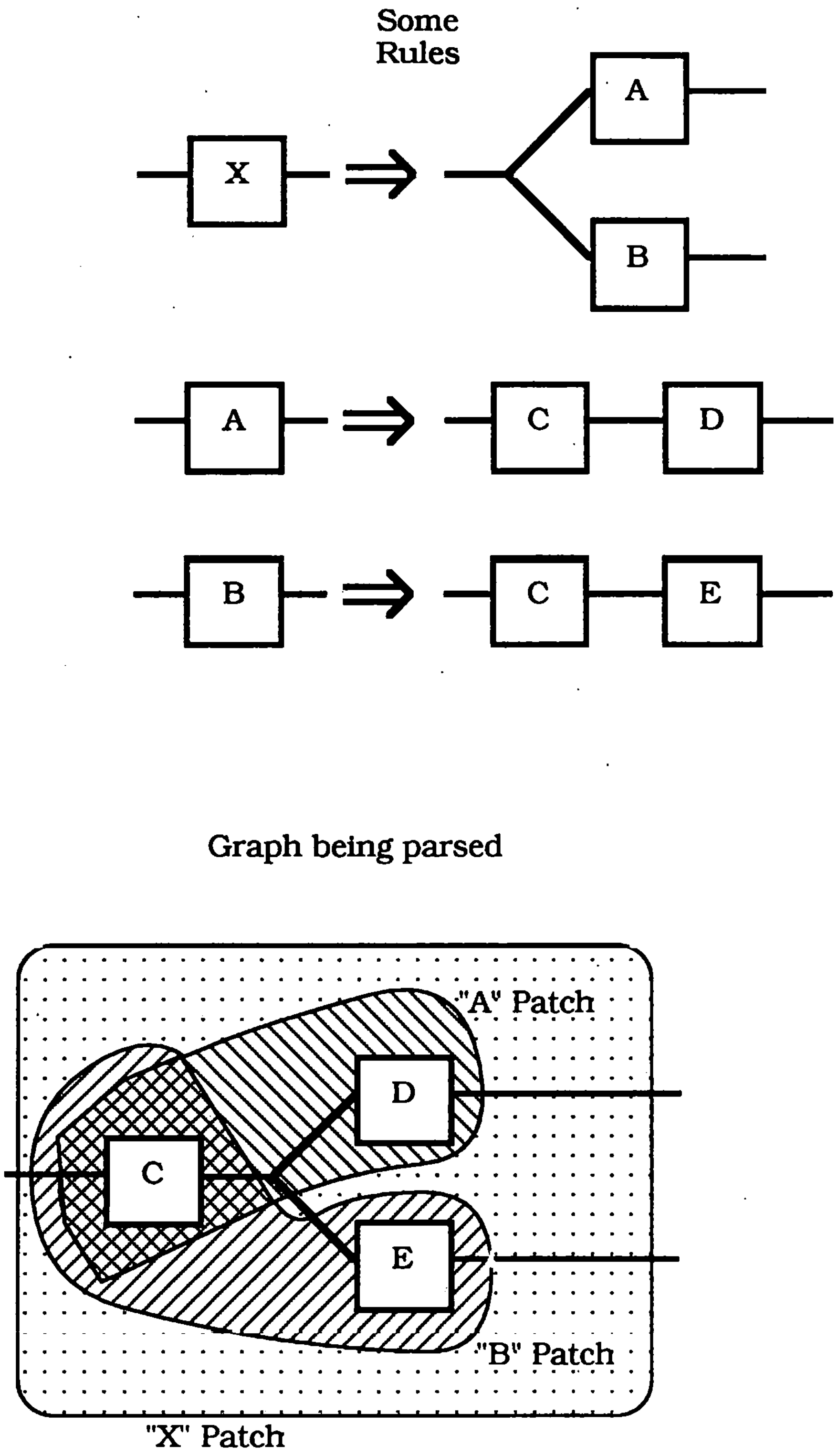


Figure 5.2
The Joining Operation

of the original partial patch minus the needed patch corresponding to the complete patch. Note that the matching of a needed patch to an actual complete patch may introduce further instantiations of tie-points in the needed entry of the new patch. On connecting the two patches all the inactive tie-points of the partial patch remain inactive. Some of its active tie-points will correspond to tie-points of the complete patch (this is where the two patches actually join). Other active tie-points remain active in the new patch since it is still looking for other patches to attach to them. Of the complete patch's (input and output) tie-points some have already been mentioned i.e. those connecting directly to the partial patch. Others will become new inactive tie-points of the resulting patch since it will not be looking for anything to attach to them. However other (input and output) tie-points of the complete patch may now become active (viewed as belonging to the new patch) since it may now expect other patches to attach to them in order to complete itself. Provided all these distinctions are kept clear there is no great difficulty in implementing the joining operation.

One further point ought to be made here. With the joining operation as just described a certain limited sort of structure sharing happens automatically. This is illustrated in Figure 5.3. If we wish to prevent this then when trying to extend a partial patch P by a complete patch C, the parser must check (recursively!) that none of the components of P have any sub-components in common with C. If this is done, then no structure sharing at any level can arise. This check will be referred to as the *no-sharing check*.

The initialisation of the chart and the agenda now needs to be described. To begin with a complete patch is made for each of the terminal NAPEs in the original graph, and these are added to the



Some Rules

Graph being parsed

Figure 5.3
Occurence of Structure Sharing Without No Sharing Check

agenda. If the algorithm is to be run top-down then an additional step is needed in which partial patches with empty components entries are made for every rule in the grammar whose left-side structure is labelled by the start symbol of the grammar. Each such rule leads to several such empty patches, one for each permutation of the input tie-points of the original graph. The inactive-inputs and active-outputs entries for each of these patches are the permuted inputs. The needed entry is just the right-side structure of the rule with any appropriate instantiations of the tie points occurring in it. These patches are also added to the agenda.

The complete algorithm is shown below:

```
initialise chart and agenda;
until the agenda is empty do
    pick a patch A from the agenda;
    unless A is already in the chart then
        add A to the chart;
        if A is complete then
            for each partial patch B in chart extendible by A do
                make a new patch extending B with A and put on agenda;
            endfor;
            if bottom-up then
                for each rule R in P such that rhs(R) has an input NAPE labelled by
                                                                  label(A) do
                  for each such NAPE X in R do
                     make new empty patch B with label(B)=lhs(R) and
                        needed(B)=rhs(R) with instantiations dependent on match between
                                                                  X and A and
                        inputs(B)=inputs(A) and
                        active-outputs(B)=inputs(A);
                     add B to agenda;
                  endfor;
                endfor;
            endif;
        else
            for each complete patch B in chart which can extend A do
                make a new patch extending A with B and put on agenda;
            endfor;
            if top-down then
                for each object C immediately needed by A do
                     for each rule R in P with lhs(R)=label(C) do
                           make new empty patch B with label(B)=label(C) and
                                 needed(B)=rhs(R) with instantiations dependent on match
                                                        between C and lhs(R) and
                                 inputs(B)=inputs(C) and
                                 active-outputs(B)=inputs(C);
                           add B to agenda;
                     endfor
                endfor
            endif
        endif
    endunless
enduntil;
```

When this algorithm terminates the parse is regarded as successful if the chart contains a complete patch for **S** whose inputs and outputs entries are the same as the input and output tie-points of the graph being parsed.

The only remaining issue is how to organise the chart so that it can be searched efficiently. The chart is first of all divided into two parts, one for complete patches, and one for partial. The part for complete patches is organised as two arrays, one for indexing each patch by its inputs, and one for indexing by its outputs. So each complete patch is entered several times into the chart, once for each of its inputs and outputs. For further efficiency each of the elements in these arrays is a hash table and the patches are actually entered into these hashed by their label. This enables efficient retrieval of all patches with a particular label at a particular place in the graph. In a similar fashion partial patches are entered into their part of the chart indexed by their input and output tie-points, and hashed by the labels of each of the patches they immediately need. Note that there may be several of these.

Finally, note that a similar trick can be used to store the grammar rules themselves in order to enable efficient retrieval of appropriate rules.

5.4 Complexity Analysis

5.4.1 A Polynomial Bound

In this section a relatively informal argument will be given to show that the algorithm just presented runs in time polynomial in the size (measured by the number of tie-points T) of the graph being parsed. It is not the intention here to give a tight upper bound on the running time, but simply to show that it is indeed polynomial. So, let

G=number of NAPEs in the graph being parsed

T=number of tie-points in the graph being parsed

K=maximum number of inputs to a NAPE

M=maximum number of outputs from a NAPE

L=number of possible labels

R=number of rules in the grammar

Q=maximum number of NAPEs in the right-side structure of a rule

A=maximum possible number of active tie-points in a partial patch.

It should be noted that K, M, L, R, Q, and A all depend on the grammar, and are independent of the graph being parsed.

Now it should be noted that, for the purposes of adding new patches to the chart, patches are only distinguished according to some of the information contained in them, rather than strict equality being necessary. Complete and partial patches will be dealt with separately:

Complete patches are distinguished which differ in at least one of their input tie-points, their output tie-points, or their label. The maximum number of inputs and maximum number of outputs in a patch is determined by the grammar, as is the number of possible labels. So the number of possible complete patches in the chart is bounded above by the product of L and the number of possible ways of selecting at most K out of T tie-points, and the number of possible ways of choosing at most M out of T tie-points. This gives us $O(L.T^{K+M})$ complete patches altogether. Note also that a similar argument shows that at a given set of K (input) tie-points, there are *at most* $O(T^{M})$ complete patches with a given label.

Partial patches are distinguished which differ in at least one of their inactive input tie-points, their inactive output tie-points, their label, or in what they need in order to complete themselves (their <u>needed</u> entry). Now, a partial patch essentially represents the partially recognised right side structure of a rule. The rule used determines the label, and there are at most 2^{Q} subsets of the (at most) Q NAPEs in the rule that could still be needed. Each such subset determines a set of (at most) A active tie-points for the patch. So there can be at most $O(R.2^{Q}.T^{A}.T^{K}.T^{M}) = O(R.2^{Q}.T^{A+K+M})$ partial patches altogether. In fact there will be very much less than this, as this includes complete patches with nothing needed, and (more importantly) ignores completely additional constraints implied by the connectivity of the graph concerned.

Now a further point needs to be dealt with. The basic operation of the chart parsing algorithm involves extending partial patches by complete ones. So for a given partial patch we need to know what is the largest number of complete patches that could possibly extend it. The partial patch can be extended at any of its (at most) A active tie-points, and any complete patch which could extend it must join at least one of these tie-points, and must share a label with at least one of the NAPEs immediately required by the partial patch. So there are *at most* $O(A.Q.T^{K+M-1})$ such complete patches that need to be considered. Similarly, given a complete patch, it can be seen that there are at most $O((K+M).R.2^{Q}.T^{K+M+A-1})$ possible matching partial patches.

Now we can use these upper bounds to demonstrate both that the algorithm terminates and that it does so in polynomial time. Suppose the algorithm is running, and let N denote the number of times we have been round the main loop.

Let:

C_N=number of complete patches in the chart after iteration N

P_N=number of partial patches in the chart after iteration N

A_N=length of agenda after iteration N

Then in the top-down case we have the following relations:

$A_0=G+R_S$.(number of permutations of inputs of graph) where R_S is the number of rules for S (the start symbol).

$C_0=0$

$P_0=0$

and for the (N+1)th iteration

$$A_{N+1} \begin{cases} = A_N-1 & \text{if patch is already present in chart} \\ <=A_N-1+AQT^{K+M-1}+QR & \text{if patch partial and not in chart} \\ <=A_N-1+(K+M).R2QT^{K+M+A-1} & \text{if patch complete and not in chart} \end{cases}$$

$$C_{N+1} \begin{cases} = C_N & \text{if patch chosen is already present in chart} \\ = C_N & \text{if patch chosen is partial and not in chart} \\ = C_N+1 & \text{if patch chosen is complete and not in chart} \end{cases}$$

$$P_{N+1} \begin{cases} = P_N & \text{if patch chosen is already present in chart} \\ = P_N+1 & \text{if patch chosen is partial and not in chart} \\ = P_N & \text{if patch chosen is complete and not in chart} \end{cases}$$

In the bottom-up case we have the following relations:

$A_0=G$

$C_0=0$

$P_0=0$

and for the (N+1)th iteration,

$$A_{N+1} \begin{cases} = A_N\text{-}1 & \text{if patch chosen is already present in chart} \\ <= A_N\text{-}1+A.Q.T^{K+M-1} & \text{if patch partial and not in chart} \\ <=A_N\text{-}1+(K+M).R.2^Q.T^{K+M+A-1}+QR & \text{if complete and not in chart} \end{cases}$$

$$C_{N+1} \begin{cases} = C_N & \text{if patch chosen is already present in chart} \\ = C_N & \text{if patch chosen is partial and not in chart} \\ = C_N+1 & \text{if patch chosen is complete and not in chart} \end{cases}$$

$$P_{N+1} \begin{cases} = P_N & \text{if patch chosen is already present in chart} \\ = P_N+1 & \text{if patch chosen is partial and not in chart} \\ = P_N & \text{if patch chosen is complete and not in chart} \end{cases}$$

Note that this means that in both the bottom-up case and the top-down case both C_N and P_N are monotonic functions (as a function of N). As discussed earlier both are bounded above. Therefore after some number of iterations they must both hit their maximum value (which will normally be <u>much less</u> than the crude figures given above). Once this happens all patches on the agenda must be already present in the chart and A_N decreases by one on each subsequent iteration until it reaches 0 (an empty agenda), and the algorithm terminates. Now on each iteration it can be seen that either:

(i) both C_N and P_N remain constant (in which case A_N decreases)

or (ii) P_N increases by 1, and items are possibly added to the agenda

or (iii) C_N increases by 1, and items are possibly added to the agenda.

Now, from the above it can be seen that at most $O(L.T^{K+M})$ iterations involve adding a complete patch to the chart and add some items to the agenda, and at most $O(R.2^Q.T^{A+K+M})$ iterations involve adding a partial patch to the chart and add some items to the agenda. All the other iterations simply remove items from the agenda. So how many items get added to the agenda?

This is given by:

(no. of items in initial agenda)+(no. added for complete patches)+(no. added for partial patches)

In the top down case this is:

$$<= A0 + O(L.T^{K+M}).O((K+M).R.2^{Q}.T^{K+M+A-1}) + \\ O(R.2^{Q}.T^{A+K+M}).(O(A.Q.T^{K+M-1})) + Q.R)$$

which clearly has a polynomial bound. Similarly, in the bottom-up case this is:

$$<= A0 + O(L.T^{K+M}).(O((K+M).R.2^{Q}.T^{K+M+A-1}) + Q.R) + \\ O(R.2^{Q}.T^{A+K+M}).O(A.Q.T^{K+M-1})$$

which again is clearly polynomially bounded. So, in both cases the number of items added to the agenda, which is the same as the number of iterations performed by the algorithm, is polynomially bounded. Now, how much work is done on each of these iterations? The cost of seeing if a patch is already in the chart can be done in polynomial time. This is because (even with no clever indexing) there are at most a polynomial number of patches in the chart that need to be checked. The cost of checking if one patch is extendible by another can be done in constant time[1] (depending on the grammar), as can the

[1] It depends on checking that the instantiated tie-points of the two patches are compatible with each other, and the number of tie-points involved depends on the grammar. If the no-sharing check is included, then the cost will no longer be constant, but can be done in time at most 2QG since, the partial patch can have at most Q components, each of which is ultimately made up of at most G NAPEs, and the complete patch is also made up of at most G NAPEs, at lowest level. Checking for intersection of two sets can be done in time linear in the sum of the sizes of the two sets. This is still polynomial, so does not affect the analysis.

cost of making a new patch. All the costs involved in checking rules etc. are purely a function of the grammar. So the total cost of the algorithm is easily seen to have an upper bound which is a polynomial function of T.

5.4.2 Finding All Parses

It should be noted that although the algorithm performs flowgraph recognition in polynomial time, it does not find all parses in polynomial time. This is because for some flowgraphs and some grammars there may well be an exponential number of parses (this is even true of Earley's algorithm operating on strings!). The algorithm will however find a parse if one exists. If an application requires all possible parses, then the algorithm can be modified to store any patch which is equal to one already in the chart in terms of its inputs, outputs, and label, but not equal in terms of its components, in an auxiliary data structure. At the end of the parsing there will then be enough information around in the chart and the auxiliary data structure to enable subsequent calculation of all possible parses.

5.5 Chart Parsing of Structure-Sharing Flowgraphs.

As stated in the introduction we are also interested in the case where structure sharing is allowed. To make this more precise we define a relation **collapses** on the set of flowgraphs over some set of NAPEs by:

G2 **collapses** G1 iff G1 and G2 are flowgraphs, and $G1^c$ contains two triples of the form $T1=(A,(t_1,\dots,t_n),(x_1,\dots,x_m))$ and $T2=(A,(t_1,\dots,t_n),(y_1,\dots,y_m))$, and $G2^c$ can be obtained from $G1^c$ by removing these two triples and replacing them by a single triple of the form $T3=(A,(t_1,\dots,t_n),(z_1,\dots,z_m))$ and then replacing all occurrences of $x_1,\dots,x_m$ and $y_1,\dots,y_m$ by $z_1,\dots,z_m$ respectively throughout the remaining triples. In other words, G2 **collapses** G1 iff G1 contains two instances of some NAPE A (say) which have the same inputs, and G2 is identical to G1 except that the two instances of A have been replaced by a single instance of A (with the same inputs) and all NAPES which originally connected to the outputs of one or other of the two instances of A now connect to the single instance (in G2). This amounts to identifying the two instances of A and their corresponding tie-points.

The reflexive, transitive, symmetric closure of **collapses** is then an equivalence relation (share-equivalence) on the set of flowgraphs, and we then want any parsing algorithm which can recognise some graph G to also be able to recognise any flowgraphs share-equivalent to G. We also want the grammatical formalism used to be able to generate not only the flowgraphs derivable directly from the grammar, but also all share-equivalent flowgraphs. This can be done if we allow at any point in the generation of a flowgraph the replacement of the graph so far generated (G1) by any graph G2 for which either G1 **collapses** G2 or G2 **collapses** G1. A flowgraph grammar with the addition of this rewriting rule will be referred to as a (full) structure sharing flowgraph grammar (a SSFG). Figure 5.4 illustrates several phenomena that can occur with SSFGs, and which motivated the above definition.

Now, it turns out that this idea of structure sharing flowgraph grammars is almost what we need to capture the structure sharing that occurs in programs, except that, for reasons already discussed in Chapter 4, we do not want to allow any two NAPEs sharing the same inputs to be collapsed, but only NAPEs with appropriate labels. To accomplish this we define a slightly more general notion. A restricted structure sharing flowgraph grammar (RSSFG) is a 5-tuple **(N,T,P,S,R)** where **N**, **T**, **P**, **S** are the same as for ordinary context free flowgraph

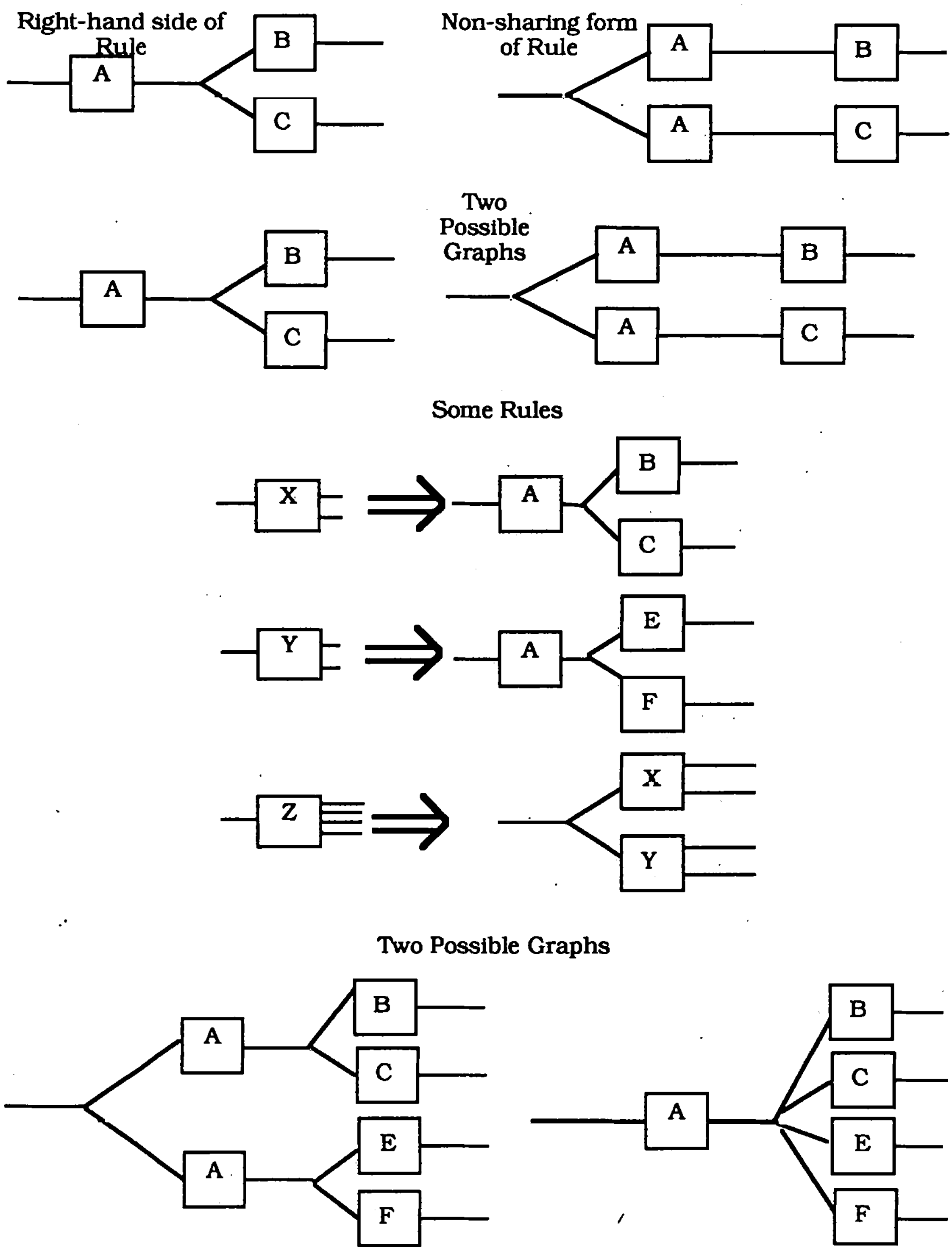


Figure 5.4
Structure Sharing and Collapsing Phenomena

grammars, and $\mathbf{N} \cup \mathbf{T} \supseteq \mathbf{R}$. **R** is the set of NAPEs for which collapsing is allowed, and we modify the above definition of collapses as follows:

G2 **R-collapses** G1 iff G1 and G2 are flowgraphs, and $G1^c$ contains two triples of the form $T1=(A,(t_1,\ldots,t_n),(x_1,\ldots,x_m))$ and $T2=(A,(t_1,\ldots,t_n),(y_1,\ldots,y_m))$, where $A \in \mathbf{R}$ and $G2^c$ can be obtained from $G1^c$ by removing these two triples and replacing them by a single triple of the form $T3=(A,(t_1,\ldots,t_n),(z_1,\ldots,z_m))$ and then replacing all occurrences of $x_1,\ldots,x_m$ and $y_1,\ldots,y_m$ by $z_1,\ldots,z_m$ respectively throughout the remaining triples. In other words, G2 **R-collapses** G1 iff G1 contains two instances of some NAPE A (whose label is in **R**) which have the same inputs, and G2 is identical to G1 except that the two instances of A have been replaced by a single instance of A (with the same inputs) and all NAPES which originally connected to the outputs of one or other of the two instances of A now connect to the single instance (in G2).

The reflexive, transitive, symmetric closure of **R-collapses** is also an equivalence relation (**R**-share-equivalence) on the set of flowgraphs, and we then want any parsing algorithm which can recognise some graph G to also be able to recognise any flowgraphs **R**-share-equivalent to G. We also want the grammatical formalism used to be able to generate not only the flowgraphs derivable directly from the grammar, but also all **R**-share-equivalent flowgraphs. This can be done if we allow at any point in the generation of a flowgraph the replacement of the graph so far generated (G1) by any graph G2 for which either G1 **R-collapses** G2 or G2 **R-collapses** G1. It is the addition of this rewriting rule which turns an ordinary flowgraph grammar into a restricted structure sharing flowgraph grammar (RSSFG). If $\mathbf{R}=\emptyset$ then the grammar is an ordinary flowgraph grammar, and if $\mathbf{R}= \mathbf{N}\cup\mathbf{T}$ then we get a (full) structure sharing flowgraph grammar as defined earlier.

To see how the chart parsing algorithm can be modified to cope with RSSFGs it should first be noted that for any flowgraph G there is a smallest flowgraph G_{min} which is R-share-equivalent to G. Secondly it should be noted that the right-side structure of any rule in a RSSFG can be replaced by any flowgraph R-share-equivalent to it without altering the generative capacity of the grammar. We can therefore define a canonical form for an RSSFG in which each rule of the form:

A ==> B

has been replaced by the rule:

A ==> B_{min}.

So the first change to the algorithm is actually to change the grammar to its canonical form, and to use this new form of the grammar for parsing. The second change is to the action of adding a complete patch to the chart. Previously the only check that was done was to see if the patch was already in the chart. Now the algorithm must additionally check that the label of the patch is in **R**, and that there is no other patch with the same label and the same inputs in the chart. If there is then the algorithm must collapse the new patch and the one that was there already into a single patch with a new set of output tie-points and identify the original outputs of the two patches with these new tie-points. Provided tie-points in the various triples making up the patches are represented as pointers to pointers to tie-points (rather than storing the tie-points directly in the triples) then simply changing the values of the second set of pointers will implement the identification universally throughout all patches in the chart. If the information that collapsing has been done is needed by an application the algorithm can make a note of this fact either by annotating the tie-points involved or by an assertion held separately. Finally, the no-sharing check must be omitted.

5.6 Degenerate Flowgraphs

With the algorithm just described there are several apparent anomalies that can occur. The following two examples were originally pointed out to me by Linda Wills (personal communication) as problems for the chart parser. However, in the light of the above discussion of structure sharing and collapsing of flowgraphs, they can be seen as natural and indeed desirable consequences of the theory provided that the other components of the program understanding system are capable of reasoning about the seemingly anomalous patches that are recognised.

5.6.1 Anomalous Example 1

The first example is where we have a rule:

A [t1 t2] [t3 t4] ==> [[b [t1] [t3 t5]] [d [t5 t2] [t4]]]

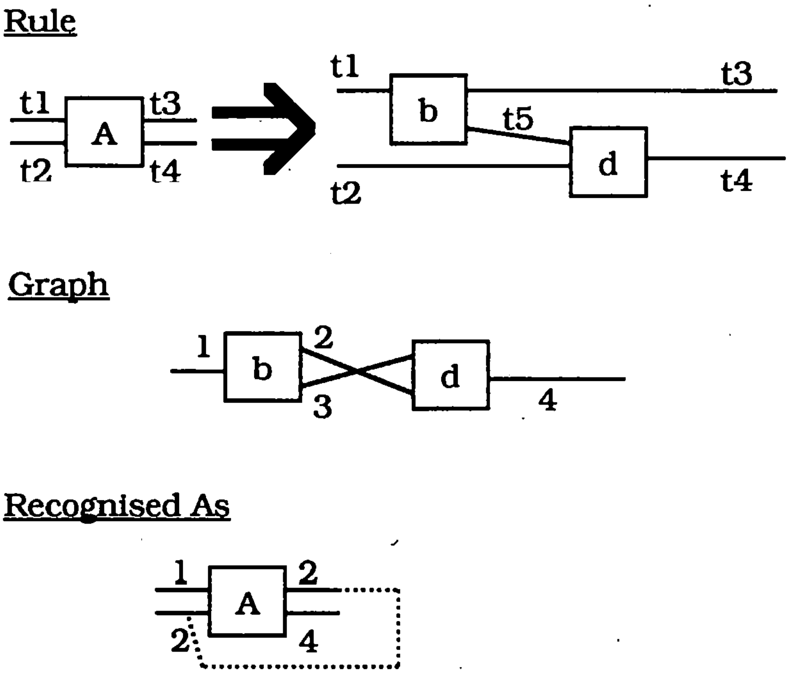
and input graph:

[[b [1] [2 3]] [d [3 2] [4]]]

which is recognised by the parser as:

[A [1 2] [2 4]]

in which a cyclic structure has been recognised even though the original graph had no cycles! Figure 5.5 shows the situation. Now, if we allow structure sharing and collapsing, then as can be seen in Figure 5.6, such a degenerate flowgraph arises quite naturally.



Rule

Graph

Recognised As

Figure 5.5
Anomolous Example 1

Anomolous Example 1 Arises
from collapsing this graph

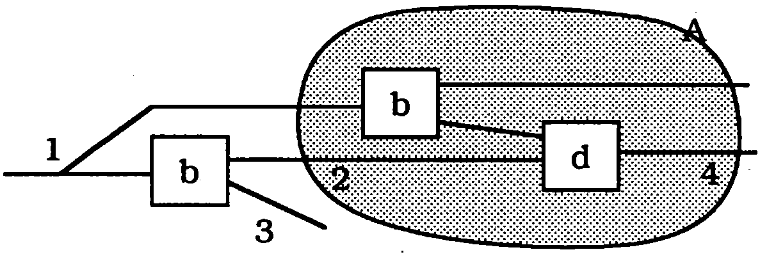


Figure 5.6
Explanation of Anomolous Example 1

5.6.2 Anomalous Example 2

The second example is where we have a rule:

```
A [t1 t2] [t3 t4] ==> [ [c [t1] [t5 t6] ]
                         [d [t5 t7] [t4] ]
                        [a [t6] [t3] ]
                        [a [t2] [t7] ]    ]
```

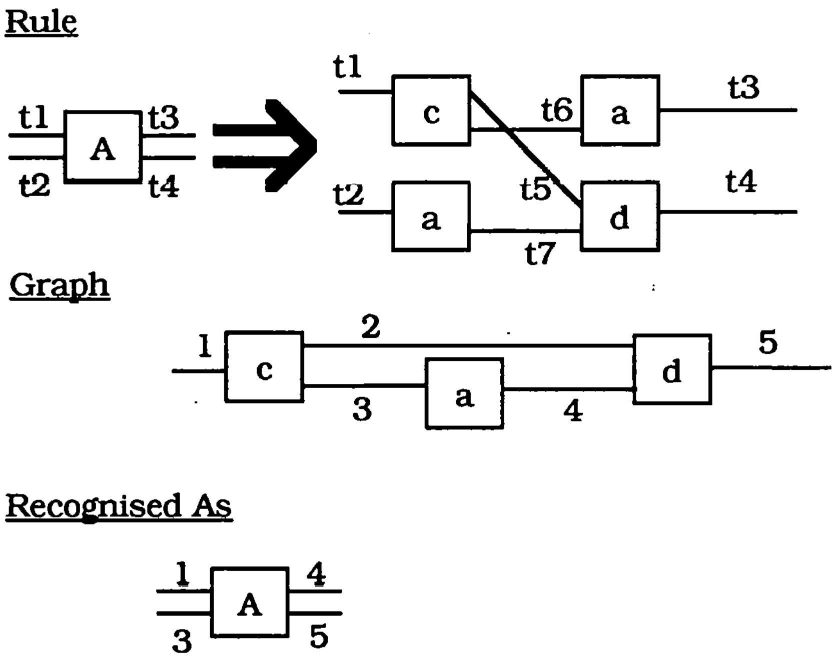
and we have input graph:

```
[ [c [1] [2 3] ] [a [3] [4] ] [d [2 4] [5] ] ]
```

In this case the parser recognises:

```
[A [1 3] [4 5] ]
```

This time we have no cycles, but one of the internal tie-points of a patch is also functioning as one of its inputs. Figure 5.7 shows this situation. Again, if we allow structure sharing and collapsing we can see that this phenomenon may arise quite naturally, as shown in Figure 5.8. If we interpret this in programming terms, it would appear that, in general, such degenerate flowgraphs represent the case where a programmer has realised that, in order to prevent a computation being done twice when some initialisation code before the call of some operation duplicates some of the internal details of the implementation of the operation, that she can expand the operation 'in-line' and remove redundant computations. In this case we do actually want the system to recognise the high level description of the operation even though the code (and graph) may look rather strange as an implementation of it.

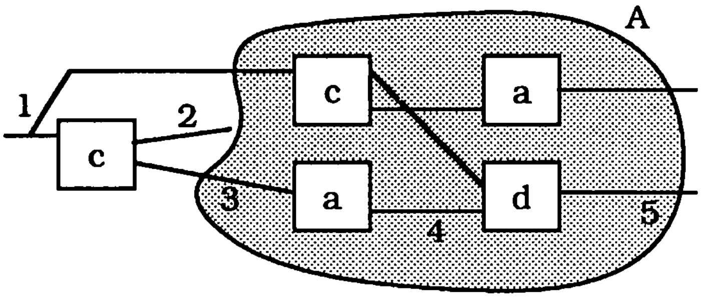

Rule

Graph

Recognised As

Figure 5.7
Anomolous Example 2

Anomolous Example 2 Arises from collapsing this graph



Collapse c then a

Figure 5.8
Explanation of Anomolous Example 2

5.7 Discussion

Although there is quite a lot of literature on the generative abilities of various types of graph grammar formalisms (see e.g. [Ehrig, 1979, Feder, 1971, Fu, 1974, Gonzalez and Thomason, 1978, Pfaltz and Rosenfeld, 1969, Rosenfeld and Milgram, 1972]), there is relatively little on parsing strategies, except for rather restricted classes of graph and web grammars [e.g. Della Vigna and Ghezzi, 1978]. In its top-down strictly left-to-right form chart parsing of context-free string languages corresponds to Earley's algorithm [Earley, 1970], which was generalised by Brotsky [1984] to parsing flowgraphs of the kind described here, except that his algorithm could not cope with fan-out at tie-points. However the approach taken here can also run bottom-up, which is particularly useful in applications in which we want to recognise as much as possible even though full recognition may be impossible (because of errors in the graph, or because the grammar is necessarily incomplete). Wills [1986, 1990] has modified Brotsky's algorithm to cope with fan-out, but her algorithm only runs in a pseudo-bottom-up fashion by starting it running top-down looking for every possible non-terminal at every possible place in the graph.

A particular advantage of a chart parser is that it quite explicitly keeps a record of all partial patches it finds. This is useful in applications for which we may not just wish to verify that some graph can be generated from some grammar, but also to enable the system to make suggestions based on "near-miss" information about how to correct the graph. It is in such applications that it may be useful to modify the algorithm to run right-to-left as well, since this may enable one to find more "near-misses" (i.e. those missing their start NAPES) than one would find if the parser only ran left-to-right.

5.8 Applications

The algorithm just described forms the basis for the plan recognition process performed by IDS. However, as will be seen in the next chapter, there are many features of the plan calculus that do not exactly fit the formalisms we have just described. Accordingly, the description of how this algorithm has to be modified to perform plan recognition will be delayed until Chapter 7 (after we have discussed the translation process from programs to surface plans), and Chapter 8 will give detailed examples of the plan recognition process on actual programs. However, as stated at the beginning of this chapter, there are other domains in which a similar ability to parse flowgraphs would be useful. In particular, digital circuit analysis is a domain which fits the structure-sharing flowgraph formalism especially well. We will illustrate this with an example. Consider Figure 5.9 which shows a circuit which performs addition of 3-bit numbers. The grammar shown in Figure 5.10 is capable of generating such a circuit, and the parser does indeed recognise this circuit as produced by the grammar. It would be interesting to try and build a tutoring system for digital circuit design based on this chart parsing algorithm, where near-miss information provided by the parser could again form the basis for guiding the tutoring strategy.

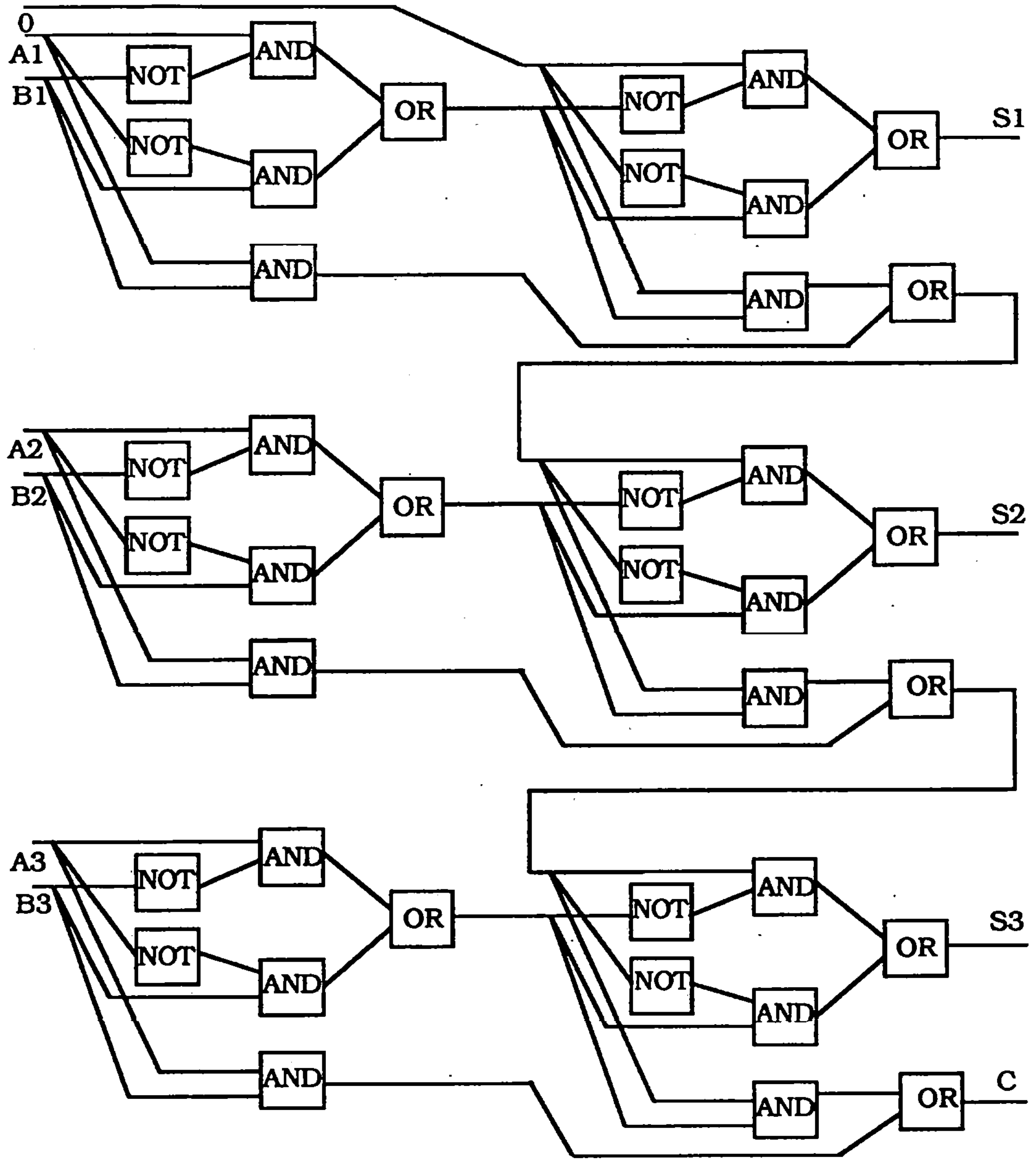


Figure 5.9
A 3-Bit Addition Circuit

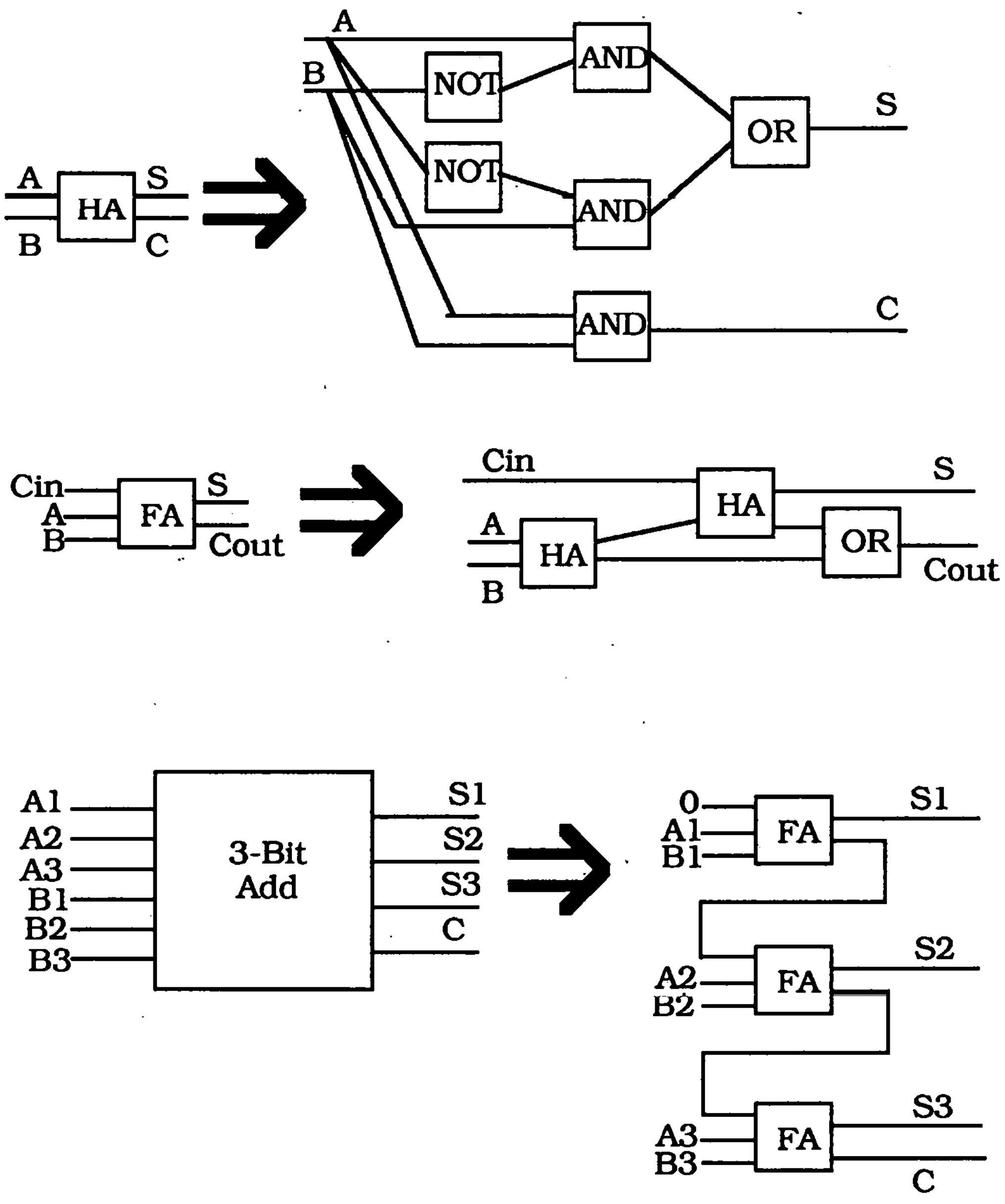


Figure 5.10
Addition Circuit Grammar

PART 3. IDS

Chapter 6.

An Overview of IDS, and The Translation Module

This chapter will give an overview of IDS - the main modules, how they interact, and the state of implementation etc. It will also discuss the translation from Pascal to surface plans.

6.1 Overall Structure of IDS

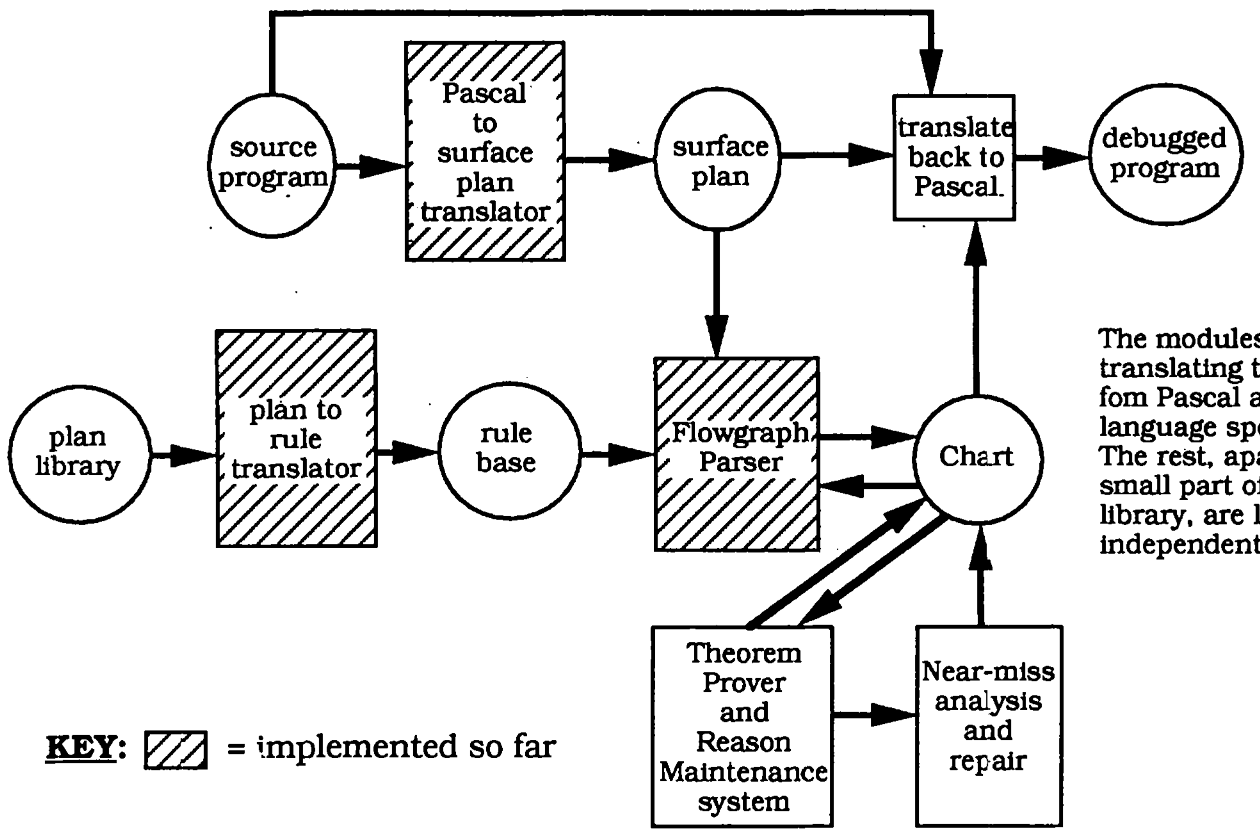
The overall structure of IDS as finally envisaged is shown in Figure 6.1. As can be seen there is much still remaining to be implemented. Of the missing modules the reason maintenance and theorem proving/symbolic evaluation modules are perhaps the most important since these are language independent parts of the system. Additionally, the translator from surface plans back to Pascal also remains to be written. However, enough has been implemented to show the feasibility of the plan calculus approach to program understanding using chart parsing as the basic recognition technique, and to allow very detailed descriptions of how debugging would work if the other modules were implemented. The strategy IDS will use to debug programs is the following:

(a) Translate the program into its surface plan.

(b) Try to understand the program by recognising all occurrences of library plans. Make a note of any "near" matches.

(c) Symbolically evaluate any remaining (i.e. unrecognised) parts of the surface plan.

(d) Check for broken preconditions of any of the recognised plans.


source program
Pascal to surface plan translator
surface plan
translate back to Pascal.
debugged program
plan library
plan to rule translator
rule base
Flowgraph Parser
Chart
Theorem Prover and Reason Maintenance system
Near-miss analysis and repair
KEY: ▨ = implemented so far

The modules for translating to and fom Pascal are language specific. The rest, apart from a small part of the plan library, are language independent

Figure 6.1
System Overview

(e) Use near match information and broken precondition information to try and repair the program.

(f) Translate the debugged surface plan back into the source language.

The rest of this chapter will give an account of the translation module, and Chapter 7 will describe the recogniser in full detail, and describe the translation process from Rich's plan notation to graph-like rules suitable for use by the recogniser.

6.2 The Translator

One of the main parts of the work described here has been the implementation of a Pascal-to-Surface-Plan translator. This translator can cope with a large subset of Pascal, and the structure is such that it can easily be extended to cope with most of the rest. As implemented so far it can cope with assignments, procedures and functions, numbers, records and pointers, arrays, if statements, and while statements. Because Pascal is quite a large language with a complex syntax (at least compared to LISP) the translation program is quite large, but follows the pattern of a recursive descent parser for Pascal.

6.2.1 Recursive Descent Data Flow Analysis

We can think of the process of translating a Pascal program into its surface plan as analogous to the process performed by a compiler which translates programs into machine code, the difference being that in this case the target language is that of surface plans. Accordingly, we perform the translation in much the same fashion as a compiler, interleaving the parsing and the surface plan generation. Since the grammar for Pascal can be expressed in an LL(1) form it can be parsed in a straightforward recursive descent fashion. Just like a compiler, the translator makes use of a symbol table, which, since Pascal is a lexically scoped language, can have a stack-like structure. At any given point in the parsing process the symbol table contains entries for each name that has been encountered (declared) so far in the program, and which is not yet out of scope. The entry for each variable contains the following information:

1) its name

2) its lexical level

3) which procedure it was declared in

4) which tie-point in the surface plan being generated represents the value of the variable

5) its kind (e.g. simple variable, or, if a formal parameter whether it is call-by-value or call-by-reference, is it a record accessing function, and so on)

6) its type (e.g. integer, real, boolean, pointer to a record type, procedure etc.)

7) definition information - this is an entry which can take on any of the values "defined", "undefined", or "possibly undefined", and enables the translator either to issue warnings to the programmer about variables that are or may be undefined when they are used, or to pass on to the recogniser as an indication of actual or possible bugs.

If the type is procedure, the entry for the procedure will also contain the following information:

8) the indices (in the symbol table) of the non-local variables used by the procedure, and the indices (in the symbol table) of the non-local variables updated by the procedure.

It is item 4) in this list which essentially constitutes the difference between the translator and a compiler. It corresponds to the symbol table entry where a compiler would hold the address of the variable. In a compiler the address of a variable is fixed for the duration of the translation process. However, in the translator this entry should be thought of as the "current address" i.e. the current tie-point in the graph representing that variable, and this will vary as the translation proceeds. The symbol table thus provides a mapping between names and tie-points, which we can think of as constituting an environment in which the translation takes place. So for example if we are translating a statement of the form:

x:=y+z;

the translator can generate an **@Pbinfunction** node[1] in the surface plan, with input .op coming from a tie-point representing "+", and input .input1 coming from the tie point (obtained from the symbol table) corresponding to y, and input .input2, coming from whatever tie-point represents z. A new tie-point is generated to represent the output from this node, and this is then stored in the symbol table against **x**. This changes the environment, and the subsequent statement is analysed in this new environment. We will refer to a mapping between names and tie-points as a data flow environment. At any particular time

[1] In the translation process we generate **@Pbinfunction** (apply Pascal binfunction) rather than **@binfunction** since the Pascal operators have different (more specialised) preconditions than the more general ones in the plan calculus. For example, addition in Pascal is only correct if the result is between -**maxint** and **maxint**. This can be a source of errors, and so representing the operators differently enables us to capture this. Of course we use overlays to map from the Pascal operators to the more general ones. This same comment applies to all the operators generated by the translator.

the mapping held in the symbol table will be referred to as the current data flow environment (CDFE). There will also be a global variable representing the current controlling condition (CCFE).

Tie-points representing known constants (e.g. "+", "*", "^", "true", "false", nil, "0" etc.) are held in a separate table (actually a hash table), so that when the translator needs to connect an operation to one of these it can simply look in the table to find which tie-point represents that value. This table is initialised at the start of the translation process, and if new constants (e.g. 99999) are encountered a new tie point is created to represent this value and stored in the known constants table.

Each ordinary (i.e. not a test or join) operation (NAPE) generated by the translator is represented as a list of the form:

[NAPE-type [NAPE-inputs][NAPE-outputs] Control-flow information]

and for ordinary operations the control flow information is simply the controlling condition for the NAPE.

Tests are represented in the form:

[test-type [test inputs] [C_{in} $C_{succeed}$ C_{fail}]]

where C_{in}, $C_{succeed}$, and C_{fail} are the input, succeed, and fail controlling conditions for the test.

Joins are represented in the form:

[join-output [succeed-input fail-input] [join-output] [$J_{succeed}$ J_{fail} J_{out}]]

where $J_{succeed}$, J_{fail}, and J_{out}, are the succeed, fail, and output controlling conditions for the join.

The graphs being generated are all held in a frame-like structure, with a frame for each sub-graph being generated. A sub-graph is generated for each conditional, while loop, and procedure, and the graph for the main program consists of any actions it actually contains directly, and a NAPE representing each subgraph it may contain. Similarly, the sub-graphs, may contain NAPEs representing other subgraphs. In the description that follows the sub-graph currently being generated will be referred to as the current segment.

The translation process will now be described. Rather than going through this syntax statement by syntax statement, we will only describe enough of main syntax forms we can currently deal with to make the technique clear. The other syntax forms that we can currently deal with are dealt with in a very similar fashion to those we do describe. In what follows we will use, as part of the meta-language for describing Pascal,angle brackets (< and >)to denote syntax elements, and curly brackets ({ and }) do denote zero or more repetitions of whatever they enclose, and we use a vertical bar (|) to denote a choice of forms. We will also use the brackets ([and]) to indicate an optional part of the syntax, and will use double quotes (" and") to indicate actual terminal items that are expected in the program at that point.

6.2.1.1 Declarations

Obviously, declarations add no nodes to the graph of the current segment. What they do is add new entries to the CDFE (symbol table). Variable declarations simply add a new entry for the variable being declared, with all the usual information a compiler would have, and with the definition information part of the entry being set to "undefined". The current translator does not handle many of the possible Pascal type declarations, other than the various number types, and records and arrays. If a variable is declared as being an array, then an entry is added to the symbol table for this variable. It is flagged as "defined". This is because, except for the case where an array is used before *any* assignments to *any* of its fields, it is in general very difficult to tell without doing a lot of sophisticated reasoning whether or not a particular entry in the array is undefined or not. This sort of task is best left to the plan recognition/theorem proving parts of IDS rather than burdening the translator with this task. The declaration of a new record type leads to new symbols being added to the symbol table - one for each of the record fields. So a declaration like that discussed in earlier chapters:

```
type listelement = record
                          numb : integer;
                          next : ^listelement;
                      end;
        plist = ^listelement;
```

leads to entries being made in the symbol table for numb and next. These are flagged as being "defined". Note that declaring a variable as being a record type also leads to a "defined" variable, but that declaration of a variable as being a pointer type leads to an "undefined" variable. This is because, in the case of the variable being a record, we are thinking of it as an immutable record cell (as discussed in Chapter 3) and hence the declaration actually defines it. However, a pointer variable does not become defined until it is given a value.

6.2.1.2 Translating Expressions and Assignment Statements

We will discuss this by considering the following fragment of Pascal syntax:

```
<assignment-statement> ::== <variable access> ":=" <expression>;
<variable access> ::== <entire variable> | <component variable>
                                         | <identified variable>
<expression> ::== [<sign>] <term> {<adding-operator> <term>}
<sign> ::== "+" | "-"
<adding-operator> ::== "+" | "or" |"-"
<term> ::== <factor> {<multiplying operator> <factor> }
<multiplying-operator> ::== "*" | "/" | "div" | "mod" | "and"
<factor> ::== <variable access>|<unsigned-constant>
                                | "(" <expression> ")"
```

The <component-variable> and <identified-variable> forms will be covered by the discussion of records and arrays later in this chapter. So let us consider a statement of the form:

```
y:= x+(y+z)*3
```

As in a normal recursive descent compiler, there is a procedure for dealing with each of the syntax forms of the language. However, the procedures for <expression>, <term>, and <factor> all return a result, which is the tie-point in the graph being generated which represents the result of the <expression>, <term>, or <factor>. So, the translator calls the procedure for <assignment statement>, and makes a note of which variable (y) occurs on the left of the assignment. It then calls the procedure for dealing with an <expressions>. Since this must be of the form:

```
[<sign>] <term> {<adding-operator> <term>}
```

it first makes a note of any <sign> if there is one, and proceeds to call the <term> procedure, which generates the graph for whatever <term> is actually present in the <expression> being analysed, and returns the output tie-point representing the result of the sub-expression corresponding to the <term>. Let us call this tie-point t_{sofar}. If there was a "-" <sign> then an **@Pfunction** node is generated, taking as its .op input the tie-point representing the unary minus operation (obtained from the known constants table), and taking as its .input input the tie-point t_{sofar}. A new tie point is generated to represent the output of this **@Pfunction**, and t_{sofar} is now set to this tie-point.

Now, if there is an <adding-operator> present, then the <term> procedure is called again to return its result (which will be referred to as t_{next}), the result of the next sub-expression. Now an **@Pbinfunction** node is generated in the graph with its .op input being the tie point corresponding to whatever the actual <adding-operator> was, and its two other inputs being t_{sofar} and t_{next}. A new tie-point is again generated to represent the output of this node, and t_{sofar} is set to this. In this fashion we eventually arrive at a situation where we have generated the graph for the <expression>, and have a tie-point t_{sofar} representing the result of executing the graph corresponding to the <expression>. This is then returned back to the calling procedure (in this case for recognising an <assignment statement>) as the result of the <expression>. The <assignment statement> procedure then stores this tie-point in the symbol table against whatever the variable on the left was, in place of its old value. So in this case the entry for y in the symbol table is changed to t_{sofar}.

The <term> procedure works in a very similar fashion to the <expression> procedure, except that it generates **@Pbinfunction** nodes with "multiplication" operators rather than "addition".

Constants and variables in expressions are dealt with by the <factor> procedure. If the <factor> procedure finds a variable when it is called, it looks up the symbol table and returns as a result whatever tie-point currently represents the value of that variable, unless the definition information for that variable is "undefined" or "possibly undefined". In this case it generates a warning message, and returns a tie-point corresponding to an undefined value (from the known constants table). It now flags the variable as "defined" to avoid generating multiple warnings. If the <factor> procedure has been called by the procedure for translating conditional statements (or inside part of the the procedure for translating loops which deals with the body of the loop) then the warning is "possible use of undefined variable", since this code may or may not be executed depending on the result of the test at the time the program is executed. If it is not inside a conditional, then the warning is "use of undefined variable".

If the <factor> procedure finds a constant, it then looks up the known constants table to see if this constant already has a tie-point associated with it. If so it return this one, otherwise it generates a new tie-point, stores it in the known constants table against the constant, and returns this new tie-point as its result.

If the <factor> procedure finds an embedded sub-expression (in brackets) then it returns the result of calling the <expression> procedure recursively.

Also note that a statement like this cannot alter the controlling conditions in any way. So all the generated nodes (NAPEs) get the current controlling condition as their controlling condition.

So suppose that, when the translator start dealing with the above statement, the symbol table contains the following entries:

y .... tie-point t1
x .... tie-point t2
z .... tie-point t3

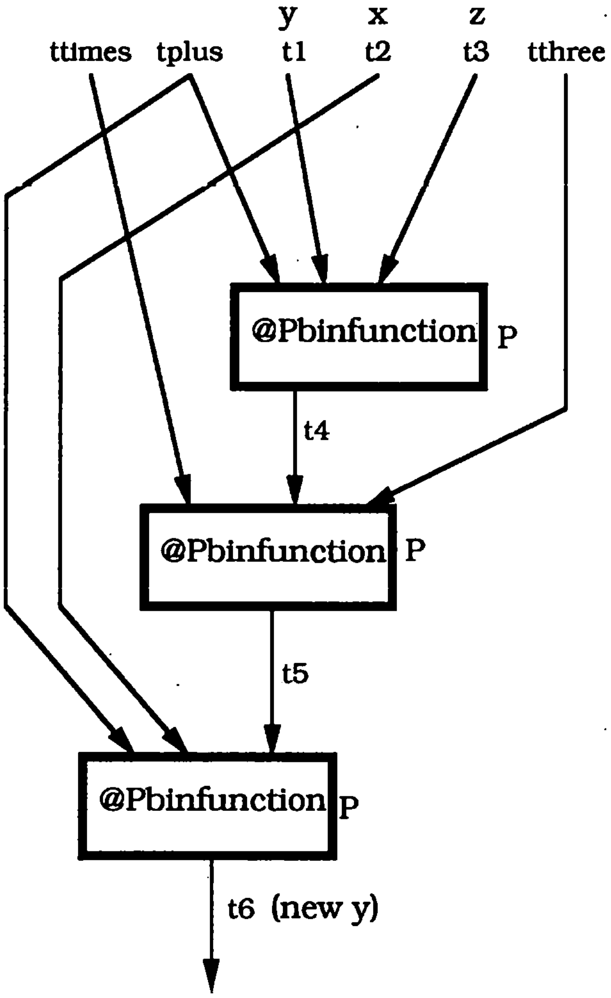


Figure 6.2
Graph For y:=x+(y+z)*3

and that the current controlling condition is P. Then the NAPEs added to the graph for the current segment by this expression are represented by:

[@Pbinfunction [t_{plus} t2 t5] [t6] P]
[@Pbinfunction [t_{times} t4 t_{three}] [t5] P]
[@Pbinfunction [t_{plus} t1 t3][t4] P]

where t4, t5, and t6 are new tie-points, t_{times} and t_{plus} are the tie-points representing the "+" and "*" operators, and t_{three} represents the value "3" and may or may not be a new tie-point depending on whether or not the constant 3 has appeared in the program prior to this statement. The graph corresponding to this is shown as Figure 6.2..

After this graph has been generated the symbol table contains the following entries:

y .... tie-point t6
x .... tie-point t2
z .... tie- point t3

6.2.1.3 Translating Conditional Statements

Conditional statements are more interesting. In the most general case they have the following form:

if <condition> **then** <statement1> **else** <statement2>

where <condition> is an <expression> which evaluates to a boolean value, and <statement1> and <statement2> are any valid Pascal statements. This is translated as follows. First the <condition> is translated, just by calling the procedure for expressions. A test node is then generated which takes as input the (boolean) output from the part of the surface plan corresponding to <condition>. C_{in} for the test is the current controlling condition $C_{current}$. A new symbol representing the test is generated (X say) and new controlling conditions $C_{current}\wedge X$ and $C_{current}\wedge\neg X$ are created and assigned to $C_{succeed}$ and C_{fail} for the test. A copy of the current data flow environment (CDFE) is then saved (i.e. the symbol table is saved) in $CDFE_{saved}$, as is the current controlling condition $(CCFE_{saved})$. $C_{succeed}$ is then assigned to CCFE, and <statement1> is then processed. At the end of this process CDFE is saved again $(CDFE_{succeed})$. CDFE is then restored from the old saved copy (i.e. CDFE is set to $CDFE_{saved}$), and CCFE is set to C_{fail}. Now <statement2> is processed. At the end of this process, join-output nodes are created, with J_{out} set to C_{in}, $J_{succeed}$ set to $C_{succeed}$, and J_{fail} set to C_{fail}. To determine the succeed data flows and the fail data flows connecting to the join-outputs, the CDFE (which is now actually the fail data flow environment) and the saved succeed data flow environment $CDFE_{succeed}$ are compared with the original saved data flow environment $CDFE_{saved}$. Any entry which is different in either CDFE(fail) or $CDFE_{succeed}$ from the entry in $CDFE_{saved}$ has a join-output NAPE created for it. The succeed-input of the join is the value in CDFEsucceed, and the fail-input is the corresponding value in CDFE. A new tie-point is then created for the output from the join-output, and this is used to update appropriate entry in the $CDFE_{saved}$. If one of either the succeed input or the fail input to the join is undefined then the new tie-point is flagged as "possibly undefined" (by storing this information in the symbol table against the appropriate variable. After this has been done for all entries in the symbol table which have changed on one or other or both branches of the conditional, the CDFE is set equal to $CDFE_{saved}$. The current controlling condition is then reset to what it was on entry to this procedure i.e. C_{in}.

So, consider the following piece of code:

```
if a<3 and b<2 then
    begin
        y:=a+b;
        z:=false;
        c:=a
    end
else
    begin
        y:=a-b;
        z:=true;
        d:=b
    end;
```

and suppose that the symbol table contains the following entries when the statement is analysed:

a ... t1

b ... t2

c ... t3

d ... t4

y ... t5

z ... t_{true}

When the <condition> is processed we get the following:

[[@Pbinfunction [t_{and} t7 t8][t9] P]
  [@Pbinrel [t_{less} t2 t_{two}][t8] P]
  [@Pbinrel [t_{less} t1 t_{three}][t7] P]
]

where as usual P is the current controlling condition, t7, t8, and t9 are new tie-points, and t_{two} and t_{three} represent the obvious constants.

Now the test node is generated, with two new controlling conditions:

[Ptest [t9][P $P \wedge X$ $P \wedge \neg X$]

The symbol table is now saved, the current controlling condition set to P∧X, and the statements in the then part of the conditional translated. This gives rise to:

[@Pbinfunction [tplus [t1 t2][t10] P∧X]

and the symbol table (CDFE$_{succeed}$) is now:

a ... t1

b ... t2

c ... t1

d ... t4

y ... t10

z ... t$_{false}$

CDFE is then restored from CDFE$_{saved}$, the current controlling condition is set to P∧¬X, and the else branch is then processed. This gives us:

[@Pbinfunction [t$_{minus}$ t1 t2][t11] P∧¬X]

and the symbol table (CDFE) is now:

a ... t1

b ... t2

c ... t3

d ... t2

y ... t11

z ... t$_{true}$

CDFE$_{succeed}$ and CDFE are now compared with CDFE$_{saved}$. It can be seen that on the then side variables c, y, and z have altered, while on the else side variables y,z, and d have altered. A join is therefore created for each variable in the *union* of these two sets of variables, giving us:

[[join-output [t1 t3][t12] $P \wedge X$ $P \wedge \neg X$ P]
[join-output [t10 t11][t13] $P \wedge X$ $P \wedge \neg X$ P]
[join-output [t_{false} t_{true}][t14] $P \wedge X$ $P \wedge \neg X$ P]
[join-output [t4 t2][t15] $P \wedge X$ $P \wedge \neg X$ P]

The graph for the whole conditional statement is shown in Figure 6.3. Note that we have shown all the joins as a single join with several succeed and fail inputs rather than as several joins with a single succeed and a single fail input each. In the segment containing the conditional a NAPE is added representing this conditional, with inputs corresponding to the variables whose values were used inside the conditional statement before they were updated (if they were updated at all), and outputs corresponding to the outputs from the join, and any variables that were altered inside the <condition> part of the statement. How can we tell which variables are used in a segment? When translating anything, it is the <factor> procedure mentioned earlier which is actually responsible for dealing with variable (or constant) usage. If it is dealing with either a constant, or a variable, then it follows that this variable or constant must be used by the current segment. However, it does not follow that because a variable is used inside a segment that its value must be an input to the current segment, since the segment might update a variable before using its value. So the translator maintains two lists for the current segment. One is the list $V_{updated}$, of all variables which are updated in the current segment. This list is kept up to date by the <assignment statement> translator (and the procedure responsible for adding a node representing another segment to the current segment). The other list V_{used} is the list of all variables which are used before being updated. So when the <factor> procedure is dealing with a variable it checks to see if that variable has been updated in this segment. If not,

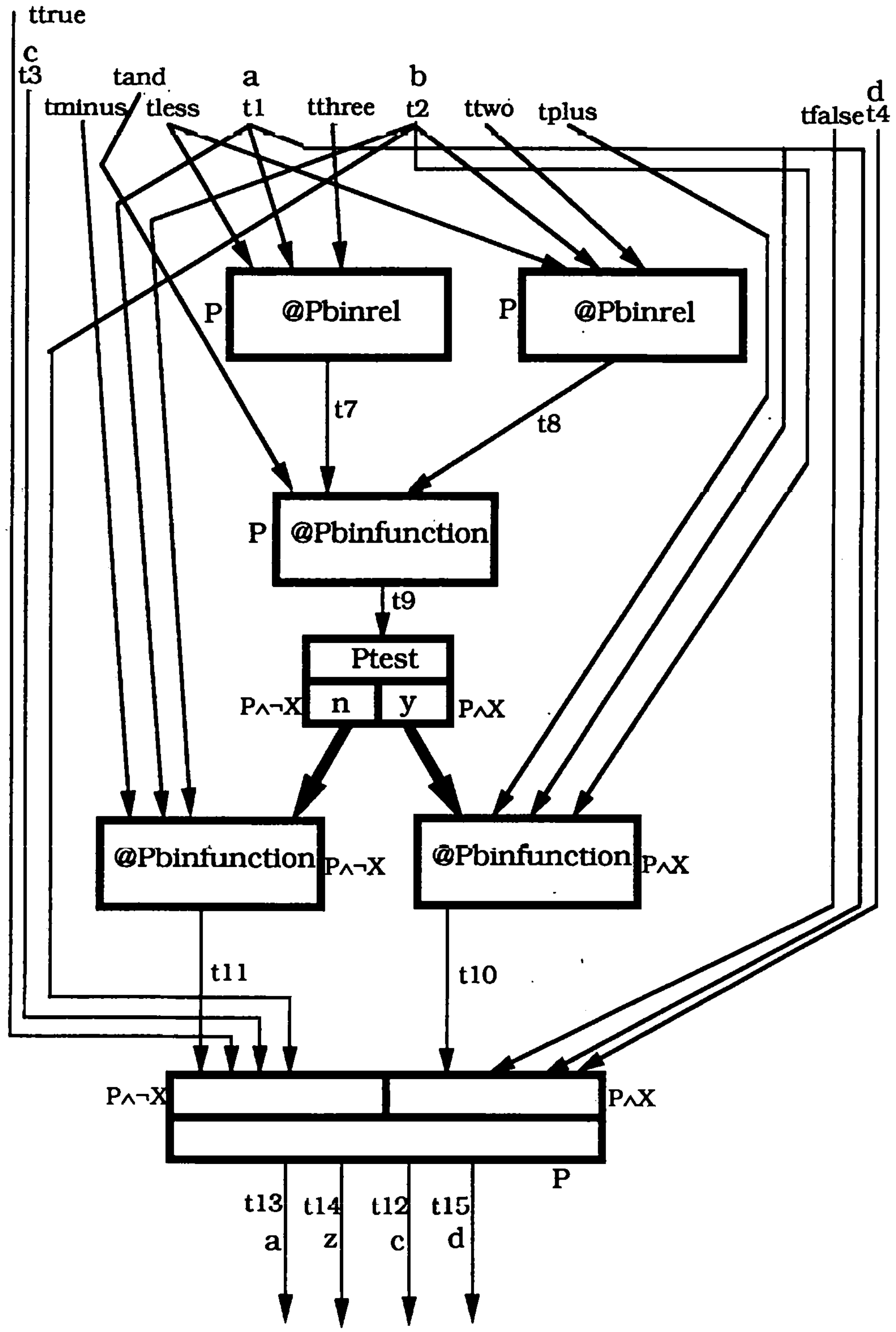


Figure 6.3
Graph for Conditional

then this use of the variable must represent a value flowing into the current segment from outside. In this case, provided the variable is not already present in V_{used} it is added to it. If constants are used they are simply added to V_{used}. It is this list V_{used} which is used by the translator for conditionals and loops to work out the inputs to the segment. In the case of the conditional just discussed, V_{used} contains:

[a b less false plus minus true]

so in its enclosing segment the conditional is represented by:

[conditional1 [t1 t2 t_{less} t_{false} t_{plus} t_{minus} t_{true}][t12 t13 t14 t15] P]

Of course when this NAPE is added to the graph of its enclosing segment, the CDFE of that segment must be updated so that the variables in V_{used} now have the output tie-points of the loop or conditional associated with them.

6.2.1.4 Translating Loops

Loops are by far the hardest case to deal with. They are dealt with in a similar fashion to conditionals, although the actual processing is more involved. We will illustrate it with reference to a while loop of the form:

while <condition> **do** <statement>;

Other loops (**for** loops, and **repeat** ... **until** ... loops) are dealt with similarly. First of all note that a loop can be thought of as a conditional, with an empty **else** part, and with a recursive role representing the whole loop again in the **then** part. This recursive role comes after the graph for the body of the loop. Note too that, in a while loop, the <condition> is part of the loop since it is re-evaluated each time round the loop. So, before translating <condition> a copy of CDFE is saved (the pre-condition data flow environment $\mathrm{CDFE}_{\mathrm{precondition}}$). The <condition> is then translated as before, and its output is connected to a test node, just as in the case of conditional statements. As before two new controlling conditions $P \wedge X$ (C_{succeed}) and $P \wedge \neg X$ (C_{fail}) are created. The current data flow environment is again saved by copying (the post-condition data flow environment $\mathrm{CDFE}_{\mathrm{postcondition}}$). The current controlling condition is then set to C_{succeed} and the <statement> is then translated. At this point a recursive role must be generated. To do this we need to know which variables pass their values into the loop so that we can pass the tie-points representing the values of these same variables after the body of the loop has been executed through to the recursive role. This is done by looking at which variables have either been simply used, or have been used before being updated, inside the loop. These constitute the external inputs to the loop. Assuming for the moment that we can identify these, then the tie-points of the corresponding variables in the current data flow environment are made the inputs to the recursive role. Now, to compute the outputs of the recursive role, we have to look at which variables are altered in the loop. These correspond to those entries in the current data flow environment which are different from entries in the *precondition* environment[2]. Since these variables are altered in the loop they must be outputs from the loop. Of course the recursive role must alter these same variables, and hence for each of these variables, a new tie-point is created as the corresponding output from the recursive role. For each of these variables the current data flow environment is updated with the corresponding new tie-point, giving us the correct data flow

[2] Note that we have to look at the precondition environment, since it is possible that evaluating the condition might alter the values of some variables (if say it involved a function call).

environment following the recursive role. We now have to generate join-output nodes, to reconnect diverging data values after the test node. Just as in the case of a conditional, we generate a join-output node for each variable that has changed in either the **then** part or the **else** part of the conditional. However, in this case the **else** part is empty, so the succeed data flow environment is simply that just computed (CDFE), and the fail data flow environment is simply the saved post condition environment $\text{CDFE}_{\text{postcondition}}$. So we only need to compare CDFE with $\text{CDFE}_{\text{postcondition}}$, and for each variable that now has a different tie-point associated with it we create a join-output node, with succeed input coming from CDFE, and fail input coming from $\text{CDFE}_{\text{postcondition}}$. For each of these a new tie-point is created to represent the output after the join, and the entries for the appropriate variables in CDFE are now updated with these new tie-points.

So consider the following piece of code:

```
sum:=0;
sumsquarcs:=0;
i:=1;
while i<10 do
    begin
        sum:=sum+i;
        sumsquares:=sumsquares+i*i;
        i:=i+1;
    end;
```

Suppose that the current controlling condition is P. Then on entry to the loop the symbol table contains the following entries:

i ....t_{one}

sum..........................t_{zero}

sumsquares...........t_{zero}

This is saved as $\text{CDFE}_{\text{precondition}}$. Then the condition is processed. This adds the following NAPE to the graph for the loop:

[@Pbinrel [t_{less} t_{one} t_{ten}][t1] P]

$\text{CDFE}_{\text{postcondition}}$ is then saved (it happens to be the same as $\text{CDFE}_{\text{precondition}}$). A test NAPE is then generated (with appropriate succeed and fail controlling conditions):

[Ptest [t1] P $P \wedge X$ $P \wedge \neg X$]

The body of the loop is then processed, adding the following NAPEs:

[@Pbinfunction [t_{plus} t_{zero} t_{one}][t2] $P \wedge X$]

[@Pbinfunction [t_{times} t_{one} t_{one}][t3] $P \wedge X$]

[@Pbinfunction [t_{plus} t_{zero} t3][t4] $P \wedge X$]

[@Pbinfunction [t_{plus} t_{one} t_{one}][t5] $P \wedge X$]

where t2, t3, t4, and t5 are all new tie-points. The symbol table (CDFE) is now in the state:

i	t5
sum	t2
sumsquares	t4

The V_{used} list for the loop segment is now:

V_{used}= [less plus sum i sumsquares times 1 10]

The recursive role is now generated. Its inputs are the tie points representing the objects in V_{used}. Its outputs are new tie-points for all the variables which have changed since entering the loop. These are i, sum, and sumsquares since these all have different values in CDFEprecondition and CDFE. So we generate a recursive role:

[recursive [t_{less} t_{plus} t2 t5 t4 t_{times} t_{one} t_{ten}][t6 t7 t8] P∧X]

and the symbol table is updated with the new tie-points giving us CDFE:

i t6
sum...........................t7
sumsquares...........t8

A join outputs-node is now generated for each of these variables, with succeed input coming from this environment, and fail input coming from $\mathrm{CDFE}_{postcondition}$, and a new tie-point for each join's output. This adds the following NAPEs:

[join-output [t6 t_{one}][t9] P∧X P∧¬X P]
[join-output [t7 t_{zero}][t10] P∧X P∧¬X P]
[join-output [t8 t_{zero}][t11] P∧X P∧¬X P]

This is shown in Figure 6.4. In the segment containing the loop it is represented as a single NAPE with inputs given by the variables in V_{used}, and outputs given by the outputs of the join(s). i.e.

[whileloop1 [t_{less} t_{plus} t_{zero} t_{one} t_{zero} t_{times} t_{one} t_{ten}][t9 t10 t11] P]

Again, when this is added to the graph of the segment containing the loop. its CDFE must be updated appropriately.

6.2.1.5 Translating Procedures and Procedure Calls

Subject to the limitations discussed below, procedure definitions are handled as if they were a complete program. At the start of the

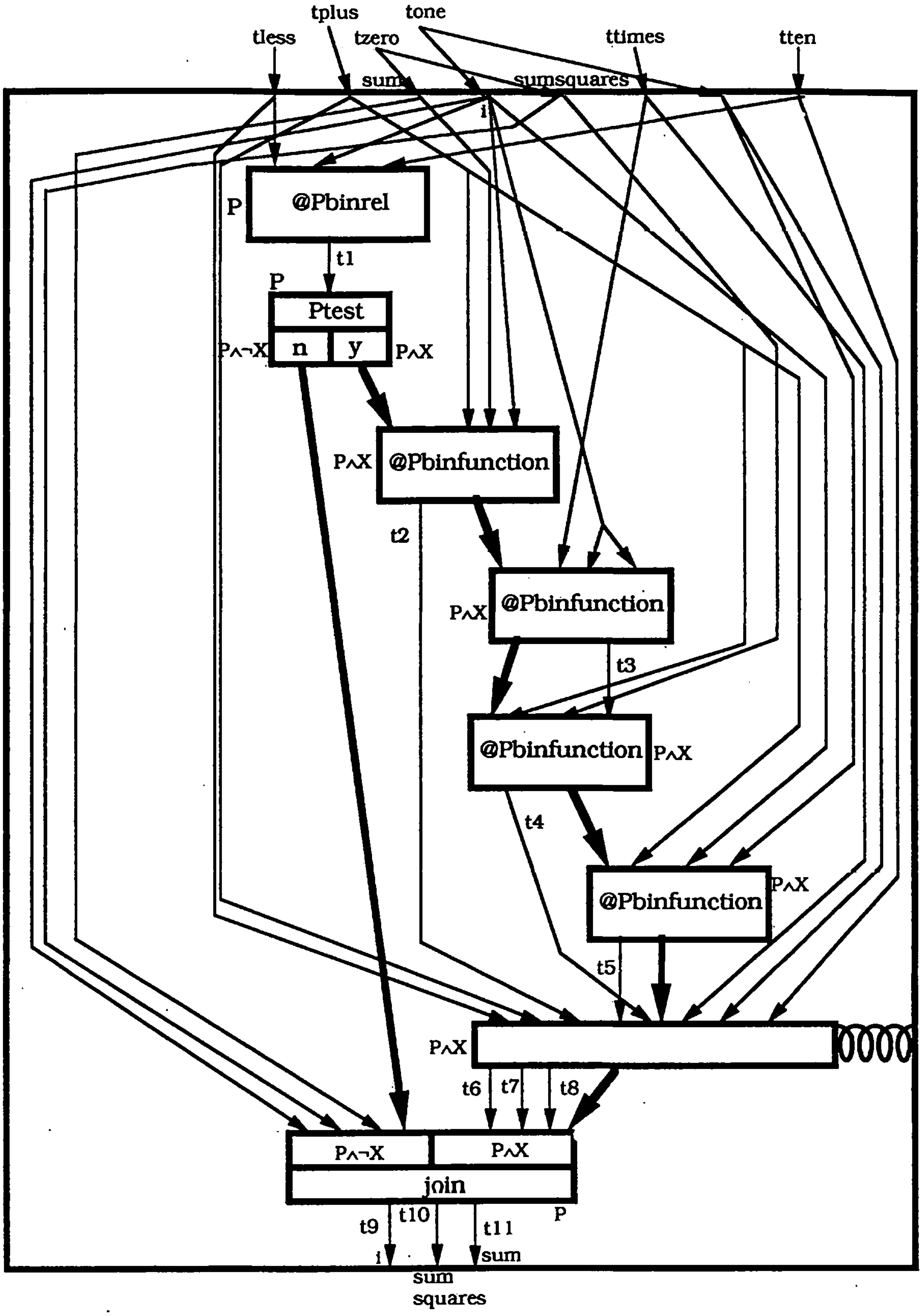


Figure 6.4
Graph For Loop

procedure definition new control and data flow environments are created. These have all the variables etc. that are currently in scope, but have new "undefined non-local" tie-points associated with them. This is to avoid generating error messages when the procedure is being analysed, since of course we can only tell at the place where a procedure is called whether or not its non-local variables are undefined. Formal parameters are treated exactly as a normal recursive descent compiler would, but have "unassigned formal" tie-points associated with them. Analysis of the procedure body etc then proceeds exactly as for a main program, producing a surface plan for the procedure. This is stored in a frame-like structure, with slots for the procedure name, its surface plan, its formal parameter inputs, its non-local inputs and outputs (non-local variables altered by the procedure are treated as outputs). Call-by-reference is treated as if it were call-by-value-result, which is equivalent provided that in a call of the procedure there is no aliasing of a call-by-reference actual parameter with a non-local variable used by the procedure. So call by reference parameters are also treated as output variables.

When a procedure call is encountered a node representing a call of the procedure is inserted into the surface plan being generated. This node has inputs corresponding to <u>all</u> the inputs of the surface plan for the procedure, and these inputs are connected to the current surface plan in such a way that non-local inputs are the tie-points in the current dataflow environment corresponding to the variables used in the procedure. It is here that any errors to do with undefined variables are detected. A new tie-point is created for each output from the procedure, and the current data flow environment is then updated to reflect the fact that these variables have had their values updated.

6.2.2 Mutable Functions and Side Effects

As promised above, we will now discuss the way in which arrays and records are dealt with. This is closely related to the phenomena of mutable functions and side effects discussed earlier in Chapter 3. Now programs which operate by side-effect present great difficulties for data flow analysis, upon which the translation process described above depends. The only way in which side effects can occur is if we update structures of some sort. This corresponds to changing the roles of some data object. As stated earlier though, rather than viewing the structures themselves as having changed, we will view this as a change to the (now mutable) accessing functions for the data type involved. So, for instance, a statement like:

z:=x.numb (where x is a listelement, and z is an integer)

will be treated as if it were something rather like:

z:=apply_function(numb,x)

and

x.numb:=z (where x is a listelement, and z is an integer)

will be treated as if it were:

numb:=newarg(numb,x,z)

This approach has the effect that all information about side effects etc. is carried by the mutable access functions. This means that the only information that is relevant when we need to determine the behaviour of some object in a situation are the current values of the mutable access functions in that situation. For instance, the

listelementcell->list behaviour function, which formerly was defined by:

α=**listelementcell->list**(p,s) ≡
[[instance(list, α)
∧ hcad(α)=numb(**listelementcell->listelement**(p,s))
∧ tail(α)=^**listelementcell->list**(next(
listelementcell->listelement(p,s)),s)]]

can now be rewritten as:

α=**listelementcell->list**(p,s) ≡
[[instance(list, α)
∧ head(α)=apply(**function**(NUMB,s),p)
∧ tail(α)=^**listelementcell->list**(apply(**function**(NEXT,s),p),s)]]

and the dependence of the behaviour function **listelementcell->list** on the mutable functions NUMB and NEXT is now made explicit. This means that the **listelementcell->list** behaviour of an object X at time s, depends on the (function) behaviour of NUMB and NEXT at time s, and the **listelementcell->list** behaviour of X at time t will only necessarily be the same if the (**function**) behaviour of NUMB and NEXT has not changed between s and t. Of course, additional reasoning will be required if NUMB and NEXT are changed between s and t in such a way as not to effect the **listelementcell->list** behaviour of X (e.g. by altering an altogether separate list implemented using listelement cells as the building blocks) in order to recognise that the behaviour of X is still the same. However, this is just a reflection of the fact that reasoning about side effects is difficult in general, and the underlying philosophy of the plan calculus, and of the graph parsing recognition process we will present later, is to try and capture as many possible common uses of side effects as plans, in order to try to alleviate the necessity for general reasoning. However, in order to make it easier for the plan recognising system, described in the next chapter, to recognise these standard (side-effecting) plans, it is important that the surface plan reflect as closely as possible the true data flows that actually happen in the program. This means that the mutable functions NUMB and NEXT in the above example must be treated as values which are passed around by the program, and the correct current values of these should be used in all the appropriate places. This can be achieved simply by treating numb and next as ordinary *variables*, with *values* NUMB and NEXT respectively.

6.2.2.1 Dealing with Records and Pointers

As described above, a record declaration is interpreted as defining a class of objects whose type is the record type. However, rather than regarding these objects as mutable objects which are accessed or updated by the field selector functions for the records of that type we choose to regard the objects as fixed, and the field selector functions as mutable. So a statement of the form:

x:=y.field

where y is a record and field is one of the fields of the record, is treated simply as if it were the application of function **field** to object y, and this is represented in the surface plan by an **@function** node, whose .op input is the tie-point in the CDFE representing **field**. Similarly,

x.field:=y

is treated as changing the definition of the function **field** so that on all objects other than x it has the same value as before, but on x it now has the value y. This is represented by an **#newarg** operation in the surface plan with .op being the tie-point for **field**, .arg being the tie-point for x, and .value being the tie-point for y. The CDFE is then updated with the

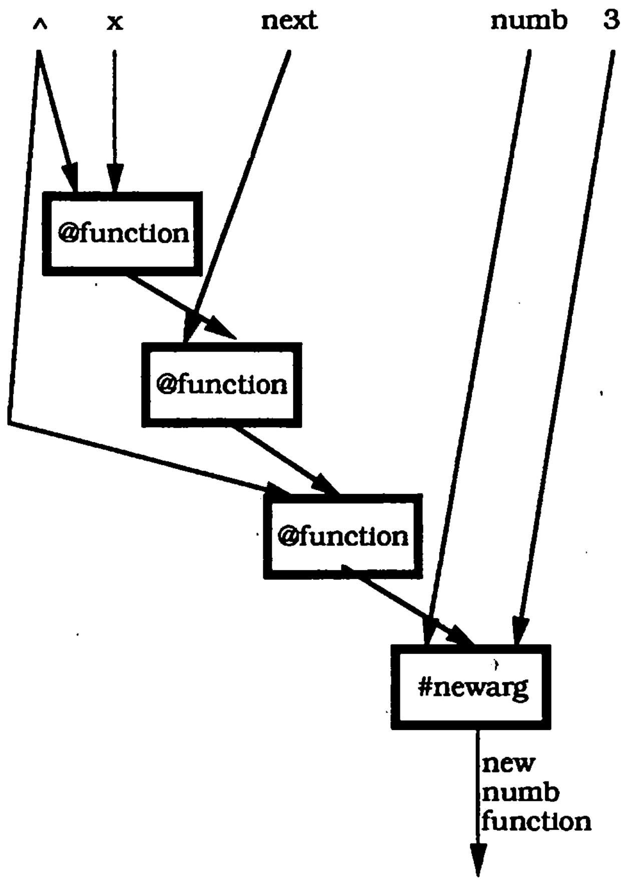


Figure 6.5
Graph For x^.next^.numb:=3

value of **field** now being represented by the (newly created) tie-point representing the output from the **#newarg** operation.

Pointers are treated similarly as being a type of object with "^" being treated as a function which maps pointers to objects. This approach enables the system to (say) recognise a list implemented as a chain of linked records as constituting a **thread**, with (field∘^)[3] constituting the successor function (via the overlay **composed-functions->function**), and the pointers themselves constituting the nodes of the **thread**. This approach means that a statement like:

x^.next^.numb := 3;

will be represented by a graph like that shown in Figure 6.5.

6.2.2.2 Dealing with Arrays

Although the approach we have just outlined works quite well for records (and cons-cells in Lisp), it does give us a problem with arrays. The problem essentially turns out to be one of *naming*. For instance, we can model one dimensional arrays as mutable sequences i.e. as a subtype of functions. Such an array A (say) is similar to a record in the sense that it can be viewed as a compound data object with fields, and so ought in principle be capable of being dealt with in a similar fashion. However, the reason why we can take the approach outlined above with records is that the access functions are named. This means that we can tell whenever one of them is used or altered - it is clear from the syntax of the programming language - and this means that the translation process from a program to a surface plan can always ensure that the tie-point representing the current value of one of these named

[3] ∘ is used to denote function composition. So (f∘g)(x)=f(g(x)).

access functions is passed through to actions when necessary. However, in the case of an array A we do not have these named access procedures. Even if we were to name them something like A_1, A_2, A_3, etc., where

$A_1(A)=A[1]$ and $A_2(A)=A[2]$ and $A_3(A)=A[3]$ etc.

we still would not be able to tell in general which one was being used (or updated) since programs often contain statements like:

```
A[2*i-j]:=3
```

and we would not know which of the access functions was actually being updated. In the case just mentioned, where we have a named array, the approach of treating the array as a mutable function works quite well, since we now have a name for the function being updated, and a statement like the above can be modelled as an **#newarg** operation on the mutable array A. This amounts to treating the whole array itself as analogous to a mutable field accessing function on records, where the domain of the mutable array function is now the set of immutable integers (or at least an appropriate subset of the integers). However, consider the following case:

```
var a : array[1..n] of ^array[1..m] of integer;
```

and consider a statement sequence like:

```
a[i]^[j]:=3;            (1)
.
.
.
x:=a[i]^[j]             (2)
```

From one point of view statement (1) leaves the array a unchanged, since a is an array of pointers and none of these have changed. This means that a[i]^ represents the same value in statements 1 and 2 (i.e.

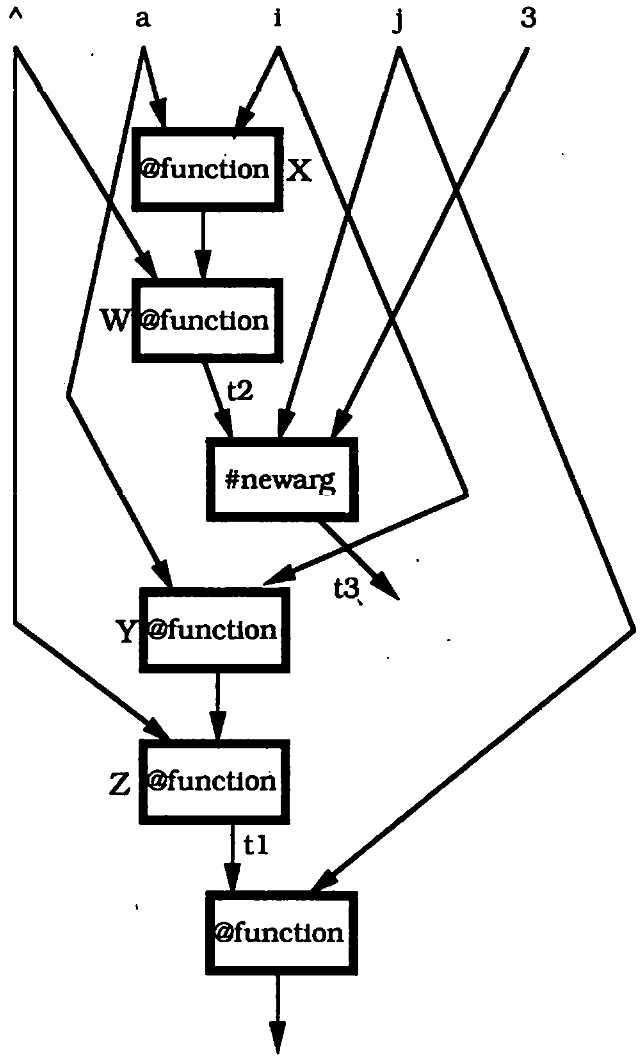


Figure 6.6
Graph for Arrays(pure mutable sequence approach)

the identity of the array involved is the same in both situations) since ^ is an immutable function. However, using this object in statement 2 involves using the **sequence**[4] behaviour of the array a[i]^ and this has changed. The problem can be made clearer if by considering Figure 6.6, showing the relevant parts of the surface plan for the above code. This diagram essentially reflects the difficulty that a translator has in determining the true data flows. Following the approach discussed earlier of treating arrays as sequences (i.e. functions) would lead to the plan diagram shown. This is highly misleading, since in the light of the earlier discussion of collapsing, we would naturally want to collapse the two **@function** operations labelled X and Y in the figure, and then collapse the two **@function** operations labelled W and Z. However this would lead to identifying tie-points t1 and t2, whereas really tie-points t1 and t3 should be identified. In fact the problem is worse than this. As can be seen from the diagram, t3 does not connect up in any way to the rest of the graph, and would thus appear to be a completely redundant computation, whereas it actually does have an effect. It should be noticed that the axiomatic formulation of the plan calculus does not have this problem. This is because the syntactic form of the axioms distinguishes between object identities and object behaviours, whereas the graph form only passes behaviours around. In the absence of side effects this is fine, since the objects are all then immutable, and we do not need to distinguish the behaviour of an object from its identity since the behaviour of the object is the same in all situations. If we were to treat tie-points as representing identities of objects rather than behaviours we would again get into trouble, since this would now lead us to identify tie-points t3 and t2 (and t1), giving rise to cycles in the graph, as well as forcing us to continually reason about the

[4] **sequence** is a subtype of **function**.

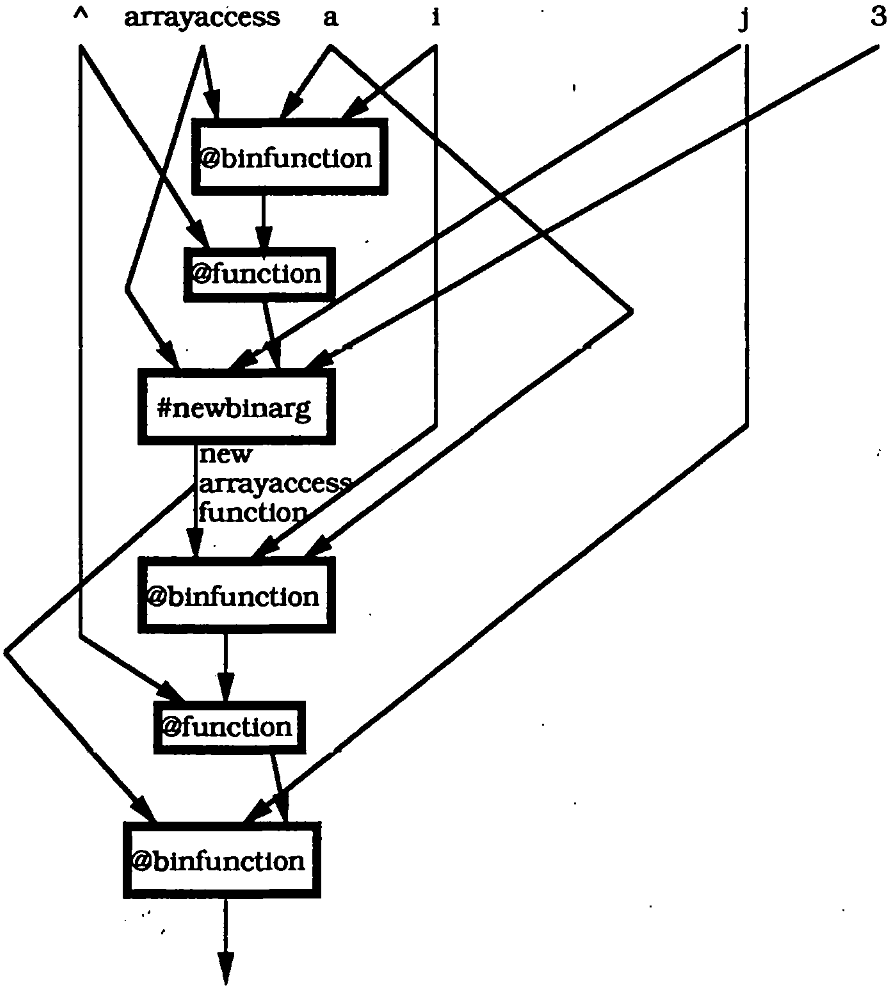


Figure 6.7
Graph For Arrays (pure arrayaccess approach)

situations in which we were considering the tie-points. This would make a graph recognition approach to program understanding very difficult.

The solution to this is to adopt an approach similar to that of the case of records. We can treat arrays as immutable objects (perhaps thought of as array 'cells'), with a mutable binary access function array-access defined on it i.e. an array access A[i] can be treated as if it were:

binapply(array-access,i,A)

and array updates can be modelled by an **#newbinarg**[5] operation. This results in Figure 6.7, which is better, but now we are no longer (from the graph alone) able to collapse anything. The best solution, and the one we will adopt, is a mixed strategy. Named arrays (i.e. those identified by a variable name in the program) will be treated as mutable functions as before. Unnamed arrays i.e. those which are components of data structures, will be treated as immutable, with the mutable function array-access defined on them. This gives rise to Figure 6.8, which collapses into Figure 6.9, which is probably the most intuitively correct representation of what is going on. It should be noted that if the plan recognition demands it we can always move from the point of

[5] **newbinarg** is not an operation defined in Rich[1981]. We have introduced it in order to model array updates. Its 'compact' definition is:

```
IOSpec newbinarg .old(binfunction) .arg1(object) .arg2(object) .input(object)
                                                        =>  .new(binfunction)
Preconditions instance(argtype-one(.old),arg1) ∧ instance(argtype-two(.old),arg2)
          ∧ instance(range-type(.old),.input)
Postconditions binapply(.new,.arg1,.arg2)=.input
                    ∧ ∀xyz[binapply(.old,x,y)=z  ∧ x≠.arg1 ∧  y≠arg2 ⊃ binapply(.new,x,y)=y]
                   ∧ argtype-one(.new)=argtype-one(.old)
                   ∧ argtype-two(.new)=argtype-two(.old)
                   ∧ binrange-type(.new)=binrange-type(.old)
```

#newbinarg is defined as a specialization of **newbinarg** in which .old=.new

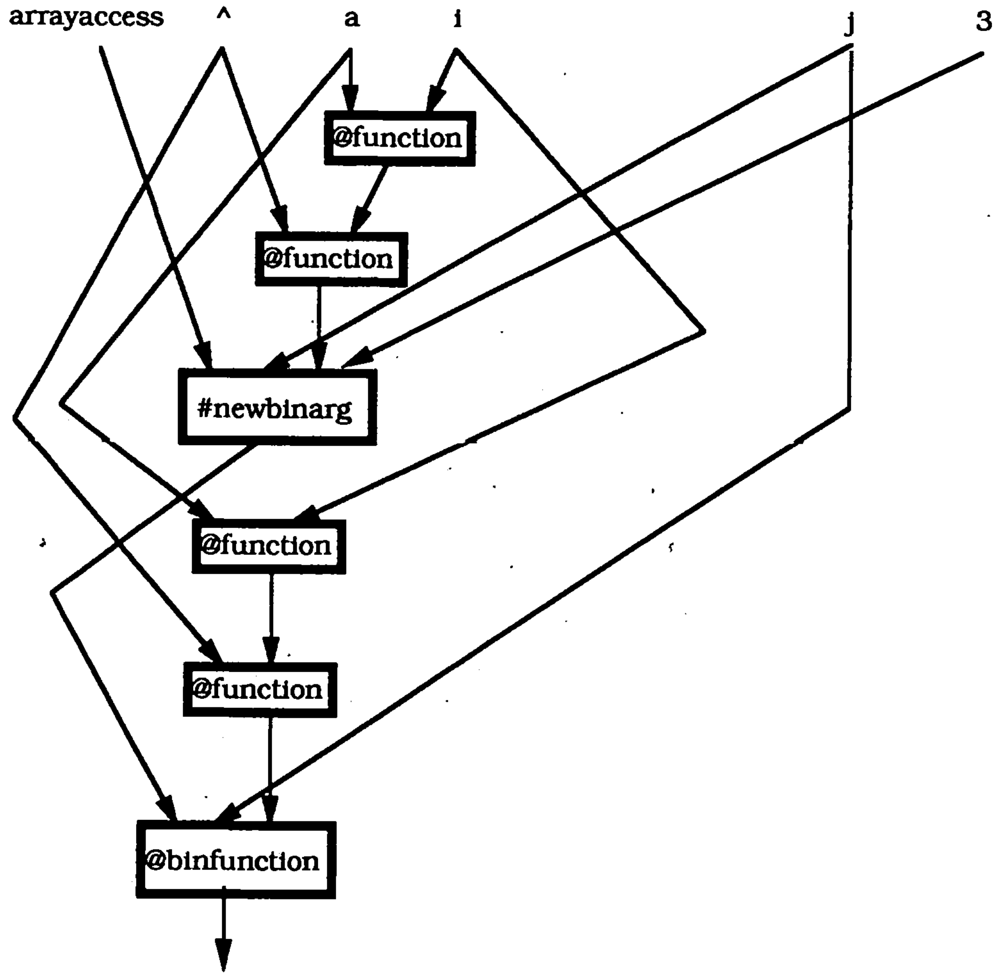


Figure 6.8
Graph For Arrays (Mixed Approach)

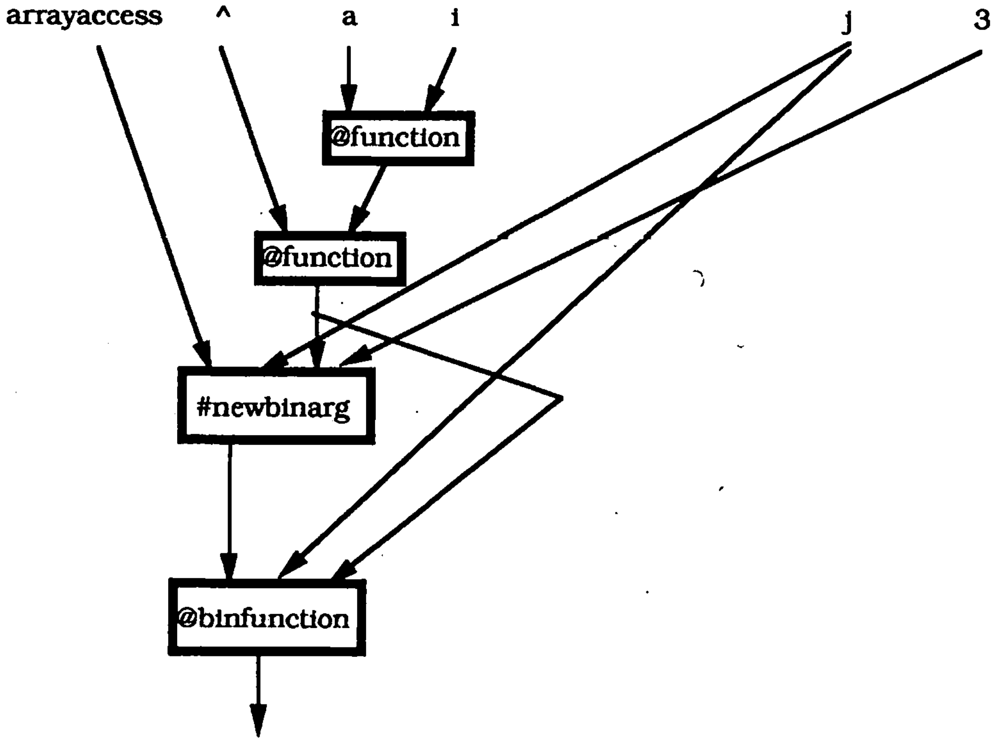


Figure 6.9
Collapsed Version of Figure 6.8

view we have when using array-access to access arrays, to one in which we have a mutable array via overlays like **@binfunction+two->@function** and **binfunction+two->function**. This amounts to currying array-access on its second argument (the array) to get a mutable function which behaves exactly like the array (treated as a mutable function).

6.3 Limitations of the Recursive Descent Approach

The recursive descent approach to translating Pascal programs into a surface plan cannot deal with all programs. In particular, it cannot deal with programs involving forwardly defined procedures, or recursive procedures. This is because, when we process a procedure call, in order to correctly join it to the surface plan being generated we need the following information about a procedure:

a) what non-local variables it accesses or updates

b) which of its formal parameters are call-by-reference rather than call-by-value.

This requires the procedure (or function) to have been previously analysed. It would not be hard to modify our translator to cope with these types of procedure. All that would be required is a pre-pass over the program collecting the above information according to the techniques outlined in Hecht [1977]. The current translator could then use this information when required.

6.4 General Limitations

There is however a more serious limitation of any data flow analysis technique. This is concerned with programs which have data structures which share parts or all of their substructure. We have already touched on this when discussing arrays and records above. The solution we described there at least has the property that it does not produce positively misleading data flows. However, without general reasoning, it is not possible for the data flow diagram representing the program to always reflect the true data flows, and hence this is another reason why a general purpose program understanding and debugging system must include a powerful reasoning system which is well-integrated with any plan recognition process. Consider the program fragment below:

```
a[6]:=z;
y:=a[3*i-j];
```

This would give rise to the graph shown in Figure 6.10. However, suppose that 3*i-j actually evaluates to 6. In this case the graph ought to be that shown in Figure 6.11. This amounts to being able to identify t1 and the tie-point representing 6, as well as t2 and the tie-point for z, in Figure 6.10.

This problem is really inherent in the whole surface plan approach to representing programs. Indeed it is quite likely to be inherent in any program analysis system that aims to cope with anything other than purely functional languages. At the moment the best we can hope for is to capture the most common usages of side effect in plans, and leave more general cases for when the plan recogniser has been supplemented by a plan calculus theorem prover.

Related to this problem is that of variable aliasing. If a procedure uses a non-local variable x, and a call of the procedure has x passed as the actual parameter corresponding to a call-by-reference formal argument, then the resulting true data flow diagram may be quite different to that obtained under the assumption that x and the formal parameter were different. In this case the procedure ought to be

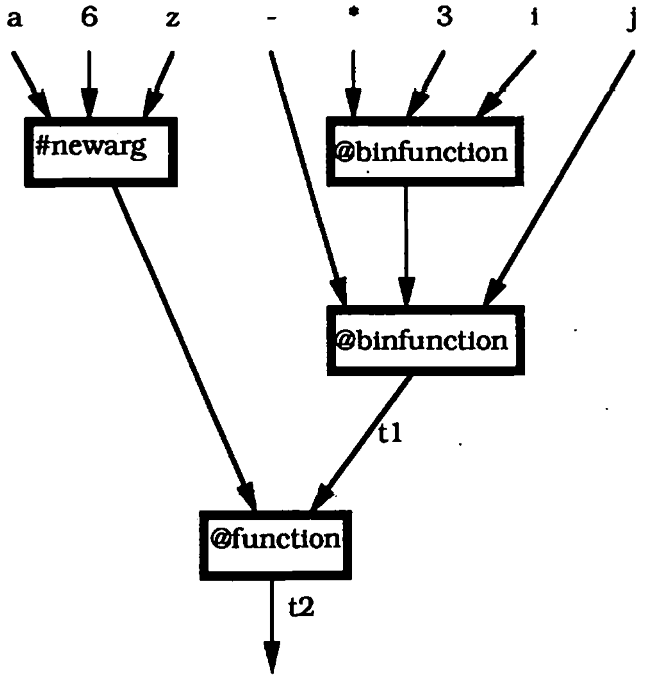


Figure 6.10
Graph For a[6]:=z; y:=a[3*i-j]

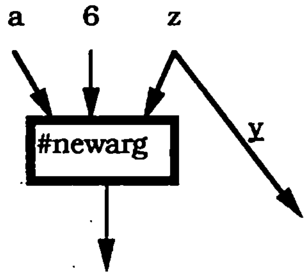


Figure 6.11
Graph For a[6]:=z; y:=a[3*i-j]
When 3*i-j=6

retranslated under this assumption to get a true description of this particular call. This has not been done, so our approach currently suffers from the limitation that it cannot cope with this particular type of call. In general this can be quite hard to recognise, and more work will need to be done on this. Note that this can happen even if there is no structure sharing of data structures going on.

Finally it should be noted that we have excluded from consideration in all of this the **goto** statement. This is because this introduces cycles into the surface plan, and everything we have done assumes the surface plan is not cyclic. In particular, if there are cycles it is much harder to assign an interpretation to tie-points as being behaviours of specific objects at specific times. To deal with **goto** statements, it would probably be better to restructure the program first using any of the available techniques for doing this, and then analyse the result.

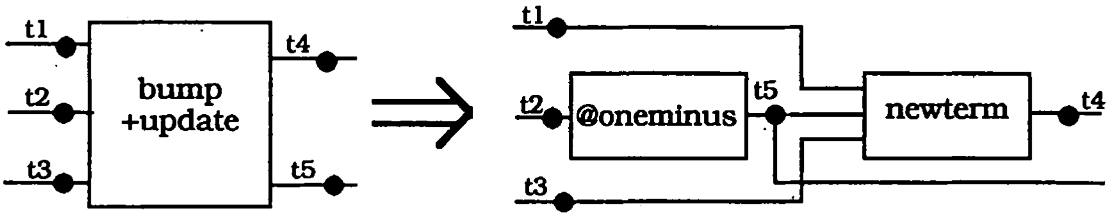
Chapter 7.

The Plan Recognition System And The Plan Library

The chart parser described in Chapter 5 clearly has many features making it suitable for program understanding within the plan calculus/plan diagram formalism. In particular we will treat the plan library as defining a set of rules for a restricted structure-sharing flowgraph grammar, where the set R of collapsible operations is precisely the set of deterministic operations as defined in Chapter 4. However, there are many features of this formalism which do not exactly fit the flowgraph formalism as described so far. This chapter will describe the various features of plan diagrams and the plan calculus which an unmodified flowgraph parser cannot cope with, and describe the changes necessary. As this project is currently only using the parser in a bottom-up mode, the changes described below have only actually been implemented for bottom-up parsing. However, it would not be difficult to do them for top-down parsing as well, and indeed if this system were to be incorporated into a tutoring system along the lines of PROUST [Johnson, 1986] this would be necessary.

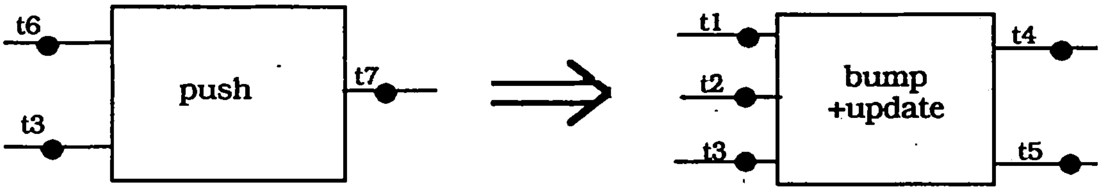
7.1 Dealing with Overlays and Data Abstraction.

As discussed in Chapter 3, the plan library not only contains temporal plans - it also contains temporal and data overlays. Additionally, it also contains data plans. Each programming cliché is encoded as a temporal plan. A typical example is **bump+update**, shown in Figure 7.1. This can be encoded very nicely in the flowgraph formalism, and recognised quite easily (ignoring constraints on the tie-points), using the flowgraph parser as described already. However it should be noted that the constraints actually imply two views of the inputs t1 and t2. These are:



where upper-segment(t1,t2) and
upper-segment(t4,t5)

Figure 7.1
Bump+update



where
t6=upper-segment->list(upper-segment(t1,t2)) and
t7=upper-segment->list(upper-segment(t4,t5))

Figure 7.2
Bump+update->push

(a) They are simply two data values providing inputs to the plan.

(b) They are the components of a compound data structure (a data plan in Rich's terminology) known as an **upper-segment**.

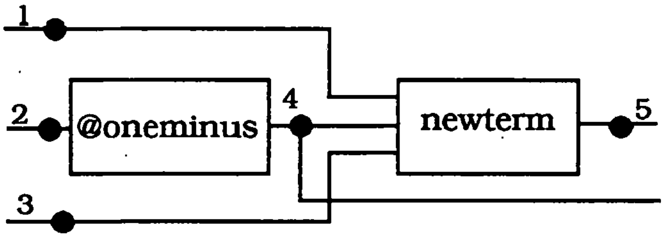
Similarly, there are two views of the outputs from the plan. Of course the constraints themselves can be checked just like any other constraints at the time the plan is recognised (has a complete patch corresponding to it added to the chart). However, the problems introduced by such data plans are made clear once we consider temporal overlays. Consider the overlay **bump+update->push**, which captures the idea that an **upper-segment** can be used to implement a **stack**, and that adding a new element to the bottom of the **upper-segment** can be viewed as pushing an element onto a **stack** (here a **stack** is thought of as a **list**, with new elements pushed onto the front). Figure 7.2 shows this overlay. Here we immediately run into a problem using the flowgraph formalism described earlier. Rules in a context-free flowgraph grammar have to have the same arities for their left and right and sides. This is clearly true for **bump+update** itself and this poses no problems. However, if we try and capture the overlay **bump+update->push** as a rule we find that we have a problem. The left hand side of the rule has arity (2,1), while the right hand side has arity (3,2). Furthermore, although the tie-point t3 occurs as input on both sides of the overlay, this is not true of t6 and any tie-point of the **bump+update** plan. It does not even correspond to the compound object (the **upper-segment**) represented by t1 and t2. It corresponds to the **upper-segment** viewed as a **list** (via a data overlay **upper-segment->list**). To cope with these features the basic bottom-up chart parser presented in Chapter 5 is modified as follows:

(1) Rules(plans) and temporal overlays are represented separately. The rules are used to drive the parsing as described in Chapter 6.

(2) When a complete patch is added to the chart its constraints are checked. If any of these constraints correspond to the fact that some of the tie-points occurring in the patch are the components of a compound data object (a data plan) then, provided these tie-points have not been recognised as forming an instance of the compound data-type previously in the recognition process, a new tie-point is created and an assertion stored in a data-plan database stating the relationship between the new tie-point and the ones occurring in the patch. This handles all data plans since we now have explicit single tie-points representing compound data objects made up of a suitably ordered collection of other tie-points.

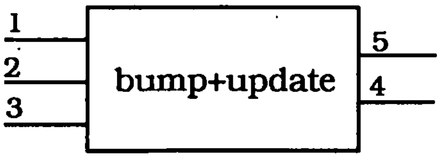
(3) Once the data-plans have been dealt with as described above, the set of temporal overlays is consulted to see if there are any overlays applicable to the patch just added. A new complete patch corresponding to the left hand side of the overlay is created and added to the chart. Tie-points in the new patch corresponding exactly to tie-points in the old patch are used directly. If the overlay specifies that a tie-point in the new patch corresponds to a data-plan type view of tie-points in the old patch then again, if this data-plan has not been used on these tie-points previously a new tie-point is created and information about it stored in the data-plan database. If the overlay specifies that a tie-point in the new patch corresponds to a data-overlay of tie-points in the old patch (or to some data-plan corresponding to tie-points in the patch) then , if this data-overlay has not been used on these tie-points previously a new tie-point is created and information about it stored in a data-overlay database. Figure 7.3

Graph to be parsed

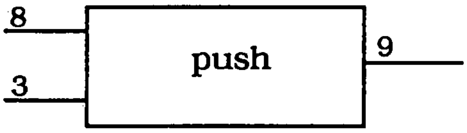


Patch added
to chart

Data-plan
Database



[6 is upper-segment(1,2)]
[7 is upper-segment(5,4)]



Data-plan Database

[6 is upper-segment(1,2)]
[7 is upper-segment(5,4)]

Data-Overlay Database

[8 is upper-segment->list(6)]
[9 is upper-segment->list(7)]

Figure 7.3
Use of Data Plan and Data Overlay Databases

shows this process at work. It is this ability of the modified parser to deal with compound data objects that leads to its ability to analyse programs to a high degree of abstraction, and that makes it the fullest implementation of Rich's ideas on program understanding. It is worth noting that the parser could also easily be modified to cope with data plans and overlays when running in top-down mode, although this has not yet been done in this project.

It should be noted that the fact that we can do the above is both a consequence of, and enforces, the following:

1) As stated already in Chapter 3 all tie-points represent a specific behaviour of some specific object at a specific time.

2) Different behaviours of the same object at the same time will be represented by different tie-points. Since the only behaviours directly applicable to the values manipulated by a program are the primitive behaviour functions the only ways in which an object can have more than behaviour is by means of data overlays mapping the primitive values to other (more abstract) data values, or by being considered to be parts of some compound data object. So, these tie-points representing other behaviours of the object will all be new tie-points with assertions in the data-overlay database. Since, for all primitive behaviours **B** we have the following:

$$\forall \mathbf{C}pst\ [\mathbf{C}(\mathbf{B}(p,s),t)=\mathbf{C}(\mathbf{B}(p,s),s)]$$

and for all non-primitive behaviours **B->C** we have:

$$\forall pst\ [\mathbf{B\text{->}C}(\mathbf{B}(p,s),t)=\mathbf{B\text{->}C}(\mathbf{B}(p,s),s)]$$

these tie-points will also all behave properly when used elsewhere[1].

3) If a behaviour **B** (say) of some mutable object p is different in situations s and t (i.e. **B**(p,s)≠**B**(p,t)) then **B**(p,s) will be represented by a different tie-point to **B**(p,t).

7.2 Problems with Control Flow

The next problem is that plan diagrams actually have two types of connections between nodes, whereas the flowgraph formalism only has one. These two types are data flow arcs, and control flow arcs. Now if control flow arcs were of the same fixed nature as the data flow arcs then we could have dealt with them simply by using the last (for example) of the entries in the inputs of a patch to represent the incoming control flow, and the last of the outputs to represent the outgoing control flow. However, because of the way parts of a plan can be widely separated in the program source code, the control flow between two parts of a plan can be separated by a large number of other operations. Furthermore, often the order in which two plans will be executed is often irrelevant, even though the control flow between them will be determined by the original program.

The key to performing control flow checking is to use the notion of control flow environments, and more particularly the notion of controlling conditions, as described in Chapter 4. In order to deal with

[1] It should be noted that this is a consequence of the fact that all structures such as lists will be recognised via a sequence of overlays such as **iterator->thread**, and **thread->list**., and **iterators** will be recognised as instances of compound data objects (consisting of a seed and a *mutable* function), thus implicitly (via the mutable function) carrying the relevant part of their situation around with them.

control flow constraints every complete patch in the chart is given a controlling condition as follows:

1) Primitive patches (i.e. those corresponding in a one-to-one fashion with nodes in the surface plan of the program being recognised) are given the controlling condition of their originating node. These nodes get their controlling conditions during the translation process from source code to surface plan (described in Chapter 6).

2) Apart from the patches for primitive operations, patches will in general contain sub-patches. The controlling condition for a patch is computed from the controlling conditions of its components, using the notion of plan conditions as described in Chapter 4. This serves both to compute the controlling condition for the patch, and to check that the component patches have the right controlling conditions to satisfy the control flow constraints.

7.3 Tests and Joins

Most of the important control flow (in the sense that the program would compute a different function were it to be changed) information in surface plans is carried by **test** and **join** nodes in the surface plan, and these can pose severe problems for the matching process. The main reason for this is that the parsing process is data-flow driven - partial patches are only ever extended by complete patches which match immediately needed entries in the partial patch. An immediately needed entry is a needed entry some of whose input or output tie-points have been instantiated (i.e. there is data flow from one or more of the (already found) components of the patch). However, **test** nodes have no output data flows so they do not necessarily connect in any direct way to the rest of the graph other than via their control

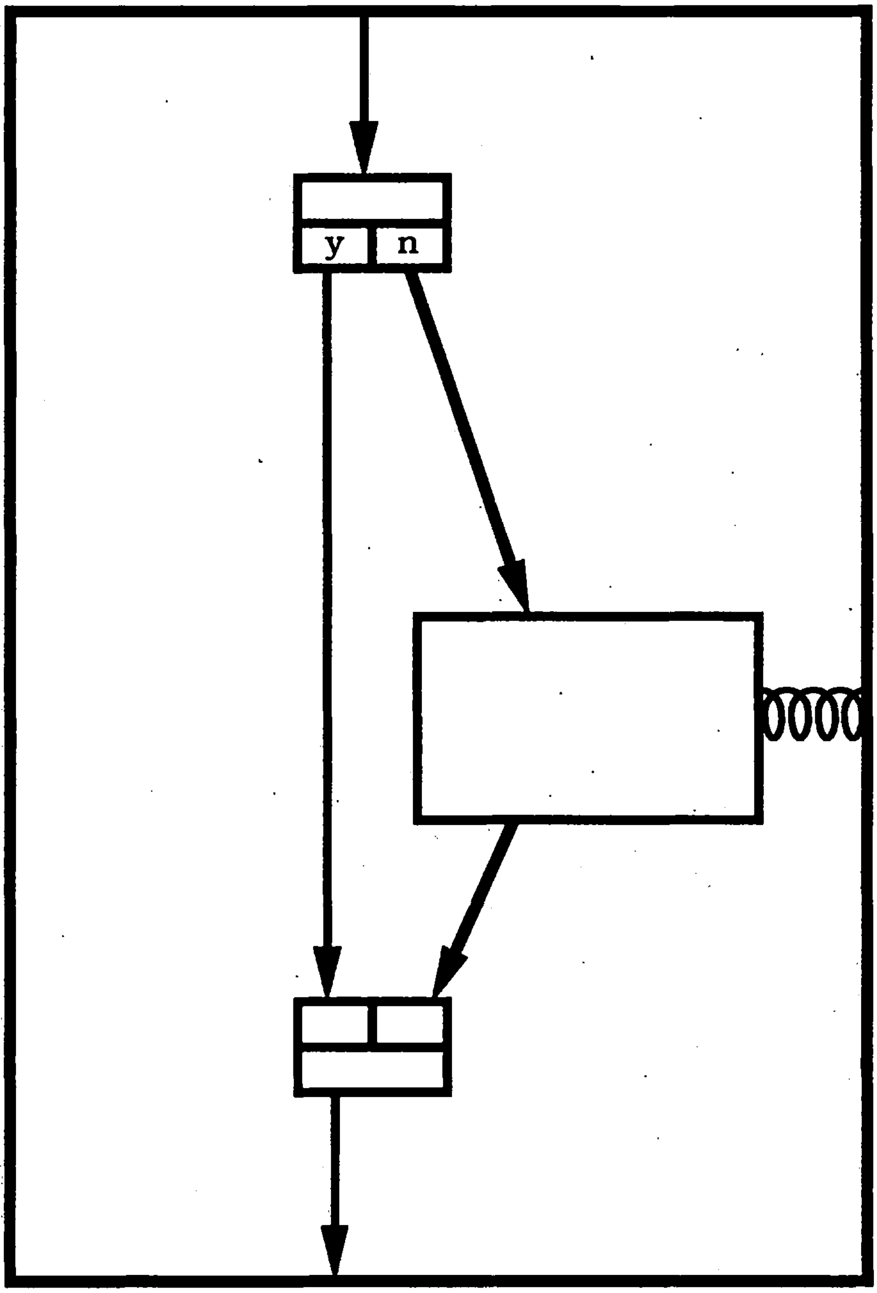


Figure 7.4
Iterative Termination

flow arcs, so it can be difficult for pure data flow parsing to recognise some plans involving **tests**. Similarly some plans only specify that they need a **join** (i.e. no data flows are specified in the plan at all for them) which again makes it impossible for pure data flow parsing to find such joins. However, all the examples of this phenomenon that we have encountered happen in one of the following situations:

1) plans like **iterative-termination**, shown in Figure 7.4, which are so general as to be of little use in themselves. Their main function in the plan calculus is to provide basic building blocks upon which other plans can be defined. Despite this we may wish to find them in order to provide a theorem prover with information to assist it in its reasoning. It should be noted that there are cflow arcs (i.e. ncflow and scflow arcs) between the test and the join in such plans.

2) Plans like **iterative-accumulation**, shown in Figure 7.5. Such plans have the property that the joins at least have data flow arcs which connect to the rest of the plan. Again there are cflow arcs between the test and the corresponding join.

3) The general case of trying to match a **test** to its corresponding **join** (in e.g. a **cond**). This is not possible by pure data flow parsing, since there is no direct connection between them.

However, the common property that all of these have is that there are cflow links between the **test** and its **join**. For the very general plans (e.g. a plain **cond** with no data flows involved anywhere, or a pure **iterative-termination** plan) we have decided to insist that the matching be strict i.e. the corresponding control flow environments must be strictly equal. So provided **tests** and **joins** are stored in the chart indexed by all their controlling conditions, as well as by their input

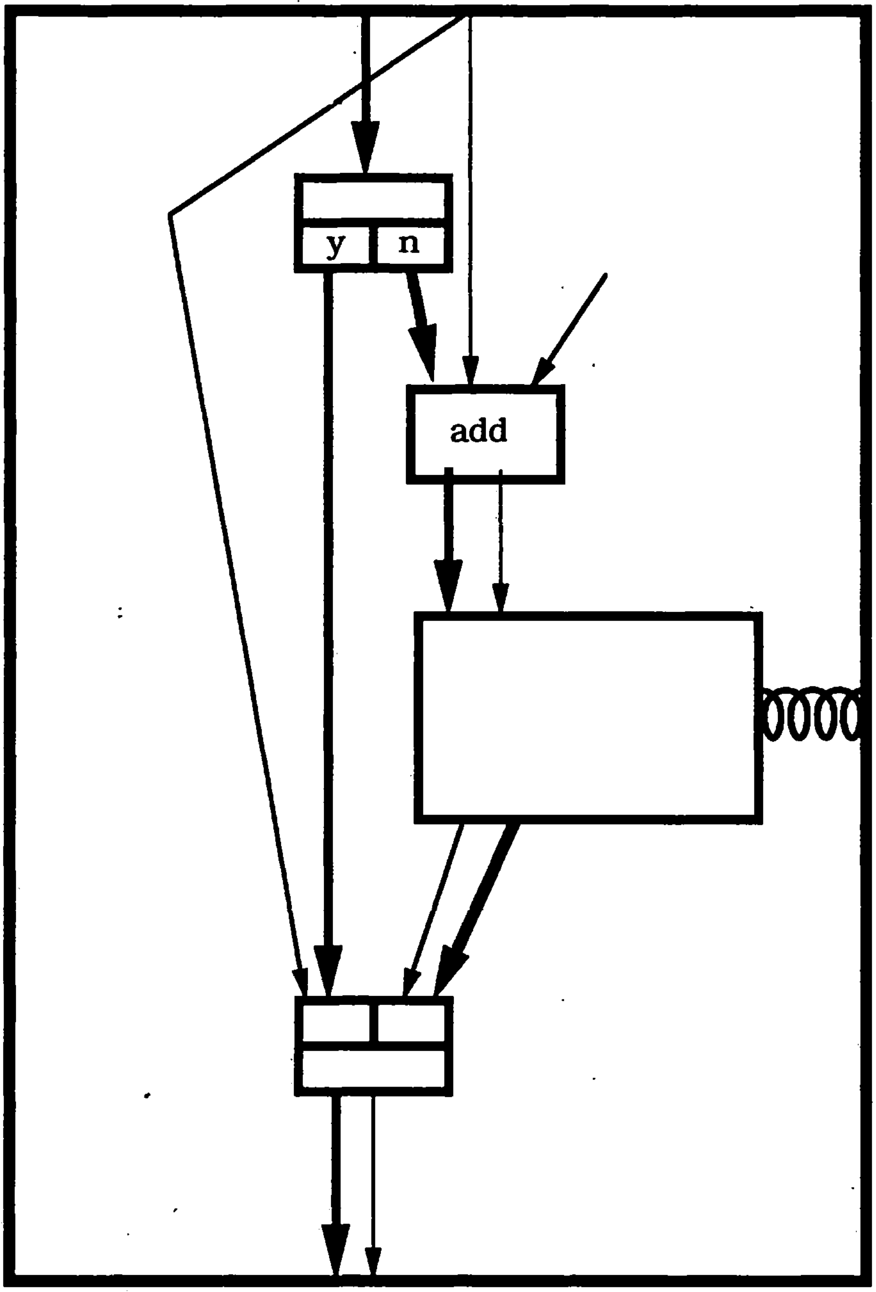


Figure 7.5
Iterative Accumulation

and/or output tie-points, then they can easily be found. Note that for other plans (e.g. **iterative-accumulation**) we have decided to adopt the following strategy:

Leave finding the **test** (unless it has been identified by data flows to one or more of its inputs) until both sides of the **join** have been instantiated by data flow parsing (note that because of the possibility of **n-join-output** nodes one side of the join will not necessarily tell us the other). This will happen automatically since none of the inputs to the needed **test** node will get instantiated in the case of a plan where the inputs to the **test** do not feed to anywhere else in the plan. Now look for a **test** such that the three control flow environments of the **test** enclose the corresponding control flow environments of the **join**. This condition is imposed by the cflow constraint between the **test** and the **join**. This is a very severe constraint on the control flow environments and, given the indexing of **tests** by their control flow environments, and the form of the controlling conditions we are dealing with, in which the encloses relation can be recognised syntactically (as mentioned in Chapter 4), enables reasonably rapid identification of the correct **test**.

Another problem which the parser has to deal with occurs when, as a result of data overlays or data plans, new tie-points are created inside a conditional, and these have to be communicated to whatever follows the **join** in the conditional. An example is shown in Figure 7.6(a). In this case we really need to add a new **join** with this new tie-point as an input to the **join** (as well as a tie-point corresponding to the same overlay of the corresponding tie-points on the other side of the **join**), and to create an extra output tie-point representing the corresponding overlay of the corresponding outputs from the conditional, as shown in Figure 7.6(b).

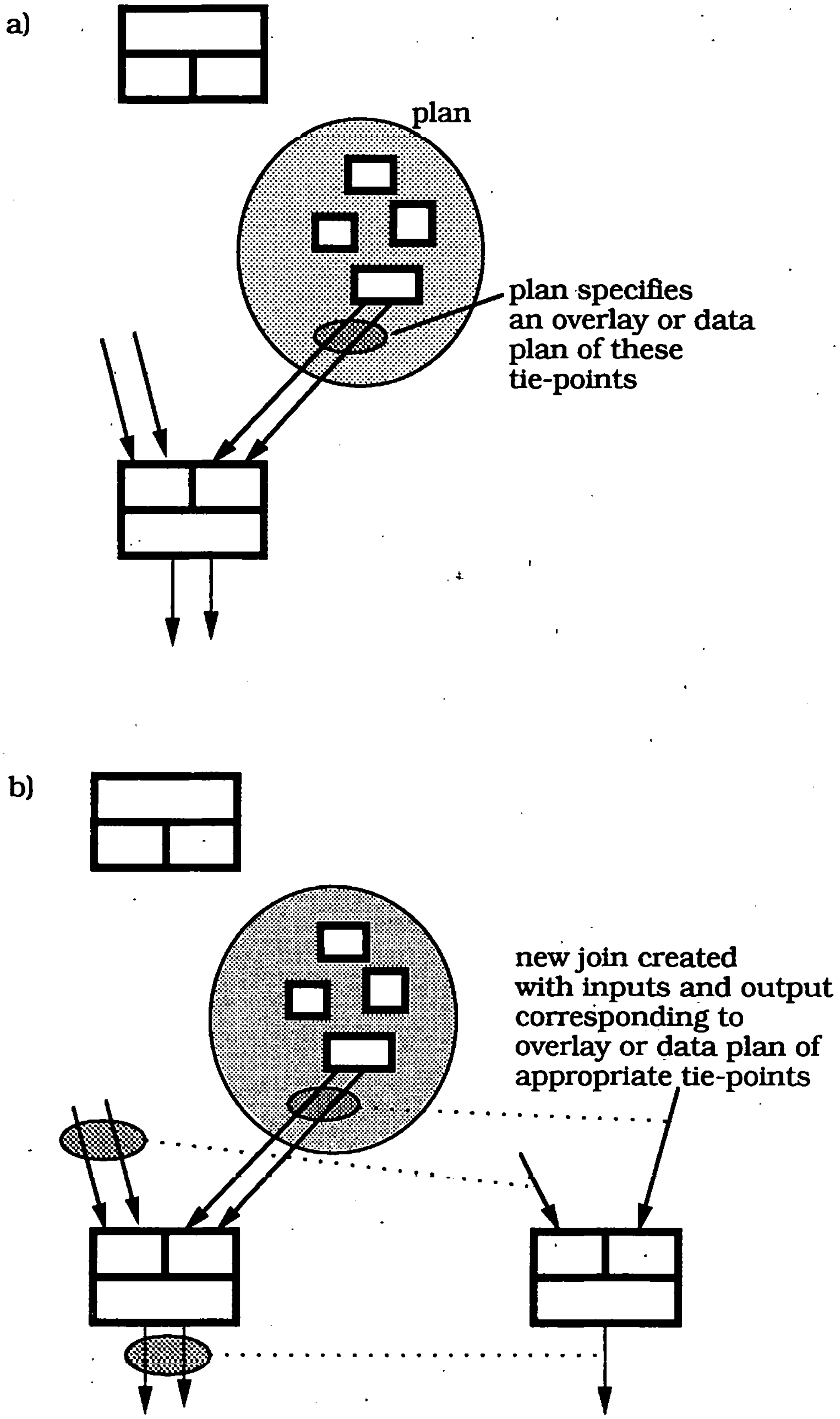


Figure 7.6
New Joins and Tie-points Corresponding to
Data Plans or Overlays

Another case in which **joins** can cause problems is the case when an appropriate **join** does not even exist. This happens when we have partially recognised a plan in which the actions which define the **join** we are looking for (i.e. whose outputs should connect to it) are either in generalised control flow environments i.e. we have conditional plans, or are inside a conditional, but feed their outputs to other actions in the conditional, rather than to a **join**. Figure 7.7(a) illustrates this situation. As noted in Chapter 4 however, adding a **join** with the appropriate controlling conditions changes nothing about the semantics of the graph, so in this case we simply create a **join** *on demand*, as shown in Figure 7.7(b).

There is one final problem caused by **joins**. This occurs in situations like that shown in Figure 7.8 where in order to recognise a plan (albeit conditionally) we need to match through a **join**. Note that this matching through a **join** might be required in either direction through the **join**. This is in some ways a rather nasty problem as it can potentially happen every time we need to extend a partial patch. The solution we have adopted is to flag each tie-point that is an input to a join as a "join-input" tie-point, and to flag every tie-point that is an output from a join as a "join-output" tie-point. Then two tables are maintained (actually hash tables) - one table for inputs to **joins** giving the corresponding output tie-point, and one table for outputs from **joins**, giving all the corresponding inputs. This enables the parser to rapidly find the appropriate matching tie-points on the other side of **joins**. Note that when matching through a **join**, whichever patch is feeding into the **join** must be treated as if its controlling condition were that of the branch of the **join** it is feeding into (even if it is not), otherwise the parser can be misled into thinking it has found a plan with a weaker controlling condition than it actually has. Figure 7.9

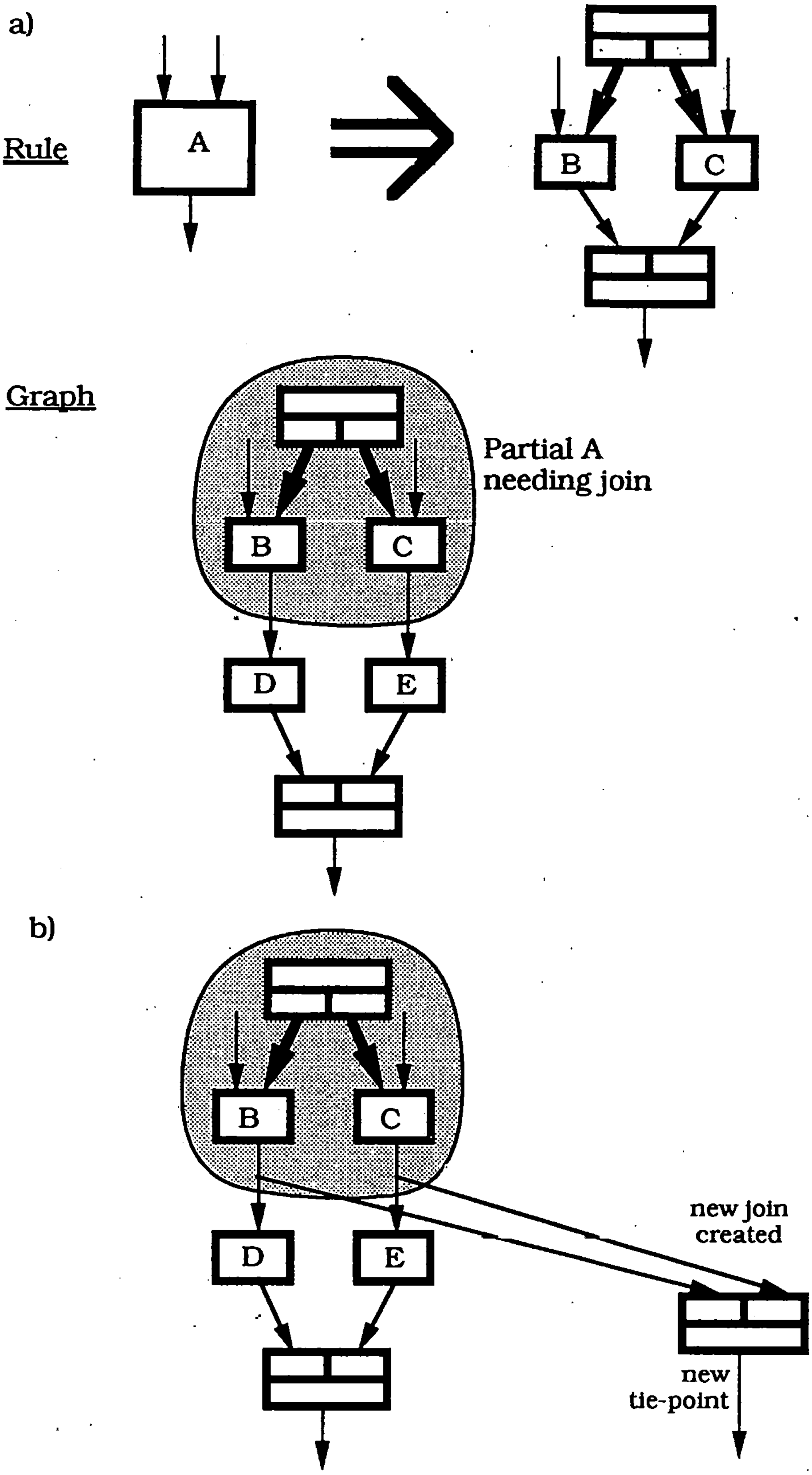
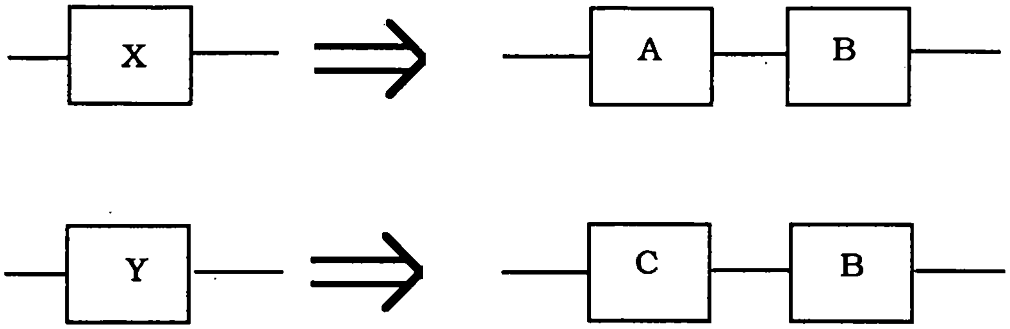
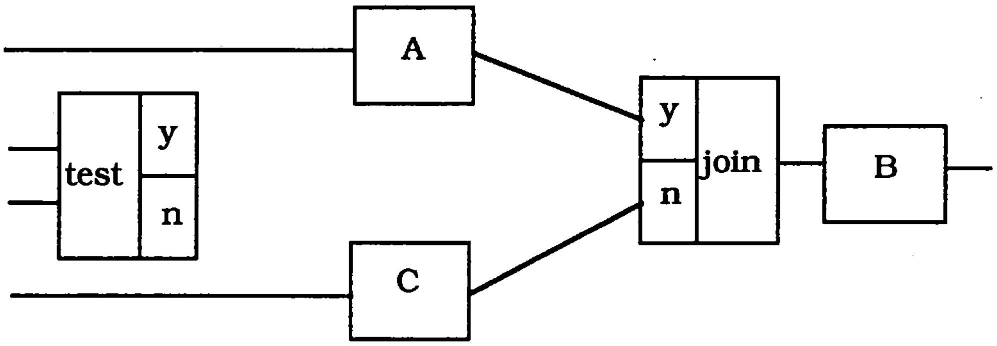
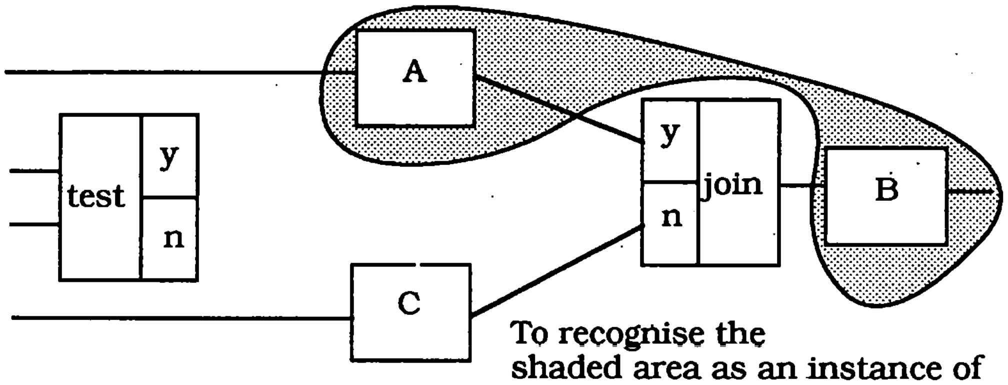


Figure 7.7
New Joins For Internal Tie-points
of Conditionals



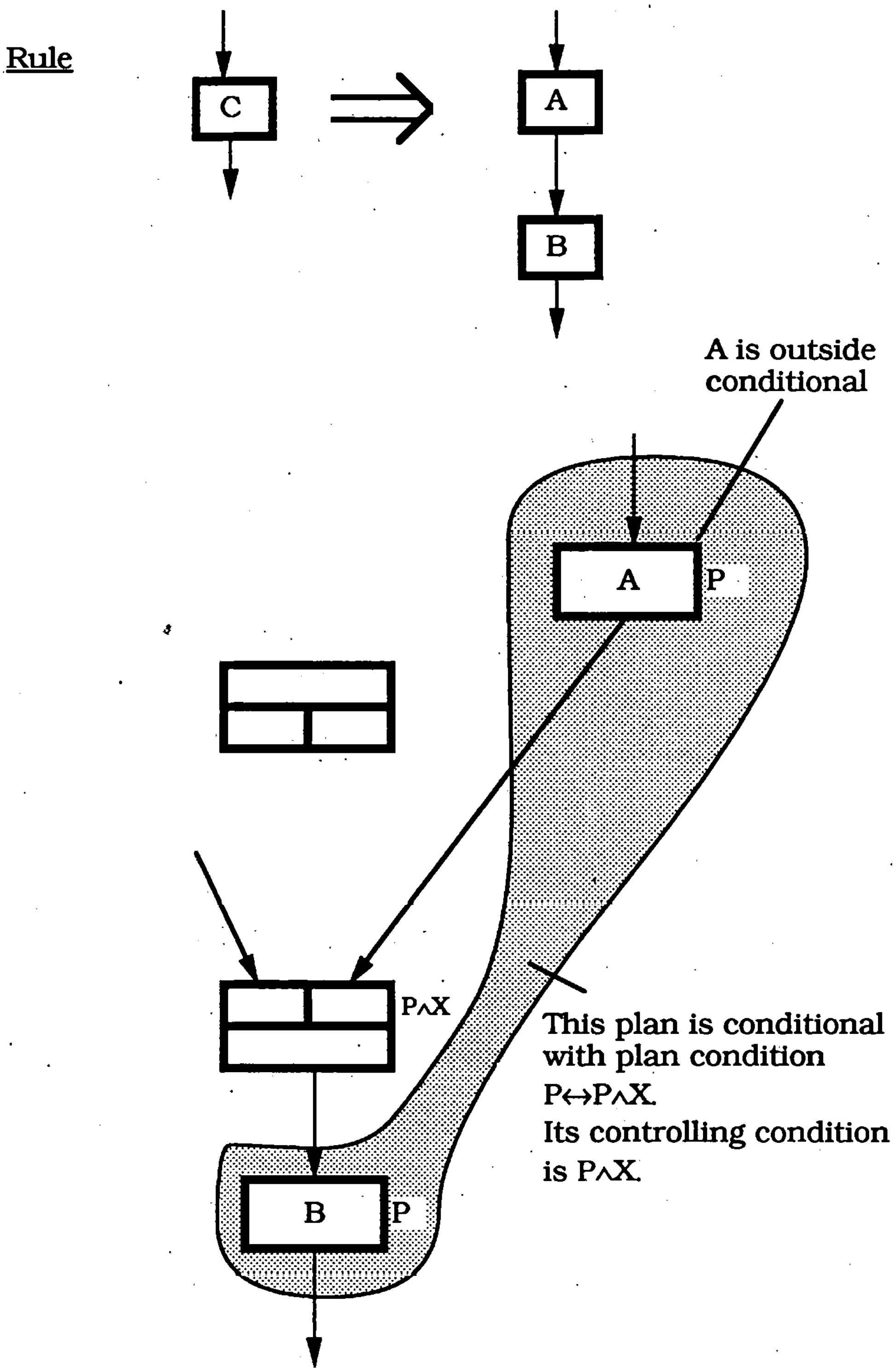
if test then X else Y endif;
=> if test then A.B else C.B endif;
=> if test then A else C endif; B;





To recognise the shaded area as an instance of X we need to match through the join.

Figure 7.8
Matching Through Joins



A is outside conditional

This plan is conditional with plan condition P↔P∧X. Its controlling condition is P∧X.

Figure 7.9
Controlling Conditions After Matching Through Joins

illustrates the problem. This is achieved by simply adding the requirement:

(controlling condition of exit of **join**)
↔ (controlling condition of appropriate branch of **join**)

to the plan condition of the patch being extended, which ensures that we get the right controlling condition for the whole patch.

One final point needs to be re-emphasised here. Despite the fact that all **join**s actually only have a single succeed and fail input, and hence a single output, we will continue to show **join**s with multiple inputs and outputs in diagrams rather than showing multiple **join**s. This is purely to keep the number of NAPEs in diagrams to a minimum.

7.4 Recursive Roles in Loops

Loops, represented recursively as described earlier, also pose problems of arity matching. This is because the recursive role in a loop essentially represents the whole loop, and a loop may have multiple plans within it. Each of these plans, as specified in the plan library, expects a recursive role of the same arity as the plan itself. So again this matching must be handled separately, and is essentially dealt with by allowing the recursive role in a loop to match recursive roles in plans <u>provided</u> that it is appropriately connected. This is done by leaving the matching of the recursive role until all the connections to it are known, in which case we just have to check that the inputs to the required recursive role are inputs in the actual recursive role which correspond to the inputs to the plan in the loop as a whole, or if the plan does not specify any inputs to the recursive role, but simply that it is recursive, then the match is simply taken to succeed.

7.5 Breaking Programs up into Smaller Segments

The surface plan corresponding to a program can be very large. Rather than dealing with it as a single undifferentiated graph it is broken up, by the translator from source code to surface plan, into a hierarchical arrangement of smaller pieces (henceforth referred to as segments). Each segment corresponds to some meaningful piece of the source program, and may themselves refer to other segments. For instance each loop in the program corresponds to a loop segment (which is recursive), a conditional corresponds to a conditional segment, and so on. Each procedure definition also gives rise to a segment, and other parts of the surface plan can use this segment as if it were a primitive. Similarly, a segment containing a loop will use it as if it were a primitive. This results in the main program's surface plan mostly consisting of segments.

The parsing algorithm's initialisation phase is then modified as follows:

1) First of all complete patches are added to the chart for each primitive operation occurring in the segment.

2) For each non-primitive segment X in the segment being considered the chart parser calls itself recursively with X as the segment being analysed. This creates a new separate chart for X. For loops and conditionals the patches in this chart are simply added into the 'calling chart'. For procedure calls copies of the patches (appropriately modified to maintain the connectivity of the surface plan) for the procedure are added to the main (calling) chart. Controlling conditions are handled as follows:

For loops and conditionals the outermost controlling condition environment is that of the "calling" segment. Controlling conditions are simply included along with the patches when they are added back to the calling chart. When procedures are analysed they are given a token initial controlling condition. When a procedure call is analysed its patches are copied into the calling chart as described above and the initial controlling condition is substituted for by the calling chart's environment. Any sub-environments of the procedure are substituted for by appropriate new sub-environments of the calling chart.

Note that partial patches are added before complete ones, and that if a patch is an extension of another, then the extended patch is added first, ensuring that a minimum of work already done is duplicated.

This mechanism has several advantages. Firstly, when analysing loops, the single recursive role in the loop segment must be the recursive role for any iterative plans being recognised in the loop. Secondly, although it involves copying patches (appropriately modified) from the chart representing the analysis of a procedure into the chart for a segment which calls the procedure, it does not involve reanalysing the procedure as in-line code expansion prior to translation, or 'in-graph' graph expansion prior to analysis would. It may even make it possible to only copy some of the patches (representing the system's best guess at the function of the procedure or loop or whatever) although for safety at the moment all the patches are copied. Thirdly, this mechanism makes it much easier to deal with some of the program transformation issues discussed below.

7.6 Program Transformations

Another problem that occurs when matching real programs is that often the surface plan contains some subgraph corresponding to some standard plan in the library, but it is not in the standard form. An example of this will be seen in Chapter 8, and this makes it necessary to perform various transformations on the graph in order to convert it into a form in which standard plans can be recognised. For example this may involve transforming the graph from one obtained from a program involving a **while** loop to the graph that would have been obtained had the program been coded using a **repeat ... until** loop.

There are several types of program transformation performed by the parser. Some of these are simply handled by rules, and to some extent simply represent mathematical properties of the various operators involved. They are there simply to avoid the need for a theorem prover to deduce the relevant properties. It should be noted that the need for these rules is one of the main criticisms that Murray[1986] makes of the plan diagram formalism. However, adding a rule expressing the relevant relationship, does not seem any worse than adding an axiom as he would have in his system. The other type of transformations are much more like the various loop transformation operations that Murray[1986] applies to programs to get them into a standard form.

The first class of transformations (i.e. those handled by rules) includes rules for doing such things as:

1) <u>not removal</u>. This is done by rules like that shown in 7.10(a). This simply adds a patch with the same inputs as the not/test combination, and with the succeed/fail controlling conditions reversed.

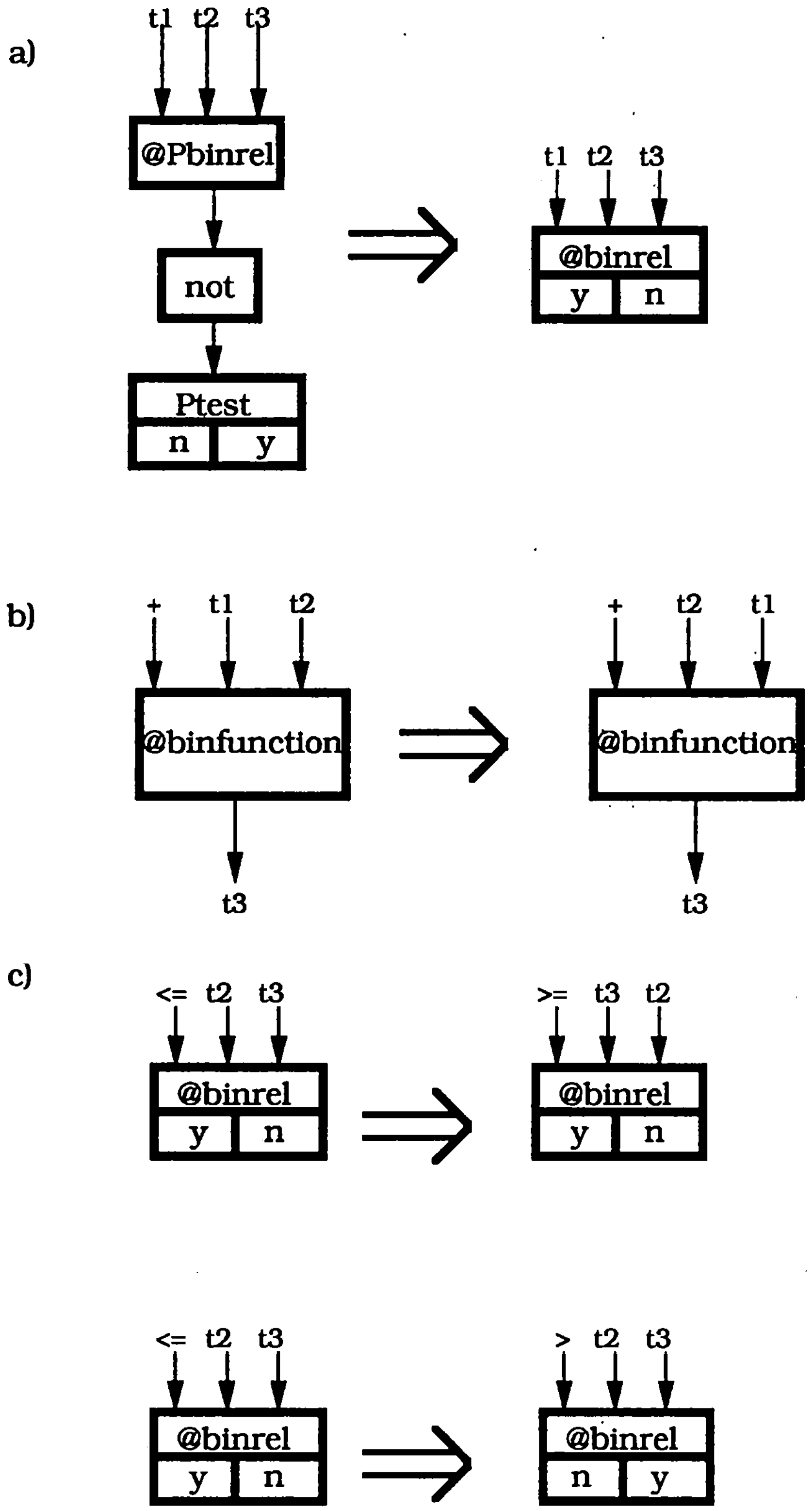


Figure 7.10
Simple Program Transformation Rules

2) expressing commutativity and other relationships of operators. For example the rule shown in Figure 7.10(b) expresses the fact that addition is commutative. Similarly their are a whole collection of rules for expressing relationships between and about the various comparison operators. For instance Figure 7.10(c) shows some of these for "<=", ">", and so on.

The second class of transformations are much more like what are normally thought of as program transformations. Although more will no doubt need to be added, there is only one of these at the moment:

3) iterative flag test removal. This is used when the programmer has used a "flag" to control the iteration, rather than the test that "really" controls the iteration. This is seen in the following type of code:

```
flag:=false;
while not flag do
begin
    .
    .
    .
    if condition then
         begin
              .
              .
              flag:=true
         end
    .
    .
    .
end;
```

where the true controlling condition is actually the condition indicated in the code template above. In surface plan terms this corresponds to Figure 7.11. As can be seen, the crucial factors for determining whether or not an iterative flag test is being used are the following:

a) The initial test is guaranteed to succeed (in the sense that the loop will be entered, rather than being exited immediately).

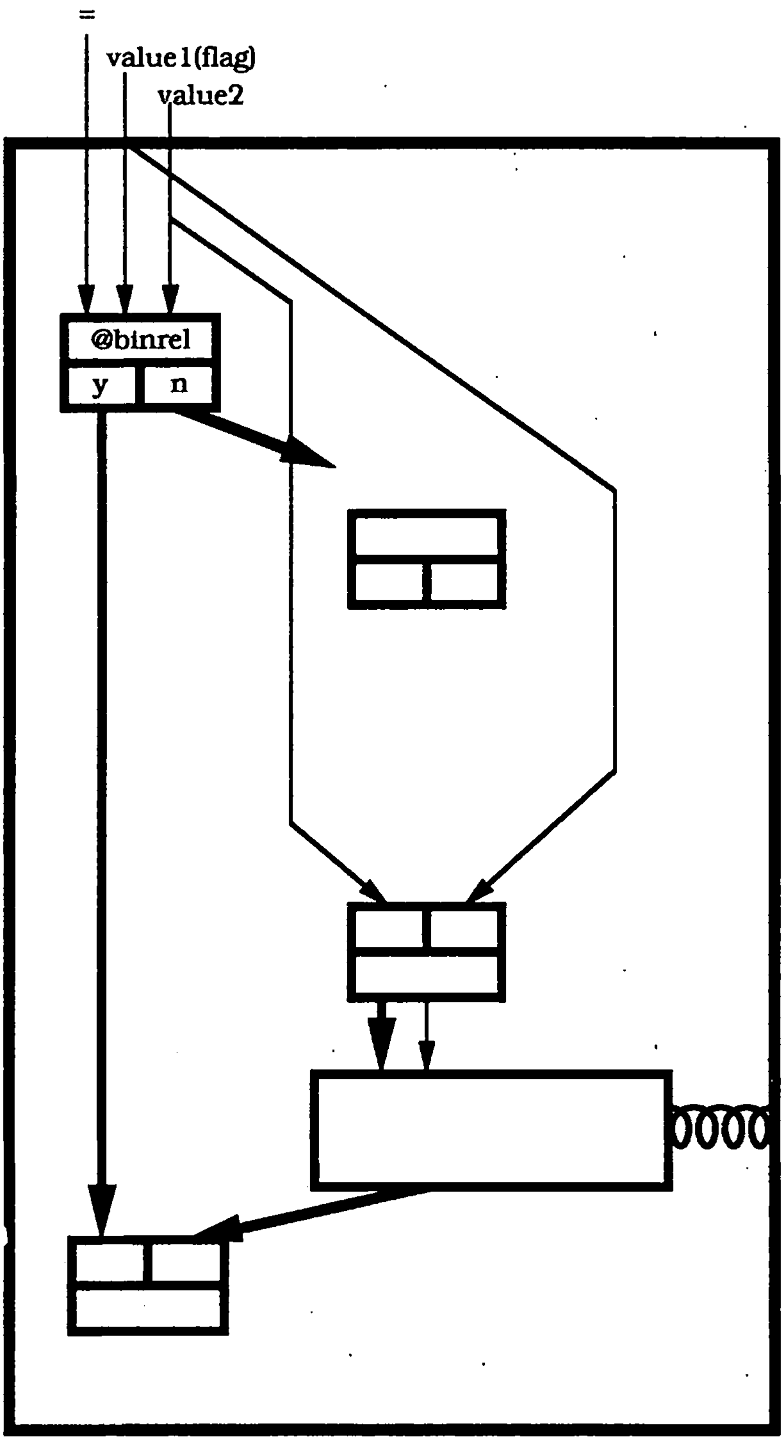


Figure 7.11
Surface Plan Containing An Iterative Flag Test

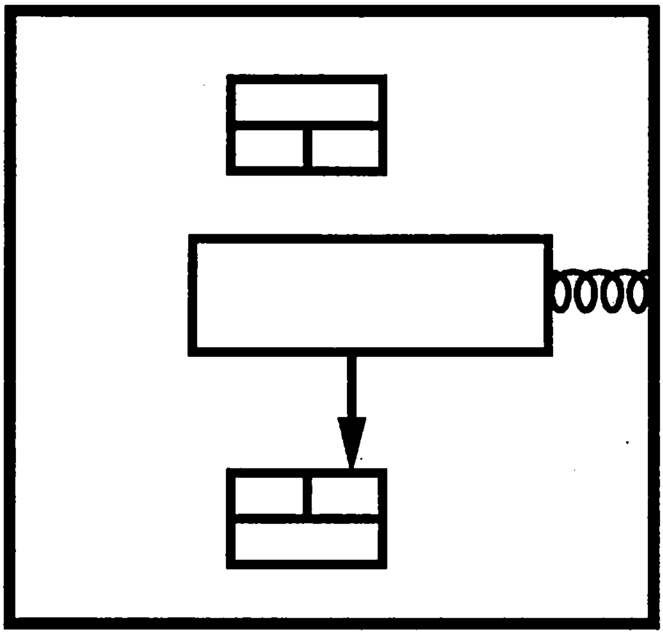


Figure 7.12
After Removal of Iterative Flag Test

b) There is a **join** before the recursive role in the loop such that one branch of the **join** sets the flag to a value (i.e. value2 in the diagram) which will cause the loop to exit next time round. In this case, *provided there is no computation in the loop before the initial test*, the loop can be transformed by simply removing the initial **test** and its corresponding **join**, and moving the recursive role to before the **join**. Figure 7.12 shows the transformed version. Note that this can be done by simply changing the controlling conditions on the recursive role to that of the branch of the loop which passes the initial value of the flag out, and removing all reference to the controlling condition of the initial **test** in the controlling conditions of all the other actions in the loop.

7.7 Related Program understanding Work

The work which has the most similarity to that reported in this and the previous chapter is that done by Brotsky [1984], and Wills [1986,1990]. Brotsky developed a parser which generalises Earley's [1970] algorithm for context-free string grammars to flowgraphs of the same type as those considered here, although his notation and formalisms were different. In particular he had no explicit mention or representation of tie-points. Brotsky's algorithm also could not cope with fan-out, or fan-in from or to nodes, making it not directly applicable to parsing of surface plans. Furthermore, his algorithm runs in a strictly top-down left-to-right fashion. Wills[1986,1990] generalised Brotsky's algorithm to cope with fan-in and fan-out, and has used this generalised parser to produce a program understander (known as the Recogniser) which does a similar job to that described in this chapter, but which is more limited in a variety of ways. She also ran into many of the same problems we encountered, although her solutions are often different. This section will describe their work, and the ways in which it differs from that described in this thesis.

As already stated, Brotsky's parser is a generalisation of Earley's [1970] algorithm for parsing context free string languages. It can be thought of as using the grammar to construct a non-deterministic finite-state push-down automata for recognising flowgraphs generated by the grammar. If this automaton then operates in a strictly top-down left-to-right fashion we obtain Brotsky's algorithm. His algorithm works by maintaining at any time a list of all the possible states in which the automaton could be. Since the automaton is non-deterministic there will in general be several such states. Any such state consists of the states of each recogniser and the sub-recognisers it has called. A state is described by several pieces of information:

(i) Where in the graph it was started (called)

(ii) Where in the graph it has got to.

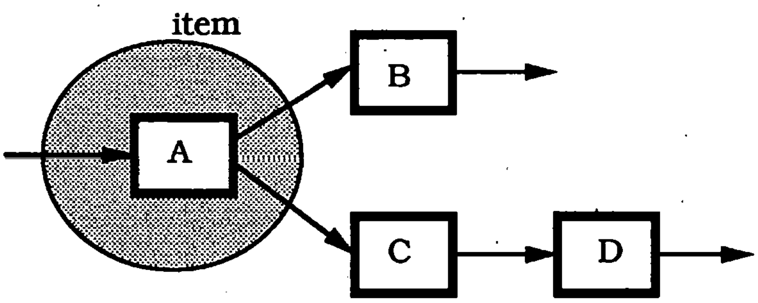
(iii) What sub-recognisers it has invoked at the place it has reached.

(iv) What recogniser invoked this one i.e. which other recogniser it should return control to if it completes its recognition task.

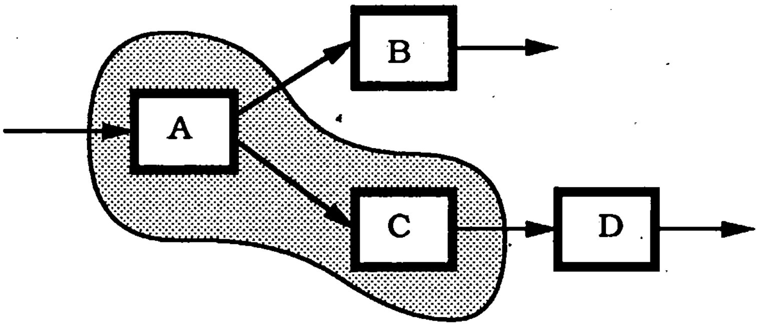
The first three of these items are very similar to information the chart parser keeps in its patches. (i) corresponds to the input tie-points of a patch. (ii) corresponds to the active tie-points of a partial patch, and (iii) corresponds essentially to the needed entry in a patch. There is no analogue to (iv) in the chart parser, unless one regards the whole chart as in some sense recording this information. This similarity reflects the fact that Earley's algorithm can be viewed as an early form of chart parser running in strictly top-down, left-to-right fashion, and indeed Brotsky's algorithm can be viewed as a special case of the parser presented here. However, there are various important differences between the chart parser presented in this thesis and Brotsky's algorithm.

The first, and perhaps most important, of these, is the fact that Brotsky's algorithm only runs top-down. This gives his algorithm a slight efficiency advantage compared to the top-down chart parser, in that rather than having to search the chart (albeit efficiently) to find out which patches can be extended by a new complete one, he explicitly keeps track of this with each patch (item (iv) above). This is possible because of the top-down nature of the parsing, and indeed, in top-down mode, our parser could easily be modified to do this. However his algorithm cannot run bottom-up, and so Will's generalisation of Brotsky's algorithm essentially starts the parser off in top-down mode at every point in the graph, looking for every possible non-terminal, thus simulating a sort of bottom-up parsing.

There is another major difference between the chart parser presented here and in Lutz[1986, 1989] and Brotsky's algorithm. The chart parser actually constructs extra patches, not corresponding to any that his algorithm constructs. This is because, although Brotsky's parser corresponds to the simulation of a non-deterministic automaton, there is one source of non-determinism, inherent in the parsing problem, that is ignored by his algorithm. The scanning operation of his algorithm always chooses a node in the graph to read next, and the items constructed follow from this choice. A different choice of node to read next will in general lead to a different set of items being constructed. For an example of this see Figure 7.13. The chart parser adopts a different strategy - where there is a choice of nodes that could be read next (i.e. where there is more than one patch that could at a particular time extend a given partial patch) the chart



If Brotsky's algorithm has found the item(patch) shown, then it has a choice of whether to extend it with B or C. If C chosen we get:



and there is now a choice about whether this is extended by B or D. His algorithm cannot go back and form the item consisting of A and B.

The chart parser described in this thesis will form patches A, AB, AC, ABC, ACB, ACD, etc.

Figure 7.13
Deterministic Node Choice in Brotsky's Algorithm

parser constructs patches (items in Brotsky's terminology) corresponding to extending the patch with each separately. This can mean the chart parser does redundant work. However this was a quite deliberate decision, as ultimately the chart parser has been designed for program debugging, and this can enable it to find more near-misses than Brotsky's algorithm would.

Related to this last point is another difference between the chart parser and Brotsky's algorithm. This is to do with the fact that the chart parser can "turn corners"[2] i.e. the chart parser can bypass a node in the graph, and then, because it has active input as well as output edges can still find the node it has bypassed. Brotsky's algorithm cannot do this because it insists that a node is only eligible to be read, if all its input tie-points connect to the item (patch) so far. Again this is useful if as many near-misses as possible are to be found. Figure 7.14 illustrates this point.

As already stated, Wills[1986, 1990] has used Brotsky's algorithm as the basis for her work on plan recognition within the plan calculus. Apart from the fact that it is not so adept at finding near-misses as the system described here, there are various other important differences. The main one is that as yet her recogniser cannot cope at all with data plans and data overlays, which greatly restricts the kinds of plans that can be recognised. As a result of this it also cannot really cope at all with programs involving side-effects.

Her system also uses the notion of control-flow environments (indeed the terminology is hers - I originally used the phrase "mutually

[2] This phrase was used by Linda Wills when we were discussing our different approaches to the plan recognition problem.

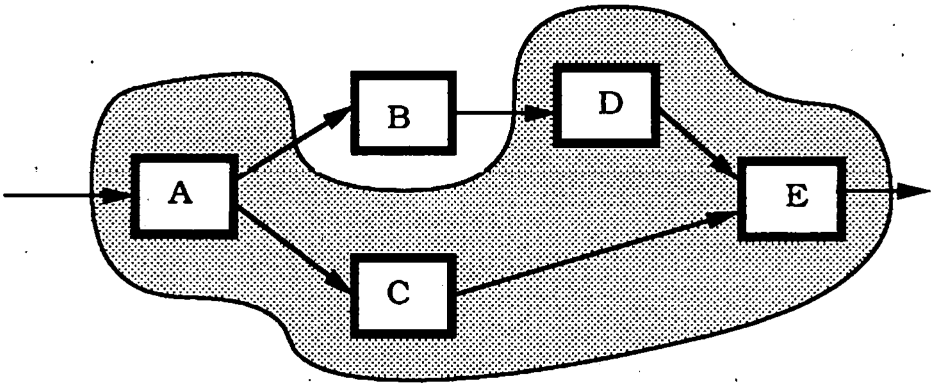
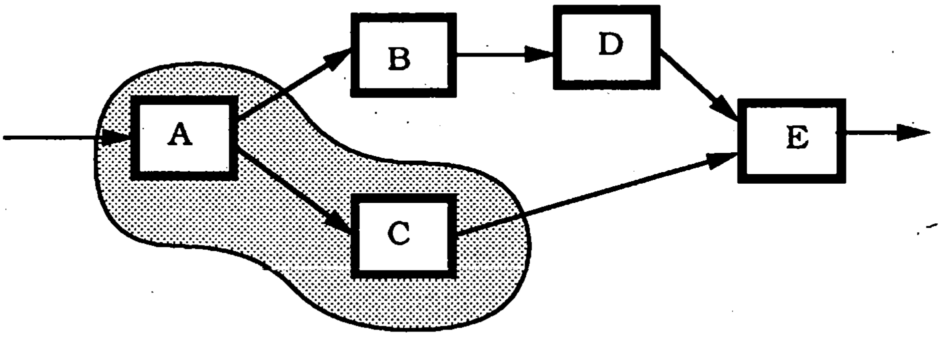


Chart parser can recognise this partial patch even if B is missing, since it can bypass B, and "turn a corner" back from E to find D.

Brotsky's algorithm can only find the item below:



since it cannot "read" E until both C and D have been "read", and it can't get to D unless B is present.

Figure 7.14
"Corner Turning" To Find More Near-Misses

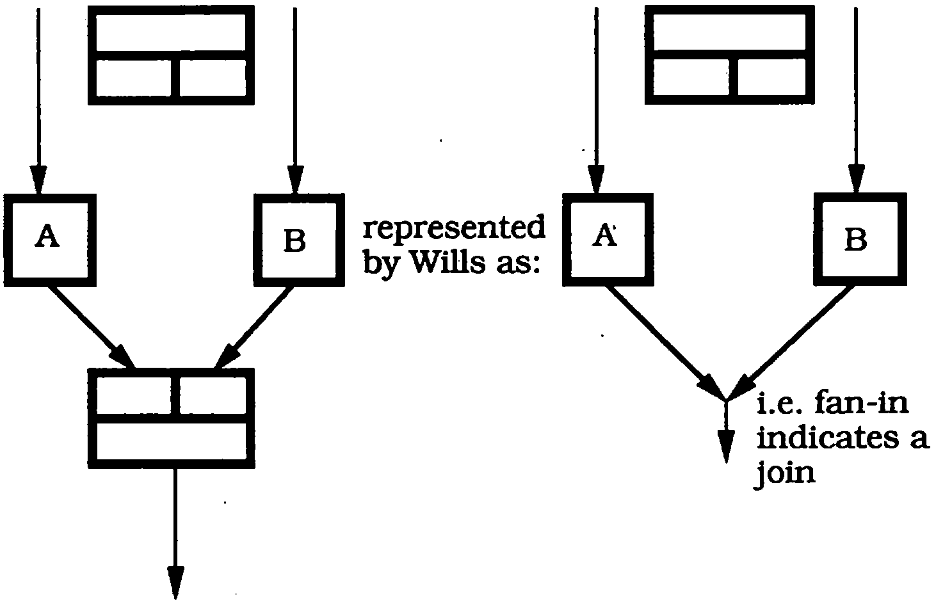
occurring actions" to capture the same idea). However, the notion of generalised control-flow environments and their relation to controlling conditions, used by the chart parser is new and enables it to recognise many plans that her system would fail to recognise, especially those like the ones described in Chapter 4.

There are many other relatively minor ways in which the system described here and Wills' differ. For instance she has done away with joins altogether, representing them by what would be fan-in at tie-points in our system. This is quite nice in many ways, as it avoids the need to match through joins as described above, but does cause a problem with "straight-through arcs" (see Figure 7.15 for an example). This leads to her having to annotate the data flow arcs themselves with control-flow environment information, in order to deal with this. Her solution also avoids a problem with the transitivity of joins, discussed in Wills [1990]. In IDS this is avoided by the introduction of n-join-output nodes as described in Chapter 4. Her solution also avoids the problem of dealing with joins in programs involving lots of succeed, and fail inputs and of trying to match them against say a join-2-outputs node in a plan. In our case we have made all joins essentially join-output, thus avoiding the problem.

Finally, it should be noted that, as already remarked, Wills' about-to-be forthcoming thesis (personal communication) is now substantially based upon the chart parser presented here and in Lutz[1989].

7.8 The Plan Library

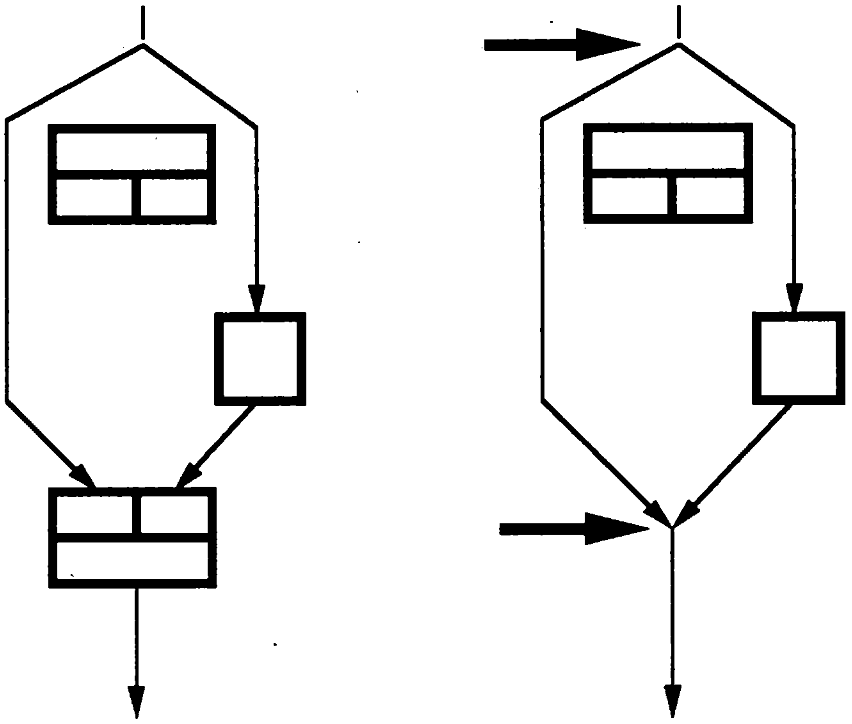
Although most of the plan library is taken directly from Rich [1981], and interested readers are referred there for a complete list of all the plans and overlays we have "borrowed", it has been necessary to add a few of our own. Some of the plans we have added have just been



represented by Wills as:

i.e. fan-in indicates a join

However, plans with "straight-through" arcs cause problems, since this approach would cause one to identify tie-points that should not be identified.



The two tie-points indicated should not be identified.

Figure 7.15
Straight-through Arcs, and Fan-in to Represent Joins

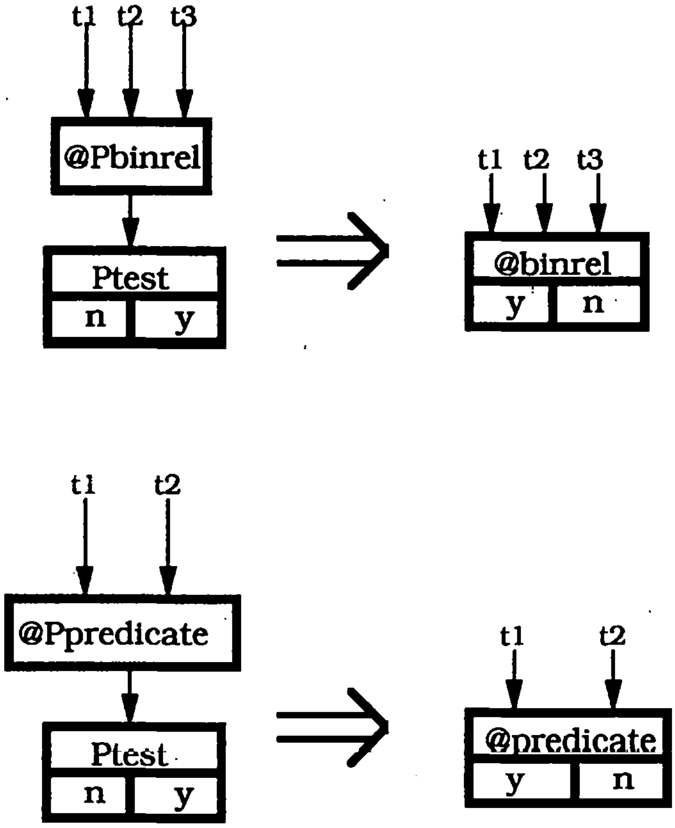


Figure 7.16
Rules for Pascal Tests

ones mapping Pascal operations onto plan calculus ones. We have not used the plan calculus ones directly in the translation process because they do not always have exactly the same semantics, and at some point these should be a given a proper semantics within the plan calculus. At the moment these plans are simply given to the system directly as grammar rules, rather than as plans and overlays etc. in Rich's compact notation. An example of such a rule(for Pascal tests) is shown in Figure 7.16.

7.8.1 Translating Plans into Flowgraph Rules

So far we have seen how plans expressed in the compact frame-like notation can be viewed as graphs by interpreting the constraints as arcs of the graph. However, this has been done informally. Wills [1986,1990] has used a subset of Rich's plan library in her program recognition system, but has converted the plans to graphs by hand. Not only is this tedious, but is also error-prone, particularly when the final plan library might contain thousands of plans. This project has automated the conversion of Rich's plans to graphs, so one of the achievements of the work reported here is to essentially make Rich's notation machine readable, so that the recognition process can use plans expressed as graphs which are guaranteed to correspond to the plans as expressed in the compact notation, and hence to the inference rules as described above. The conversion process is surprisingly tricky to get right, so will be described in some detail. However, before describing this translation process it is necessary to describe the format in which the graphs will be represented.

A rule will be expressed as a list of nodes (which as seen in Chapter 5 are referred to as NAPEs). There will be one NAPE for each box(action) in the graph. Each operation NAPE will be represented as a

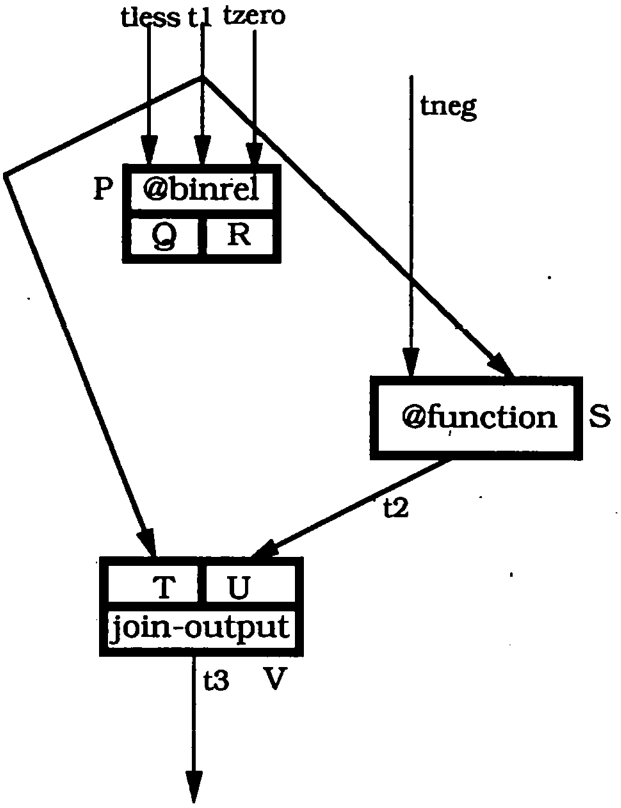


Figure 7.17
A typical rule right hand side

4-element structure. The first element will simply be the type of the corresponding action (e.g. **@binfunction**). The second element will be an ordered list of the inputs to the NAPE, and the third element will be an ordered list of the outputs from the NAPE. Where the i^{th} output of one NAPE connects to the j^{th} input of another, the i^{th} entry in the output list of the first NAPE will be equal to the j^{th} entry in the input list of the second. The last entry will be a variable denoting the controlling condition of the NAPE.

Test NAPEs are represented by a 5 element structure. The first element is the type of test (e.g. **@binrel** or **@predicate**), the second is an input list as described above, and the last three are variables denoting the controlling conditions for the test itself and its succeed and fail environments.

N-join-output NAPEs are represented by an n+4 element structure. The first element is the word **"join-output"**, the second is an input list (with n inputs) as described above, the third is a single element output list, and the last n+1 elements consist of variables denoting the controlling conditions for each of the n inputs, followed by a variable for the controlling condition of the **join** itself.

Where an input value flows into more than one input port of the NAPEs in the graph, all the appropriate entries in the relevant input lists will all have the same value. So the plan in Figure 7.17, will be represented as follows:

```
[ [@binrel [tless t1 tzero] P Q R]
  [@function [tneg t1][t2] S]
  [join-output [t1 t2][t3] T U V] ]
```

This is very like the notation used in Chapter 6 for representing surface plans, except that in rules the "tie-points" will be variables rather than actual tie-points. The one exception to this is when a rule references a known constant, in which case it will have the same (instantiated i.e. non-variable) tie-point in the appropriate places in the input or output lists of the rule, as would the surface plan if a program referenced the constant.

7.8.1.1 Translating Temporal Plans into Rule Form.

The translation process from plans expressed in the plan notation to graph-like rules for use in the recognition process proceeds as follows:

1) The plans are read in, and stored in simple KRL-like frames [Bobrow and Winograd, 1977a, 1977b], with the obvious slots such as preconditions, postconditions, and constraints etc.

2) IOSpecs are used to define the set of possible nodes which can occur in rules. Essentially the information in an IOspec is used to decide how many inputs the operation has, and how many outputs. The order in which these occur in the plan definition is used to determine the order in which the inputs will appear in the inputs list to a NAPE representing an operation of this type. The names of the inputs are used as suffixes to determine the names of the inputs to any NAPEs of this type occurring as roles in a rule. So, if a rule has a role op1 (say) of type **@binfunction**, then initially it will be represented by a NAPE of the form:

```
[@binfunction
      [op1.op  op1.input1  op1.input2]
      [op1.output]
]
```

2) Data plan definitions are processed to determine what roles an instance of the data plan will have.

3) Temporal Plans are processed. For each temporal plan, a list of its roles is first constructed. These are then separated into data roles, and operation roles. Each role has a name and a type. The right hand side of the rule is constructed as follows:

a) Each operation role is converted into a NAPE as described above. This gives us a completely unconnected graph structure G since no two NAPES have any inputs or outputs in common.

b) The constraints for the plan are collected together. As described earlier this may involve recursively following **specialization** and **extension** links in the frame structure to collect together all the relevant constraints. When we follow these links, if a plan reached in this way has not been processed yet, then it is processed. As will be seen this processing essentially involves making substitutions for some of the tie-points in the input or output lists of the NAPES representing the plan (to represent the data-flows). These substitutions, along with the constraints are passed back up the recursive call (to process the node) to the plan which caused the call to be made. So a plan when being processed has a set of constraints, and a set of substitutions which have been made in plans of which this one is a specialisation or extension. The full set of constraints for the plan are those which it has inherited (passed up from the recursion) together with its own constraints. These are then separated into various categories. The first of these consists of all the control flow constraints. These are dealt with separately, as described below. The second category is that of extra type restrictions on the roles. These are all of the form:

instance(<object or action>, <type>)

and are dealt with simply by choosing for each <object or action> mentioned the most restrictive of the type constraints available. The remaining constraints all represent data flows of one kind or another. The main steps of the processing of these data flow constraints are:

Step 1 Create "template" NAPEs for each non-recursive operation role in the plan. This effectively gives us a completely unconnected graph for the right hand side of the rule.

Step 2 Process any non-recursive constraints i.e. those not flowing to or from recursive roles of the plan. This "connects up" the graph created in step 1 so that it is consistent with the non-recursive constraints.

Step 3 If the plan has a recursive role, then:

(a) Work out the input and outputs of the graph created so far. Create a "template" for the recursive role, consistent with these inputs and outputs.

(b) Now process the remaining (i.e. recursive) constraints, if any. This connects the recursive role (if any) to the rest of the graph.

(c) Work out the inputs and outputs of the resulting graph. Remove the recursive role from the graph, and replace by a new one, compatible with these new inputs and outputs, and any substitutions that have been made. This graph forms the right hand side of the rule.

Step 4 Create the left hand side of the rule.

This will now be illustrated by a sequence of examples, each illustrating some new subtlety, since this is the easiest way to explain the process in detail. To start with we will consider quite a simple plan - **iterative-application**. Its compact definition is given below:

Temporal Plan **iterative-application**
    extension **single-recursion**
    roles .action(**@function**) .tail(**iterative-application**)
    constraints .action.op=.tail.action.op
        ∧ cflow(.action.out, .tail.action.in)

First the constraints are collected together. **Iterative-application** is an extension of **single-recursion**, which has no constraints, so the full set of constraint consists simply of those coming from the definition of **iterative-application** itself. These are divided into the categories mentioned above, and the data flow constraints are:

.action.op=.tail.action.op

Step 1. Since .action is defined to be an **@function**, and the definition of **@function** tells the system that an **@function** takes two inputs .op and .input, and produces a single output .output, a NAPE is created for this role:

[@fimction [.action.op .action.input][.action.output] .action.cf]

The recursive role is left for the moment.

Step 2. Now, the constraints not involving .tail are processed. In this case there aren't any, so the non-recursive part of the plan is considered finished.

Step 3 (a). Now the NAPEs so far are analysed to find the inputs and outputs to the graph. In this case these are:

INPUTS .action.op .action.input
OUTPUTS .action.output

A recursive role is now created, with the same inputs and outputs, but with names prefixed by .tail. This gives us:

```
[[@function [.action.op .action.input][.action.output] .action.cf]
 [recursive [.tail.action.op .tail.action.input][.tail.action.output] .tail.cf]]
```

for the complete graph so far.

Step 3 (b). Now the remaining data flow constraints are processed. We have one only:

```
.action.op=.tail.action.op
```

This is used to substitute .action.op for .tail.action.op throughout the graph, and the set of remaining constraints. In this case this results in:

```
[[@function [.action.op .action.input][.action.output] .action.cf]
 [recursive [.action.op .tail.action.input][.tail.action.output] .tail.cf]]
```

for the graph. The substitution made is stored in the frame for **iterative-application**, for use during the processing of other plans and overlays.

Step 3 (c). This almost gives us the right hand side of the rule, except that we need to create a new recursive role for it (with inputs and outputs that correspond to the inputs and outputs of the graph as a whole. The graph so far created is analysed to see what its inputs and outputs are. This is done by collecting all the inputs to all the NAPEs (apart from the recursive role) as a set of potential inputs to the graph. Similarly, a set of potential outputs is created. Note that the recursive role is not considered as contributing to these, since it is considered "internal" to the plan. The set of potential inputs is analysed to see which of them are not produced as outputs from any action (including the recursive role). These form the true inputs to the graph. Similarly the set of potential outputs is analysed to see which do not feed in as inputs to any action (including the recursive role). These form the set of true outputs. In this case this gives us:

```
INPUTS    .action.op .action.input
OUTPUTS .action.output
```

Now the recursive role is removed, and replaced by a new one with inputs and outputs created as follows (note in this case this will not change anything, but it will in later examples!):

For each input to the graph as a whole create an input to the recursive role, but prefixed with .tail. Then look up the table of substitutions (essentially in reverse) to find out what name is actually being used for this. If there is an entry in the table use that name, otherwise use the name as it is. The outputs are treated similarly.

Step 4. Finally the left hand side of the rule can be created. It essentially consists of a single NAPE with name equal to the name of the plan, and inputs and outputs those just computed (the true inputs and outputs). This gives us:

```
[iterative-application
        [.action.op .action.input][.action.output] cf] =>
        [[@function [.action.op .action.input][.action.output] .action.cf]
         [recursive [.action.op .tail.action.input][.tail.action.output] .tail.cf]]
```

in a similar notation to that used in Chapter 5. If another plan P is an extension or specialisation of **iterative-application**, then its full set of constraints, together with the substitutions used, will be passed up to P when required.

As a second example, consider the plan **iterative-search**. Its definition is:

Temporal Plan **iterative-search**
    specialization **iterative-termination-predicate**
        iterative-termination-output
    roles .exit(**cond**) .tail(**iterative-search+nil**)
    constraints .exit.if.input=.exit.end.succeed-input

As can be seen it is a *specialisation* of **iterative-termination-predicate** and **iterative-termination-output**. These two plans are processed in the same way as **iterative-application**, so when **iterative-search** is processed it receives the following constraints and substitutions from **iterative-termination-predicate** (we will miss out the control flow constraints in this discussion):

CONSTRAINTS	instance(**@predicate**, .exit.if) .exit.if.criterion=.tail.exit.if.criterion
SUBSTITUTIONS	exit.if.criterion FOR .tail.exit.if.criterion

and the following constraints and substitutions from **iterative-termination-output**:

CONSTRAINTS	instance(**join-output**,.exit.end)
	exit.end.fail-input=.tail.exit.end.output

SUBSTITUTIONS	exit.end.fail-input FOR .tail.exit.end.output

These substitutions are completely compatible with each other (i.e. nothing in one set of substitutions over-rides something in the other), so the union of them is simply formed. The constraints coming from the **iterative-search** itself are now added to these (they too are compatible), giving a final set of substitutions:

exit.if.criterion FOR .tail.exit.if.criterion
exit.end.fail-input FOR .tail.exit.end.output
.exit.if.input FOR .exit.end.succeed-input

and now the **iterative-search** plan can be processed. It has two roles - .exit, a **cond**, and the recursive role. However, **cond** is a plan in its own right, so its roles and constraints are fetched. Its roles are .if(**test**) and .end(**join**), and the constraints are all control flow constraints, whose processing will be briefly described later. So the true roles of **iterative-search** are .exit.if, .exit.end, and the recursive role. When we collect all the constraints together, we find that the .exit.end role is constrained to be a **join-output**, and the .exit.if role is an **@predicate**. So,

Step 1. The following two NAPEs are created:

```
[@predicate [.exit.if.criterion .exit.if.input]
                                  .exit.if.in .exit.if.succeed .exit.if.fail]
[join-output [.exit.end.succeed-input .exit.end.fail-input]
     [.exit.end.output] .exit.end.succeed .exit.end.fail.exit.end.out]
```

Step 2. The substitutions above are now separated into those involving the recursive role, and those that do not. The latter are substituted into the graph so far, giving:

```
[@predicate [.exit.if.criterion .exit.if.input]
                                  .exit.if.in .exit.if.succeed .exit.if.fail]
[join-output [.exit.if.input .exit.end.fail-input]
     [.exit.end.output] .exit.end.succeed .exit.end.fail.exit.end.out]
```

Step 3 (a). This graph is then processed to find its inputs and outputs. These are:

```
INPUTS    .exit.if.criterion .exit.if.input .exit.end.fail-input
OUTPUTS   .exit.end.output
```

A recursive role is now created for the graph, giving us:

```
[@predicate [.exit.if.criterion .exit.if.input]
                              .exit.if.in .exit.if.succeed .exit.if.fail]
[join-output [.exit.if.input .exit.end.fail-input]
   [.exit.end.output] .exit.end.succeed .exit.end.fail .exit.end.out]
[recursive  [tail.exit.if.criterion
                            .tail.exit.if.input .tail.exit.end.fail-input]
       [.tail.exit.end.output] .tail.cf]
```

Step 3 (b). The remaining substitutions are now made into this:

```
[@predicate [.exit.if.criterion .exit.if.input]
                              .exit.if.in .exit.if.succeed .exit.if.fail]
[join-output [.exit.if.input .exit.end.fail-input]
   [.exit.end.output] .exit.end.succeed .exit.end.fail .exit.end.out]
[recursive [exit.if.criterion .tail.exit.if.input .tail.exit.end.fail-input]
       [exit.end.fail-input] .tail.cf]
```

Step 3 (c). The true inputs and outputs to this are:

```
INPUTS    .exit.if.criterion .exit.if.input
OUTPUTS   .exit.end.output
```

A new recursive role is now created:

```
[recursive [.tail.exit.if.criterion .tail.exit.if.input]
       [.tail.exit.end.output] .tail.cf]
```

and, by looking up the table of substitutions, we see that for instance .tail.exit.if.criterion has been replaced by ..exit.if.criterion. So we make this replacement in the new recursive role, and similarly for its other inputs and outputs. This finally gives us (after **Step 4**):

```
[iterative-search [.exit.if.criterion .exit.if.input][.exit.end.output] cf] =>
        [@predicate [.exit.if.criterion .exit.if.input]
                                      .exit.if.in .exit.if.succeed .exit.if.fail]
        [join-output [.exit.if.input .exit.end.fail-input]
             [.exit.end.output] .exit.end.succeed .exit.end.fail .exit.end.out]
        [recursive [exit.if.criterion .tail.exit.if.input]
                    [exit.end.fail-input] .tail.cf]
```

for the entire rule.

As a third example, consider the plan **trailing-search**:

```
Temporal Plan trailing-search
        extension iterative-search trailing
        roles .current(object) .previous(object) .exit(cond)
              .tail(trailing-search)
        constraints instance(join-two-outputs, .exit.end)
                    ∧ .current=.exit.if.input
                    ∧ .previous=.exit.end.succeed-input-two
                    ∧ .tail.exit.end.output-two=.exit.end.fail-input-two
```

It inherits constraints from both **iterative-search** (just discussed) and **trailing**. From **iterative-search** it inherits the substitutions just discussed, and from **trailing** it inherits (Figures 7.18 and 7.19 show the way constraints and substitutions are passed up the hierarchy):

.current FOR .tail.previous

These are compatible with each other, but care must be taken when these are combined with the substitutions implied by the definition of **trailing-search** itself:

.current FOR .exit.if.input

.previous FOR .exit.end.succeed-input-two

.tail.exit.end.output-two FOR .exit.end.fail-input-two

since there is a conflict with those from **iterative-search**:

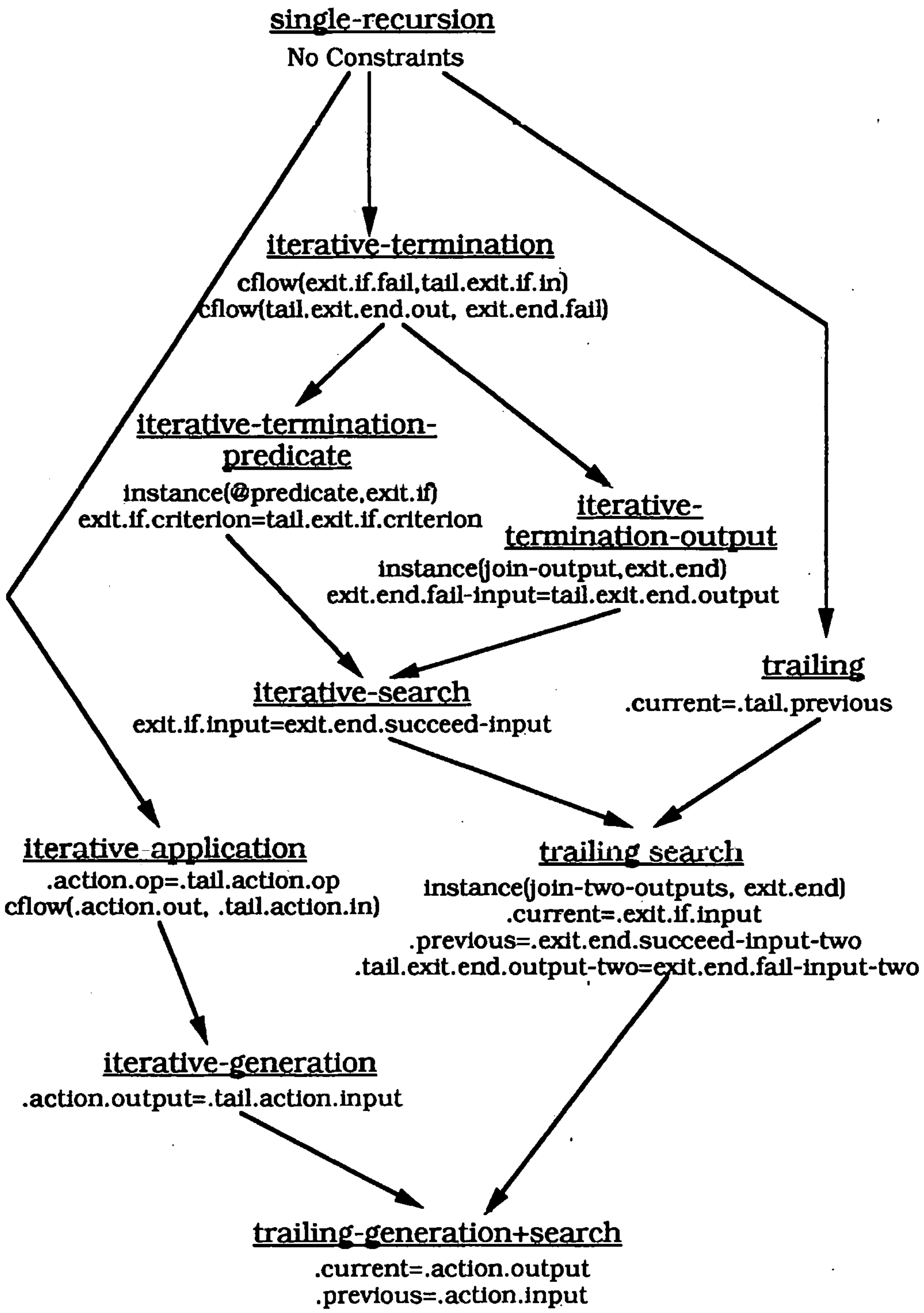

single-recursion
No Constraints
iterative-termination
cflow(exit.if.fail,tail.exit.if.in)
cflow(tail.exit.end.out, exit.end.fail)
iterative-termination-predicate
instance(@predicate,exit.if)
exit.if.criterion=tail.exit.if.criterion
iterative-termination-output
instance(join-output,exit.end)
exit.end.fail-input=tail.exit.end.output
trailing
.current=.tail.previous
iterative-search
exit.if.input=exit.end.succeed-input
iterative-application
.action.op=.tail.action.op
cflow(.action.out, .tail.action.in)
trailing search
instance(join-two-outputs, exit.end)
.current=.exit.if.input
.previous=.exit.end.succeed-input-two
.tail.exit.end.output-two=exit.end.fail-input-two
iterative-generation
.action.output=.tail.action.input
trailing-generation+search
.current=.action.output
.previous=.action.input

Figure 7.18
Constraint Heirarchy for
Trailing-Generation+Search

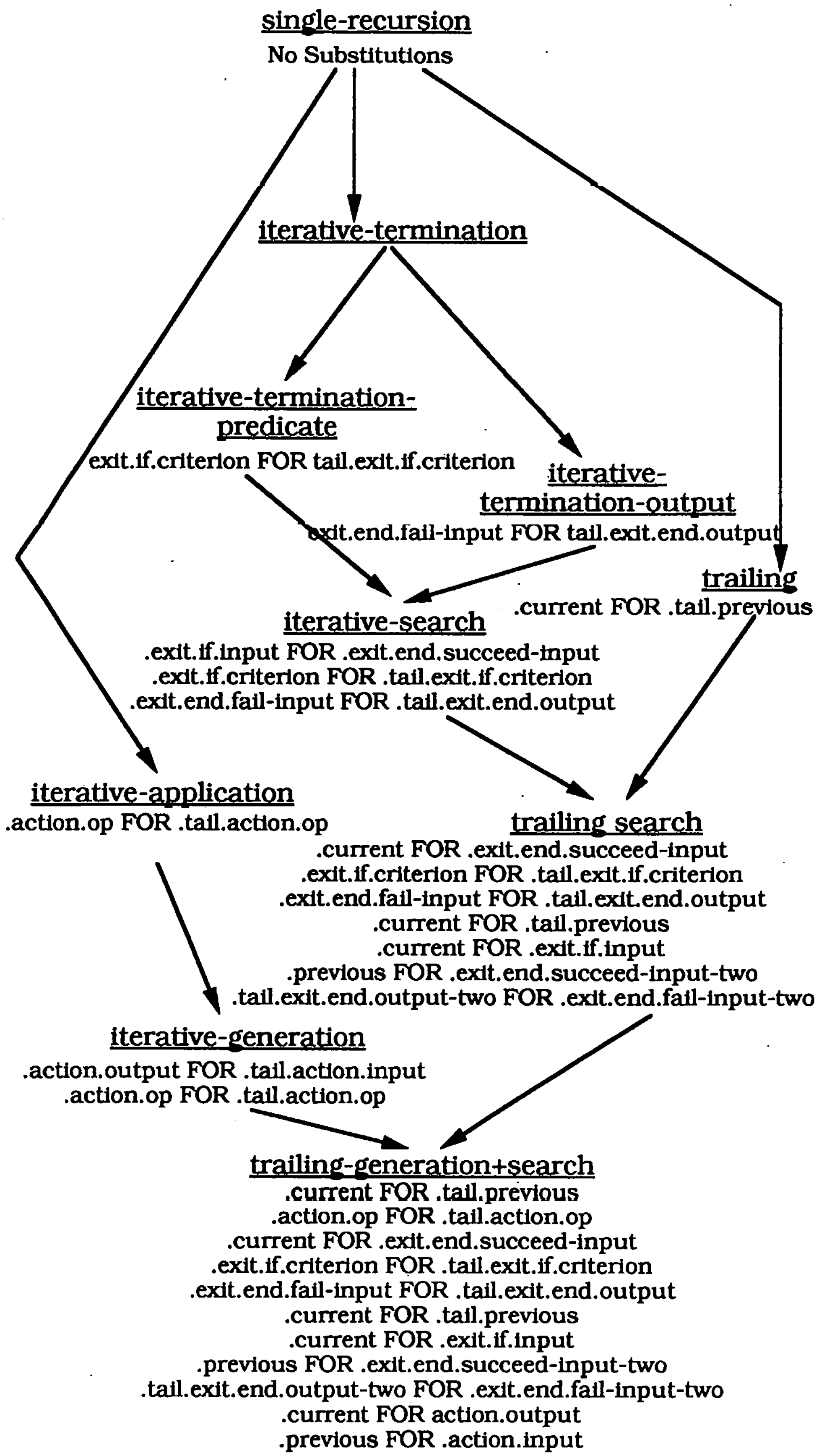

single-recursion
No Substitutions
iterative-termination
iterative-termination-predicate
exit.if.criterion FOR tail.exit.if.criterion
iterative-termination-output
exit.end.fail-input FOR tail.exit.end.output
trailing
.current FOR .tail.previous
iterative-search
.exit.if.input FOR .exit.end.succeed-input
.exit.if.criterion FOR .tail.exit.if.criterion
.exit.end.fail-input FOR .tail.exit.end.output
iterative-application
.action.op FOR .tail.action.op
trailing search
.current FOR .exit.end.succeed-input
.exit.if.criterion FOR .tail.exit.if.criterion
.exit.end.fail-input FOR .tail.exit.end.output
.current FOR .tail.previous
.current FOR .exit.if.input
.previous FOR .exit.end.succeed-input-two
.tail.exit.end.output-two FOR .exit.end.fail-input-two
iterative-generation
.action.output FOR .tail.action.input
.action.op FOR .tail.action.op
trailing-generation+search
.current FOR .tail.previous
.action.op FOR .tail.action.op
.current FOR .exit.end.succeed-input
.exit.if.criterion FOR .tail.exit.if.criterion
.exit.end.fail-input FOR .tail.exit.end.output
.current FOR .tail.previous
.current FOR .exit.if.input
.previous FOR .exit.end.succeed-input-two
.tail.exit.end.output-two FOR .exit.end.fail-input-two
.current FOR action.output
.previous FOR .action.input

Figure 7.19
Substitution Heirarchy

exit.if.criterion FOR .tail.exit.if.criterion
exit.end.fail-input FOR .tail.exit.end.output
.exit.if.input FOR .exit.end.succeed-input

since they could over-ride the use of exit.if.input. These substitutions must be reconciled, which involves choosing one of the conflicting substitutions, and replacing it by the other. In this case this gives us as the full set of substitutions:

.current FOR .tail.previous
.current FOR .exit.if.input
.previous FOR .exit.end.succeed-input-two
.tail.exit.end.output-two FOR .exit.end.fail-input-two
exit.if.criterion FOR .tail.exit.if.criterion
exit.end.fail-input FOR .tail.exit.end.output
.current FOR .exit.end.succeed-input

Now the graph for the right hand side of the rule can be created. From the full set of constraints passed up, and the roles of the plan itself, we see that the NAPEs in the plan are .exit.if(**@predicate**), .exit.end(**join-two-outputs**), and a recursive role.

Step 1. As before, the non-recursive NAPEs are processed first, giving:

```
[@predicate [.exit.if.criterion .exit.if.input]
                              .exit.if.in .exit.if.fail .exit.if.succeed]]
[join-two-outputs
      [.exit.end.succeed-input .exit.end.fail-input
              .exit.end.succeed-input-two .exit.end.fail-input-two]
      [.exit.end.output .exit.end.output-two]
       .exit.end.succeed .exit.end.fail.exit.end.out]
```

Step 2. As before, the substitutions not involving .tail are made first, giving:

```
[@predicate [.exit.if.criterion .current]
                                   .exit.if.in .exit.if.fail .exit.if.succeed]]
[join-two-outputs
     [.current .exit.end.fail-input
           .previous .exit.end.fail-input-two]
     [.exit.end.output .exit.end.output-two]
      .exit.end.succeed .exit.end.fail.exit.end.out]
```

Step 3 (a). This is now processed, to yield a recursive role. This results in:

```
[recursive
     [tail.exit.if.criterion .tail.current .tail.exit.end.fail-input
           .tail.previous .tail.exit.end.fail-input-two]
     [.tail.exit.end.output .tail.exit.end.output-two] .tail.cf]
```

Step 3 (b). The remaining substitutions are made into the resulting graph, giving:

```
[@predicate [.exit.if.criterion .current]
                                   .exit.if.in .exit.if.fail .exit.if.succeed]]
[join-two-outputs
     [.current .exit.end.fail-input
           .previous .tail.exit.end.output-two]
     [.exit.end.output .exit.end.output-two]
      .exit.end.succeed .exit.end.fail.exit.end.out]
[recursive
     [exit.if.criterion .tail.current .tail.exit.end.fail-input
           .current .tail.exit.end.fail-input-two]
     [exit.end.fail-input .tail.exit.end.output-two] .tail.cf]
```

Step 3 (c). The true inputs and outputs for this are computed as:

```
INPUTS    .exit.if.criterion .current .previous
OUTPUTS  .exit.end.output .exit.end.output-two
```

Use these to create the new recursive role.

Step 4. Finally we get:

```
[trailing-search [.exit.if.criterion .current .previous]
                 [.exit.end.output .exit.end.output-two] cf] =>

    [@predicate [.exit.if.criterion .current]
                                  .exit.if.in .exit.if.fail .exit.if.succeed]]
    [join-two-outputs
        [.current .exit.end.fail-input
            .previous .tail.exit.end.output-two]
        [.exit.end.output .exit.end.output-two]
         .exit.end.succeed .exit.end.fail .exit.end.out]
    [recursive
        [exit.if.criterion .tail.current .current]
        [exit.end.fail-input .tail.exit.end.output-two] .tail.cf]
```

for the entire rule.

Now consider the **trailing-generation+search** plan (discussed in Chapter 3):

Temporal Plan **trailing-generation+search**

extension **iterative-generation trailing-search**

roles .current(**object**) .previous(**object**) .exit(**cond**)

.action(**@function**) .tail(**trailing-generation+search**)

constraints .current=.action.output ∧ .previous=.action.input

It inherits the following substitutions from **trailing-search**:

.current FOR .tail.previous

.current FOR .exit.if.input

.previous FOR .exit.end.succeed-input-two

.tail.exit.end.output-two FOR .exit.end.fail-input-two

exit.if.criterion FOR .tail.exit.if.criterion

exit.end.fail-input FOR .tail.exit.end.output

.current FOR .exit.end.succeed-input

and the following from **iterative-generation** (processed as described above):

.action.output FOR .tail.action.input

.action.op FOR .tail.action.op

These are completely compatible, so are simply added into the full set of substitutions for the plan. The substitutions implied by the definition of **trailing-generation+search** itself are:

.current FOR .action.output

.previous FOR .action.input

These are then reconciled with the above substitutions, yielding:

.current FOR .tail.previous

.current FOR .exit.if.input

.previous FOR .exit.end.succeed-input-two

.tail.exit.end.output-two FOR .exit.end.fail-input-two

.exit.if.criterion FOR .tail.exit.if.criterion

.exit.end.fail-input FOR .tail.exit.end.output

.current FOR .exit.end.succeed-input

.current FOR .tail.action.input..(*)

.action.op FOR .tail.action.op

.current FOR .action.output

.previous FOR .action.input

However, this is not quite right. The substitution indicated (*) needs to be altered. This is because of the way in which we process substitutions not involving .tail first, which means that when we make the substitution .previous FOR .action.input in the graph, we get as input to the @function NAPE .previous (instead of .action.input). This means that when we create the recursive role we end up giving it an input of

.tail.previous, instead of .tail.action.input which is referred to in the other substitutions. So, when we make the first substitutions (the ones not involving .tail), we must also adjust the remaining substitutions so that if we make a substitution of the form A FOR B we must replace any occurrence of .tail.B in the remaining substitutions by .tail.A. In this case this will give us .current FOR .tail.previous, which happens to be one of the substitutions we already have for this plan, but in many cases it won't be.

So, processing the rule as before, but taking note of the above subtlety, yields the rule:

```
[trailing-generation+search [.exit.if.criterion .action.op .previous]
                        [.exit.end.output .exit.end.output-two] cf] =>

    [@predicate [.exit.if.criterion .current]
                .exit.if.in .exit.if.succeed .exit.if.fail]
    [@function [.action.op .previous][.current] .action.cf]
    [join-two-outputs
        [.current .exit.end.fail-input
            .previous .tail.exit.end.output-two]
        [.exit.end.output .exit.end.output-two]
        .exit.end.succeed .exit.end.fail .exit.end.out]]
    [recursive [exit.if.criterion .action.op .current ]
               [.exit.end.fail-input .tail.exit.end.output-two] .tail.cf]
```

which a little thought will show corresponds to Figure 3.4.

It should of course be stressed that the process for plans without recursive roles is exactly the same as this, without all the "messing about" with the recursive roles.

Control flow constraints are used to create the plan condition, as described in Chapter 4. The variables for each control environment are those indicated on the right hand side of the above rules. The control flow constraints passed up the inheritance hierarchy (in exactly the same fashion as the data flow constraints), together with any coming from roles which are *plans* (e.g. **cond**), are grouped together and processed as indicated in Chapter 4 to form the plan condition, which is then substituted for the variable cf on the left hand side of the rule.

Before going on to describe the processing of temporal overlays, we will briefly describe how plans containing roles which are data *plans* (rather than simple objects) are processed. Consider the plan:

TemporalPlan **internal-labelled-thread-add**
Roles .old(**labelled-thread**) .add(**internal-thread-add**) .update(**newarg**)
.new(**labelled-thread**)
Constraints .old.spine=.add.old ∧ .old.label=.update.old
∧ .add.input=.update.arg ∧ .add.new=.new.spine
∧ .update.new=.new.label

This is the plan shown in Figure 7.20, which essentially adds a new node into a thread, and updates the labelling function on the new node. It is the abstraction of adding a new element into a list implemented by linked objects of some type or other. When the rule is processed the definitions of the data plans are accessed, and used to create names for the roles of the data plans themselves. In this case this gives us .old.spine and .old.label for the .old role, and .new.spine and .new.label for the .new role. Assertions are stored with the rule stating:

.old=labelled-thread(.old.spine, .old.label)
.new=labelled-thread(.new.spine, .new.label)

Further more, this is done recursively. So, since .new.spine and .old.spine are both threads (this information is obtained from the definition of a labelled thread) we also add assertions:

.old.spine=thread(.old.spine.nodes, .old.spine.edge)
.new.spine=thread(.new.spine.nodes, .new.spine.edge)

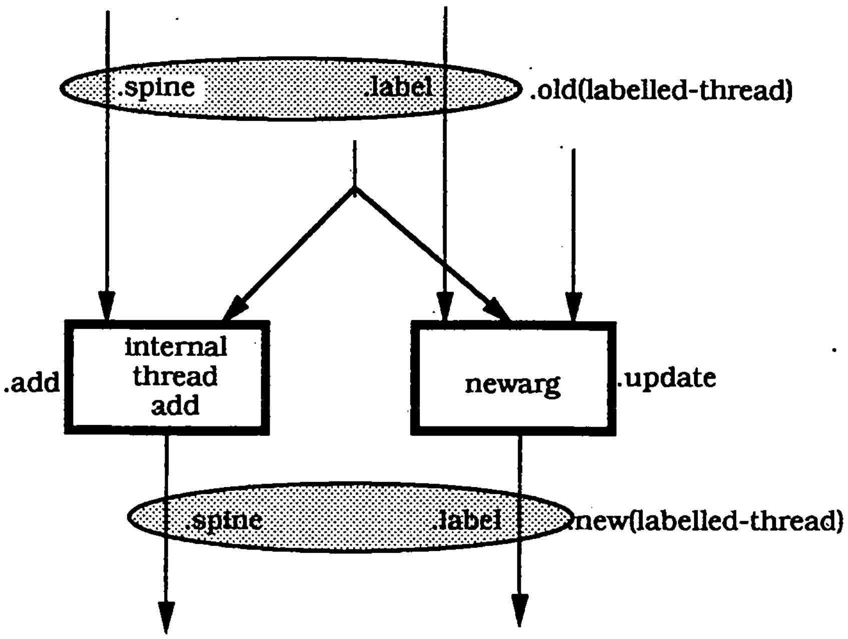


Figure 7.20
Internal Labelled Thread Add

since we cannot know how deeply into this structure the rule (or overlays of the plan) may need to go. It also helps in connecting the plan up to other plans as the parsing proceeds. Now the rule is processed exactly as before, but with the additional step that any substitutions that are made in the NAPEs of the rule are also made into these assertions. This results in the rule:

```
[internal-labelled-thread-add [.old.spine .update.arg
                                          .old.label .update.input]
                              [.add.new .update.new] cf] =>
    [internal-thread-add [.old.spine .update.arg][.add.new] .add.cf]
    [newarg [.old.label .update.arg .update.input]
          [.update.new] .update.cf]
```

with data plan assertions:

```
    .old=labelled-thread(.old.spine, .old.label)
    .new=labelled-thread(.add.new, .update.new)
    .old.spine=thread(.old.spine.nodes, .old.spine.edge)
    .add.new=thread(.new.spine.nodes, .new.spine.edge)
```

These assertions (or at least copies of them) are included in each partial patch for this plan. As the tie-points in the patch get instantiated (when the patch is extended) the instantiations are propagated into the assertions, enabling the proper creation of, or matching against, assertions in the data plan database as the parsing proceeds (as described in 7.1). So these assertions must be considered part of the rule.

7.8.1.2 Translating Temporal Overlays into Rule Form

Now we are ready to discuss the conversion of temporal overlays. Consider the overlay:

Temporal Overlay **trailing-generation+search->find** :
trailing-generation+search -> **internal-thread-find**
correspondences
generator->digraph(temporal-iterator(trailing-generation+search))=
internal-thread-find.universe
∧ **trailing-generation+search**.exit.if.criterion=
internal-thread-find.criterion
∧ **trailing-generation+search**.exit.end.output=
internal-thread-find.output
∧ **trailing-generation+search**.exit.end.two=
internal-thread-find.previous
∧ **trailing-generation+search**.action.in=**internal-thread-find**.in
∧ **trailing-generation+search**.exit.out=**internal-thread-find**.out

The rule corresponding to a temporal overlay **A->B** is expressed in the form:

NAPE for A => NAPE for B

So the left hand side of this rule is created essentially by copying the left hand side of the rule for the plan **trailing-generation+search**, giving us:

[trailing-generation+search [.exit.if.criterion .action.op .previous]
[.exit.end.output .exit.end.output-two] cf]

The right hand side is created initially by instantiating a NAPE for the appropriate action, based on the *IOSpec* definition for the action. This gives us:

[internal-thread-find [.universe .criterion][.output .previous] cf]

Now the correspondences are processed, and used to define a set of substitutions. Most of these are straightforward and give us the following substitutions (we can ignore those to do with input and output situations as these are handled by the plan conditions etc., as described in Chapter 4):

exit.if.criterion FOR .criterion
.exit.end.output FOR .output
.exit.end.two FOR .previous

However, the first needs more processing. It states:

generator->digraph(temporal-iterator(trailing-generation+search))=
internal-thread-find.universe

Such nested overlays are processed recursively. The innermost one is:

temporal-iterator(trailing-generation+search)

where the **temporal-iterator** overlay is defined by:

Temporal Overlay **temporal-iterator** : **iterative-generation -> iterator**
correspondences **iterative-generation**.action.input=**iterator**.seed
∧ **function->binrel(iterative-generation**.action.op)=**iterator**.op

This is used to create assertions about the data objects involved. It starts by making an assertion:

tnew=iterator(tnew.seed, tnew.op)

where tnew is a new name. Now the substitutions stored with each plan come into effect. From the definition of **temporal-iterator** it needs to know which name in the **trailing-generation+search** plan (viewed as an **iterative-generation** of which it is an *extension*) corresponds to .action.input. Looking up the table of substitutions it finds that .previous was used for this purpose. So it can substitute .previous into the above assertion, giving:

tnew=iterator(.previous, tnew.op)

Now it tries to find a substitution for tnew.op. This results in another assertion:

tnew.op=**function->binrel**(.action.op)

and again it looks in the table of substitutions to find that .action.op is the name used. It has now finished with the innermost overlay, so these assertions are passed back out, and it now makes an assertion:

.universe=**generator->digraph**(tnew)

So the overlay is finally represented by the rule:

```
[trailing-generation+search [.exit.if.criterion .action.op .previous]
                            [.exit.end.output .exit.end.output-two] cf] =>
[internal-thread-find [.universe .exit.if.criterion]
                      [.exit.end.output .exit.end.two] cf]
```

together with the assertions:

tnew=iterator(.previous, tnew.op)

tnew.op=function->binrel(.action.op)

.universe=generator->digraph(tnew)

These are used during the parsing process as described in 7.1 to add assertions to the data plan and data overlay databases.

One more point needs to be made. The variables (names of tie-points) in a plan are made into a table which is included in the patch information when an empty patch corresponding to a plan is created. This table is used to store information about which actual tie-point each name gets instantiated to during the parsing process. Then, when an overlay is made of a plan instance, this table can be consulted "by the overlay", in order to work out how to instantiate its own tie-points.

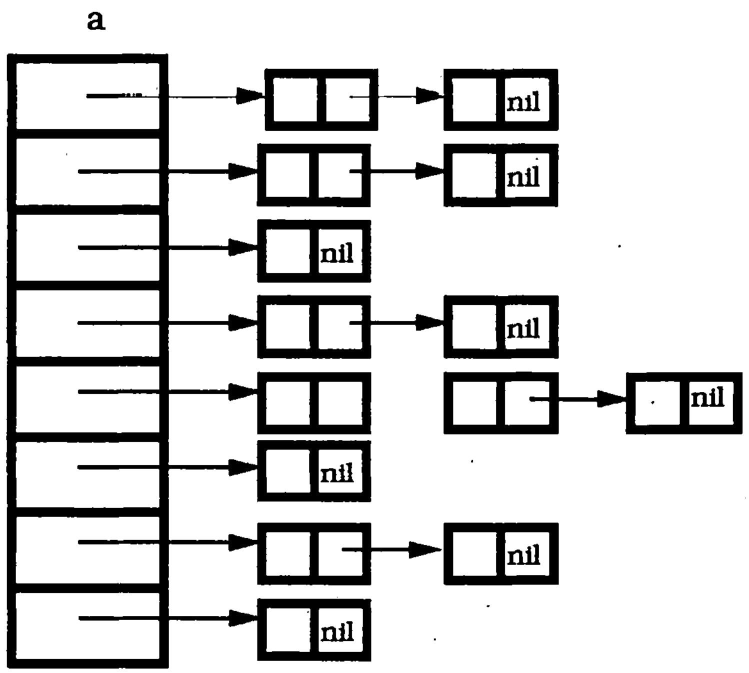


Figure 7.21
An Array of Sets(implemented as linked records)

Without the mapping from names to instantiated tie-points, the overlay mechanism could not work.

7.8.2 Additional Plans Needed to Cope with User Defined Data Types

It has been mentioned several times that user defined data types will be dealt with by means of overlays such as **thread->list** etc. However there is a problem that arises as a result of the way we have treated data overlays of tie-points. Consider the case of an array, where the entries in the entries in the array are lists implemented by means of linked records. The situation is shown in Figure 7.21. Now the proper analysis of this may be that we have a sequence (the array) of sets (implemented as lists). Hash tables are an example where this kind of analysis is necessary in order to recognise that hash table lookup is sometimes used simply as a membership test. So suppose we have the following declarations:

```
type listelement = record
                        numb : integer;
                        next : ^listelement;
                    end;
        plist = ^listelement;
var a: array[1..30] of plist;
     n :integer;
     p, q : plist;
```

and suppose that somewhere in the program we have the following piece of code:

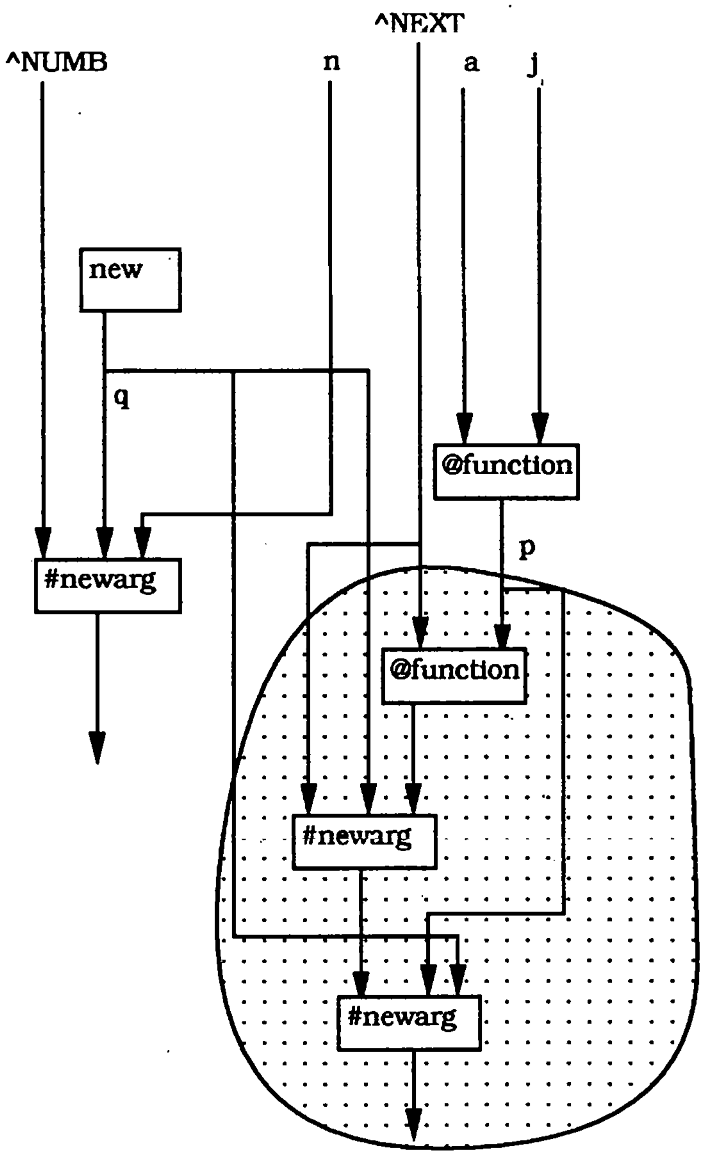


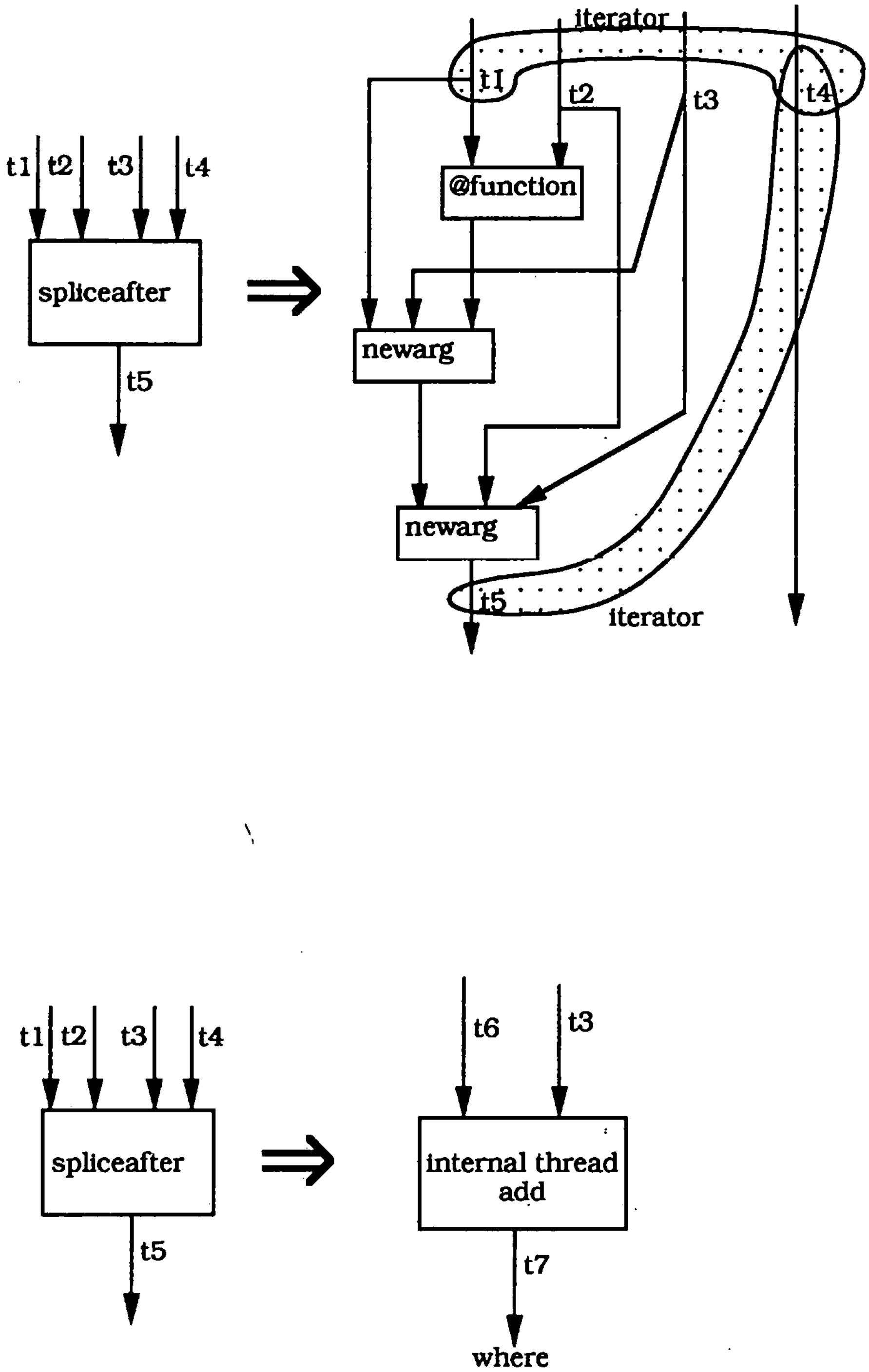
Figure 7.22
Surface Plan for Code which Adds
New Element Into List a[j]

```
.
.
.
new(q);
q^.numb:=n;
p:=a[j];
q^.next:=p^.next;
p^.next:=q;
.
.
.
```

which splices a new element into the list a[j]. The surface plan[3] for this is shown in Figure 7.22. Before going on to describe the recognition process it is important to note that from now on, when we are describing the recognition process we will usually talk about it as resulting in a graph of some sort. This is not strictly accurate since, of course, what has really happened is that some patches have been added to the chart. However, it is not feasible to show all the patches, so we simply show the ones that are relevant in understanding the main steps whereby something is recognised. It should always be understood that in fact there will be lots of other patches in the chart apart from the ones we show, and lots added between the steps we show.

Now suppose the parser goes to work on this, and it recognises this as involving an **#set-add** operation. To do this it has to first recognise the shaded part of the surface plan as a **#spliceafter** plan, shown in Figure 7.23, then use the overlay **spliceafter->internal-thread-add**, also shown in Figure 7.23. This results in Figure 7.24. It then has to recognise the shaded portion of this as forming an

[3] Note that we have already grouped the "^" function and NUMB together as a composed function object for clarity. This would have been done via the composed-functions->function overlay in the library.



where
t6=iterator->thread(iterator(t2,t4))
and
t7=iterator->thread(iterator(t5,t4))

Figure 7.23
Spliceafter and
Overlay Splideafter->Internal-Thread-Add

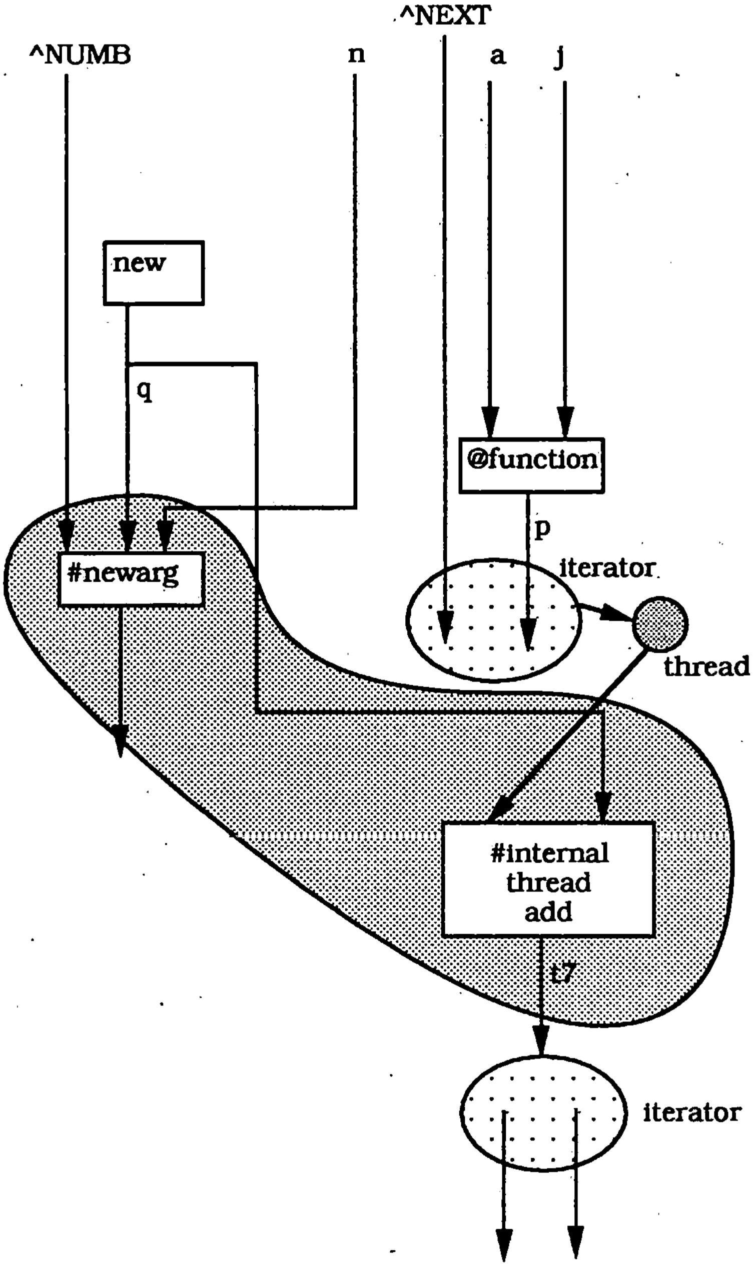


Figure 7.24
After Internal Thread Add

internal-labelled-thread-add, resulting in Figure 7.25, and then uses another overlay **internal-labelled-thread-add->set-add** to give Figure 7.26. This is all fine, but we now have a problem. The **set** has been recognised via a sequence of overlays, first of all grouping the tie-point representing p with the tie-point representing NEXT together as an **iterator** and this is represented by a <u>new</u> tie-point as described above. Another new tie-point is then created representing this **iterator** viewed as a **thread**, and this new tie-point is the input to the **spliceafter** operation. This **thread** was then grouped with the tie-point for NUMB, to form an instance of the data plan **labelled-thread**, represented by yet another new tie-point, which is the input to the **internal-labelled-thread-add** operation. Finally, using the **labelled-thread->set overlay** applied to the **labelled-thread** tie-point gives us yet another new tie-point representing the **set**, and the **set** (and its updated version) have become rather detached from the function which was supposed to have produced it in the first place. This would make it rather hard for a data-flow based parser to recognise plans involving functions with sets as values (where the sets are implemented via some other data structure), unless the parser were to spend a lot of its time computing something a bit like the "transitive closure" of the assertions in the data plan and data overlay databases. However, it turns out that to deal with this we do need to change the recogniser as such. We do, however, need to add a variety of extra plans and overlays not discussed in Rich[1981] to the plan library. Some of these are given below:

DataPlan **composite-functions**
Roles .op1(**function**) .op2(**function**)
Constraints domain-type(.op2)=range-type(.op1)
        ∧ range-type(.op2)=domain-type(.op2)

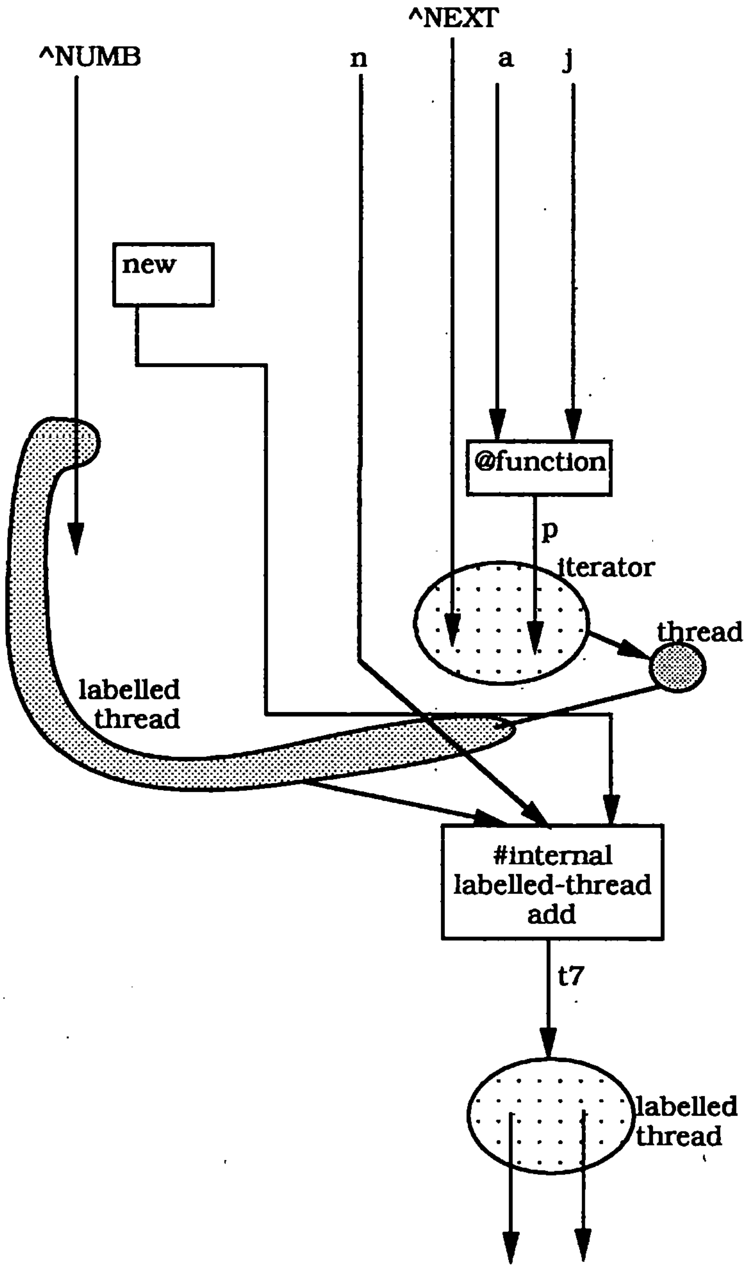


Figure 7.25
After Internal Labelled Thread Add

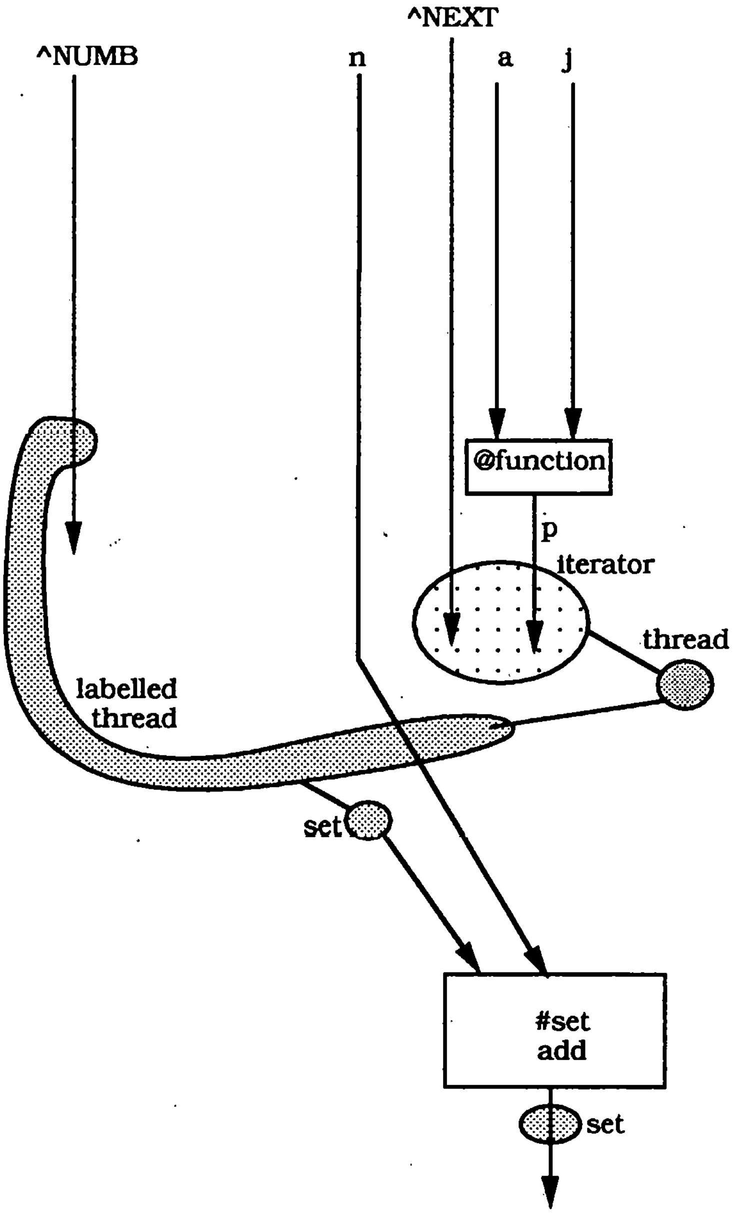


Figure 7.26
After Set Add

DataOverlay **composite-functions->function**
Definition F=**composite-functions->function**(p,s) ≡
[instance(function,F)
∧ [[apply(**function**(F,s),x)=y]
↔ [**iterator**(y,s) ≠ *undefined*
∧ **iterator**(y,s).seed=apply(**composite-functions**(p,s).op1,x)
∧ **iterator**(y,s).op=**function->binrel**(**composite-functions**(p,s).op2,s)]

TemporalPlan **@function-composite**
Roles .action(**@function**) .comp(**composite-functions**) .gen(**iterator**)
Constraints .action.op=.comp.op1
∧ .gen.seed=.action.output
∧ .gen.op=**function->binrel**(.comp.op2)

TemporalOverlay **@function-composite->@function-to-iterator**
Correspondences
@function-to-iterator.op=
composite-functions->function(**@function-composite**.comp)
∧ **@function-to-iterator**.input=**@function-composite**.action.input
∧ **@function-to-iterator**.output=**@function-composite**.gen

These first four plans and overlays are what are required to recognise functions which give **iterators** as their result (where the functions are implemented as a function which returns the seed of the iterator). The **composite-function** object involved essentially groups the iterating function of the **iterator** together with the function).

DataPlan **composite-functions+label**
Roles .comp(**composite-functions**) .label(**function**)
Constraints domain-type(.label)=range-type(.comp.op1)

```
TemporalPlan @function-composite+label
Extension @function-composite
Roles
    .action(@function-composite) .comp+lab(composite-function+label)
    .labt(labelled-thread)
Constraints  .labt.spine=iterator->thread(.action.gen)
           ∧ .labt.label=.comp+lab.label
           ∧ .action.comp=.comp+lab.comp
```

```
DataOverlay composite-functions+label->function
Definition F=composite-functions+label->function(p,s) ≡
 [instance(function,F)
∧ [[apply(function(F,s),x)=y]
  ↔ labelled-thread(y,s) ≠ undefined
  ∧ labelled-thread(y,s).spine=
                    iterator->thread(apply(function(p.comp.op1,s),x),s)
  ∧ labelled-thread(y,s).label=composite-functions+label.label
```

```
TemporalOverlay @function-composite+label->
                                        @function-to-labelled-thread
Correspondences
 @function-to-labelled-thread.op=
  composite-functions+label->function(
                               @function-composite+label.comp+lab)
∧ @function-to-labelled-thread.input=
                         @function-composite+label.action.action.input
∧ @function-to-labelled-thread.output=@function-composite+label.labt
```

These plans are what is needed in order to deal with functions which producc **labelled-threads** (an abstraction of lists built out of linked records).

In addition for each data-type **B** we need operations of the general form:

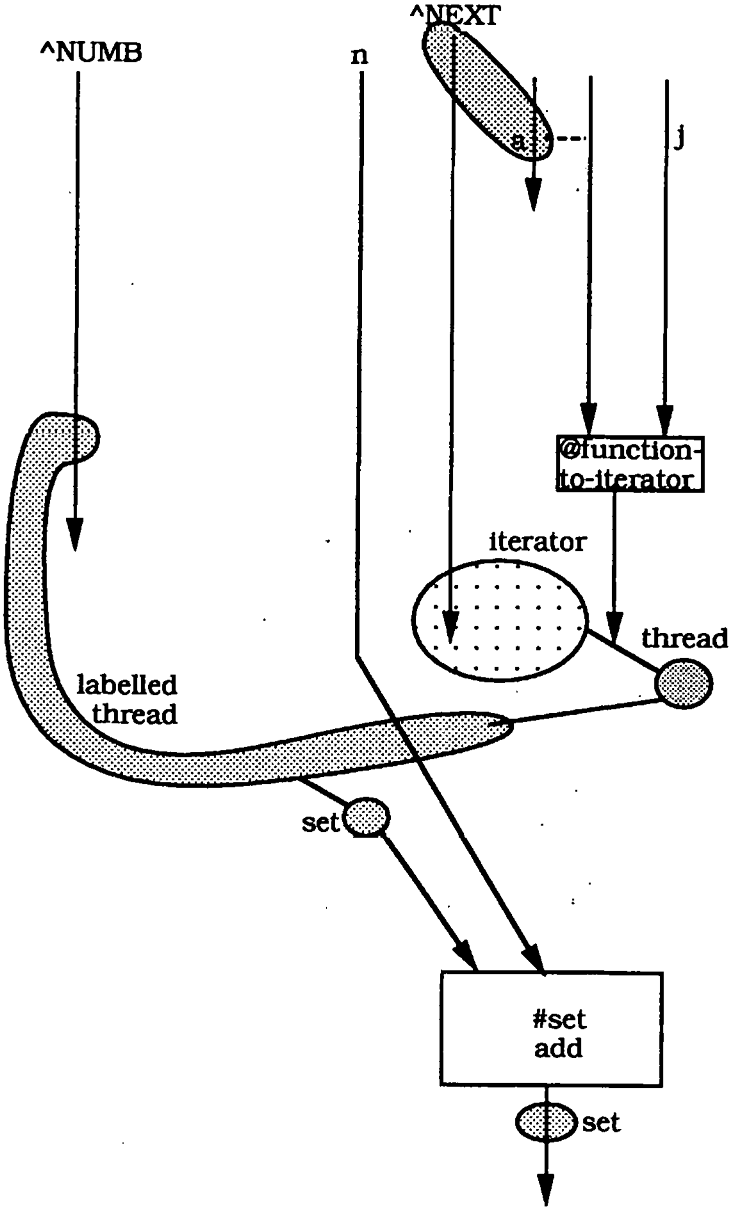


Figure 7.27
After @function-to-iterator recognised

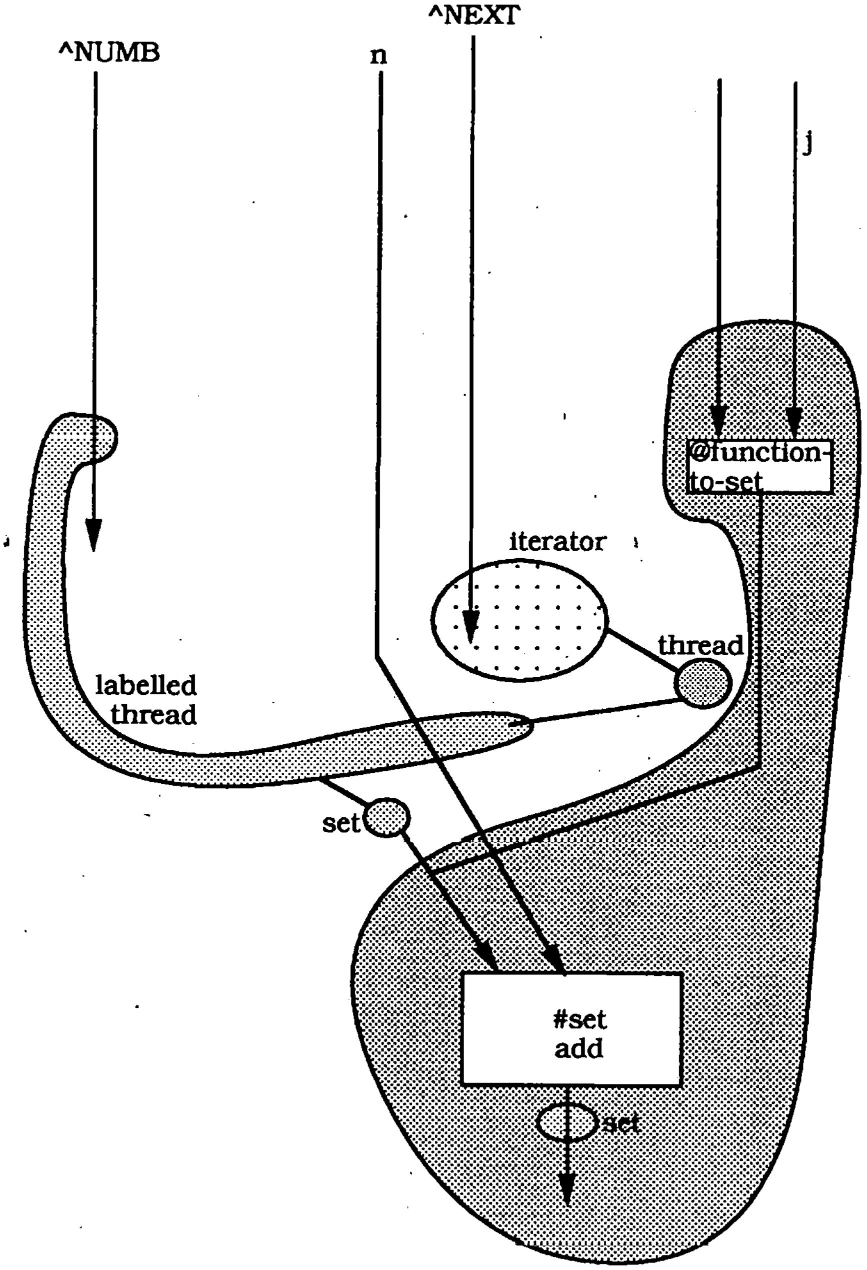


Figure 7.28
After @function-to-set recognised

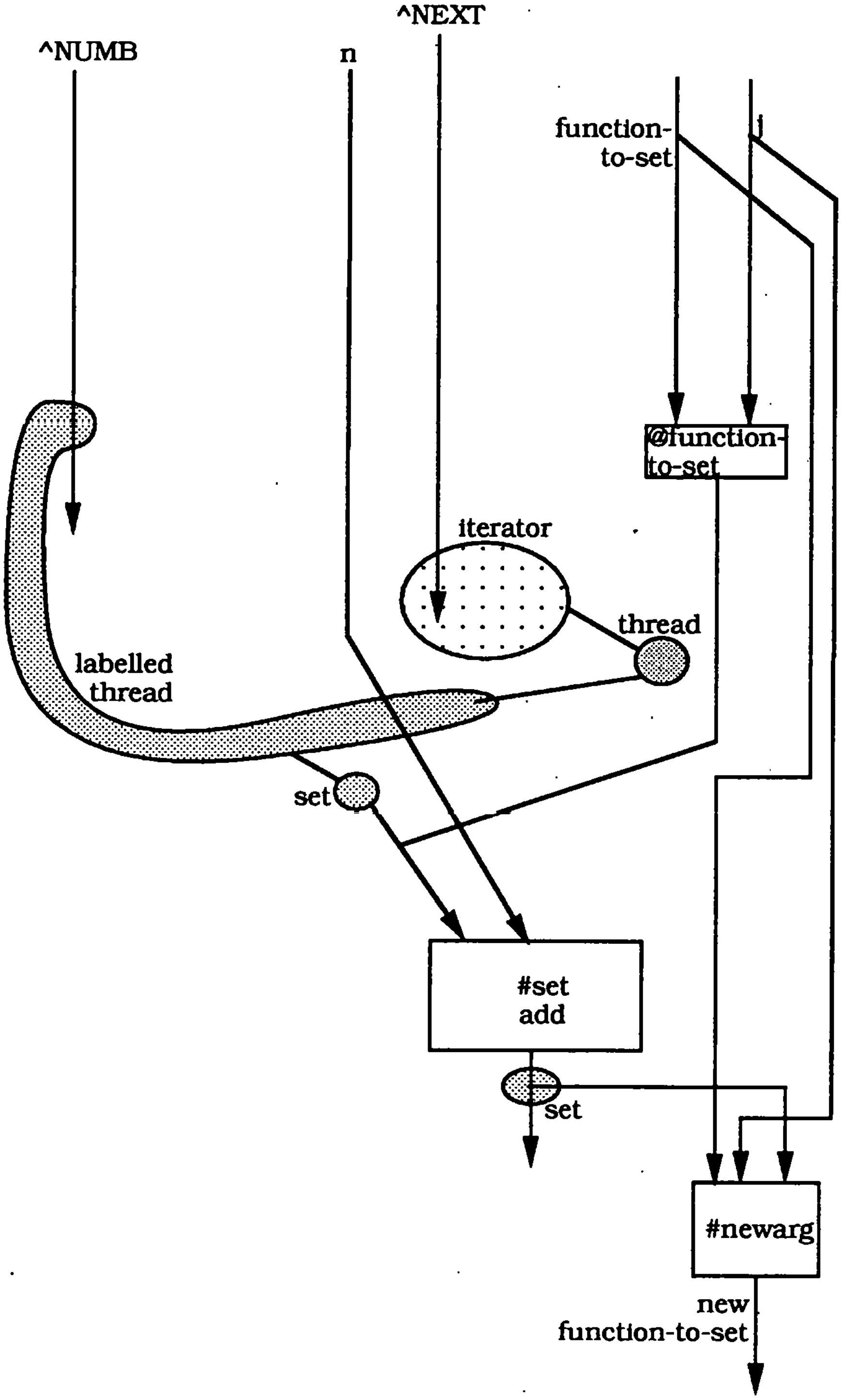


Figure 7.29
After #newarg of function-to-set recognised

IOSpec **@function-to-B**
Specialisation **@function**
Constraints range-type(.op)=B

The operations **@function-to-iterator** and **@function-to-labelled-thread,** used above, are specific examples of this general form. Finally, for each data-overlay of the form **B->C** we need an overlay of the general form:

TemporalOverlay **@function-to-B->@function-to-C**
Correspondences **@function-to-C**.op=**@function->B**.op
∧ **@function-to-C**.input=**@function->B**.input
∧ **@function-to-C**.output=**B->C**(**@function->B**.output)

Combining these with plans and overlays of the general form:

TemporalPlan **@function+#action**
Roles .action(**@function**) .impure(**#action**)
Constraints .action.output=.impure.input

and

TemporalOverlay **@function+#action->#newarg**
Correspondences **#newarg**.op=**@function+#action**.action.op
#newarg.arg=**@function+#action**.action.output
#newarg.value=**@function+#action**.impure.output

gives us all the machinery we need in order to make sure that:

a) plans connect up properly even after recognising data abstractions, and

b) we realise that some views of functions change when we side effect the structures they compute.

This process is illustrated for the above example in Figures 7.27-7.29. It should again be noted that the reason we need all of this is because we are dealing with graphs which do not distinguish properly between the behaviours and identities of objects.

Chapter 8.

Program Understanding in Practice

Before showing how a combination of plan recognition and general reasoning on plan diagrams can enable IDS to find type 1 (and type 3) errors in programs it is instructive to see plan recognition in practice. This has previously been described in a variety of papers [Rich 1981, Wills 1986 and 1990, Waters 1978, Schrobe, Waters, and Sussman 1979], but the process is not as entirely straightforward as their descriptions would suggest. Many of the difficulties arise from the fact that the programs considered here have not been written using KBEMacs [Waters 1982] so that the particularly close match between the programs and the plan library that Rich and Waters have in the first place does not exist in our case. Furthermore, no other system can cope with the data overlays etc involved in compound data structures such as the ones we are considering. In addition, as mentioned earlier, it is not possible to have a completely canonical representation of programs, and a certain amount of what are essentially program transformation techniques have to be used to get round some of the differences between the actual surface plans that arise in practice and the plans in the plan library. Furthermore (and perhaps obviously) the language being used has an affect on the way people not only do, but even can, write their code, and this leads yet again to the actual surface plans differing in significant ways from the plans in the library. So the process of understanding a correct program will be shown first, and in the next chapter it will be shown how this differs in the case where the program has a bug in it. In what follows reference will be made not only to the plans and overlays already discussed, but also to many from Rich's library, using his original terminology (to avoid introducing yet more jargon into the program

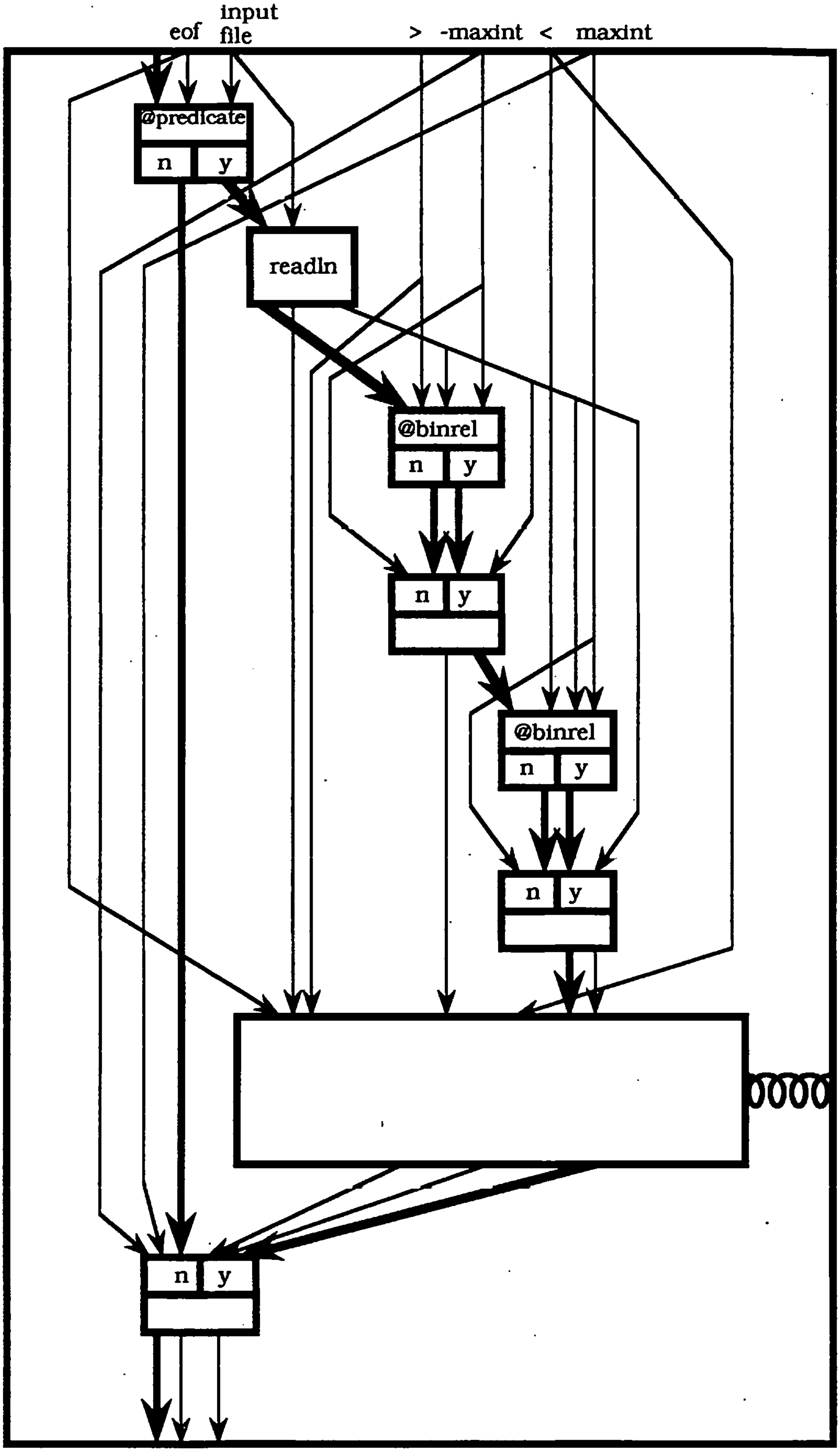


Figure 8.1
Max-Min Surface Plan (loop only)

understanding literature). Hopefully it will be clear what these plans and overlays are, and readers requiring more information are referred to [Rich 1981].

8.1 Understanding Programs using Plan Diagrams

To begin with we will consider a fairly simple program, whose analysis is quite straightforward:

```
program filemaxmin(input,output);
var n, biggest, smallest : integer;
begin
    biggest:= -maxint;
    smallest:= maxint;
    while not eof do
    begin
        readln(n);
        if n>biggest then
            biggest:=n;
        if n<smallest then
            smallest:=n
    end;
    writeln(biggest);
    writeln(smallest);
end.
```

This program computes the largest, and smallest of a set of numbers in a file. In this program the plans for computing the largest and smallest of a set of numbers share the code for actually sequentially generating the set of numbers. The surface plan for its loop is shown in Figure 8.1.

The analysis of this loop proceeds in several stages (not necessarily in the order given):

Step 1. Using the overlays discussed in Chapter 7, first of all the not is removed, essentially by interchanging the succeed and fail controlling conditions of the test.

Step 2. The rules for rewriting conditions such as "a<b" in terms of "b>=" described in Chapter 7 come into effect. This gives us the plan diagram shown in Figure 8.2.

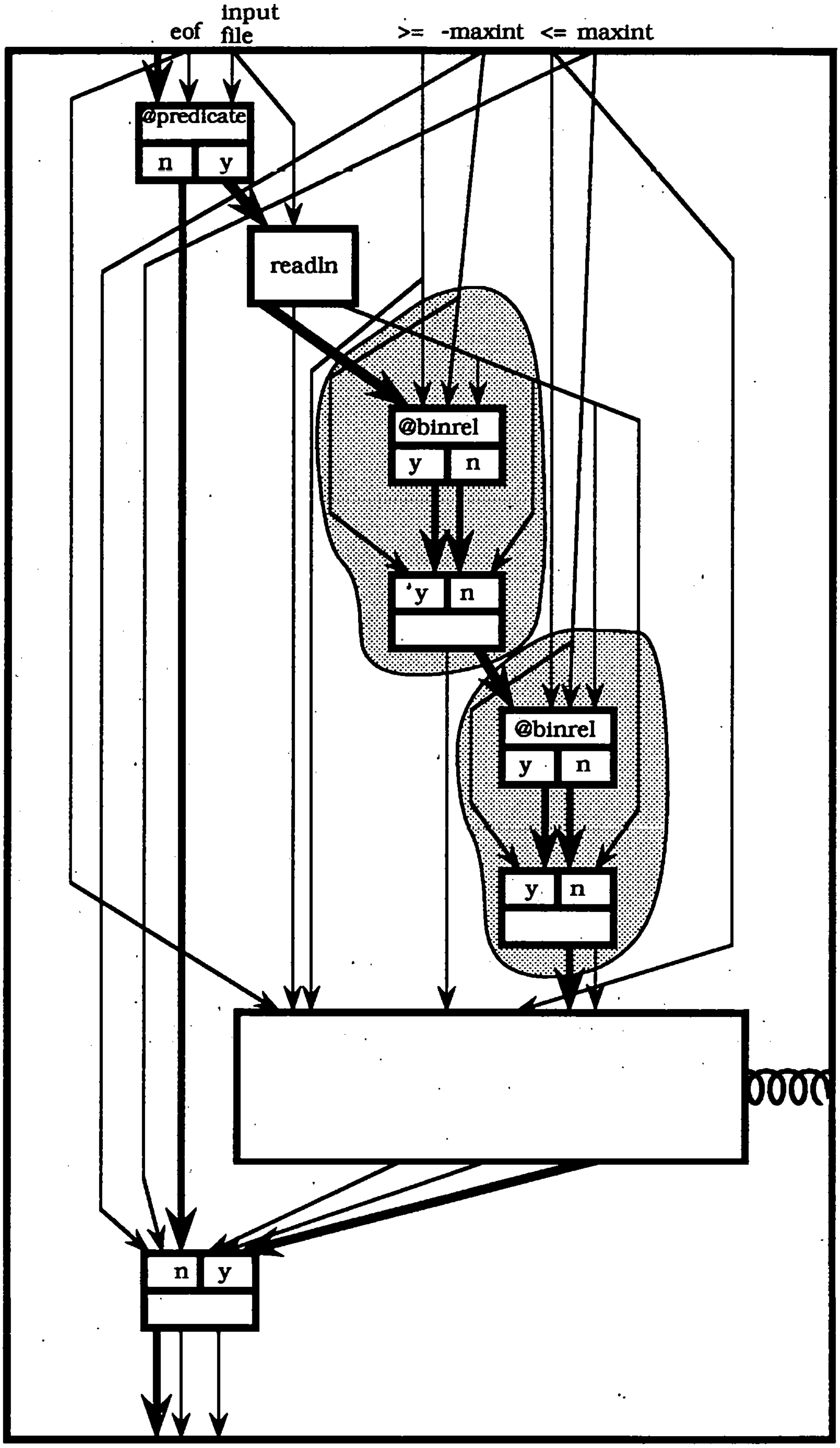
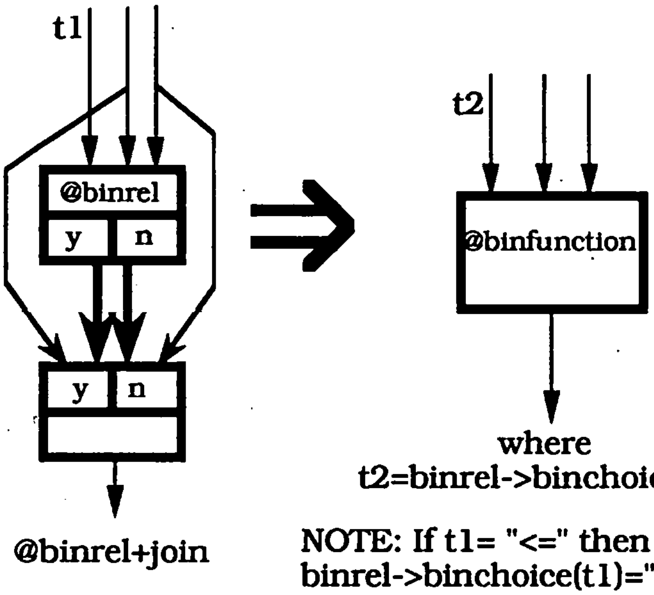


Figure 8.2
After Not Removal Etc.



where
t2=binrel->binchoice(t1)

NOTE: If t1= "<=" then
binrel->binchoice(t1)="lesser".

If t1= ">=" then
binrel->binchoice(t1)="greater".

Figure 8.3
Binrel+Join->@Choice

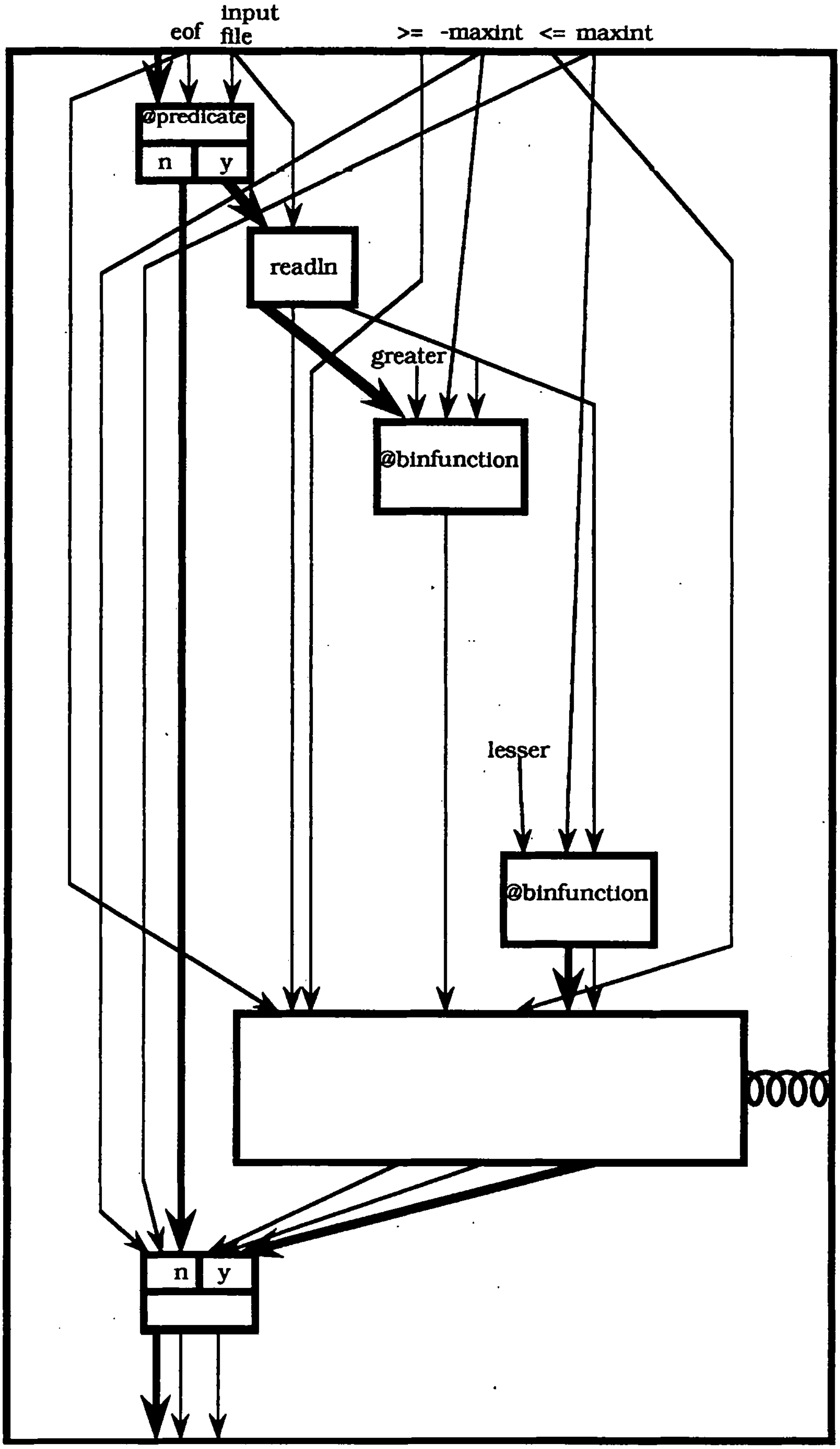


Figure 8.4
After Greater And Lesser Found

Step 3. The shaded areas of Figure 8.2 are now recognised as being instances of **@binrel+join** plans. Using the overlay shown in Figure 8.3, these are recognised as implementing **choice** operations i.e. operations which choose between their inputs depending on the result of the test. In particular they are recognised as implementing choosing the greater of two numbers (**greater**), and choosing the lesser of two numbers(**lesser**), plans. This results in Figure 8.4.

Step 4. Now the shaded portion of Figure 8.5 is recognised as being an instance of an **iterative-accumulation** plan. **Iterative accumulation** is the plan common to such things as iteratively adding a set of numbers, iteratively multiplying a set of numbers, and computing the maximum or minimum of a set of numbers. For instance for adding a set of numbers the initial value input into the plan is zero and the operation is "+". For computing the maximum of a set of numbers, the initial value can be anything which is less than than all the numbers in the set (-maxint in this case), and the operation is **greater**. In a similar fashion an **iterative accumulation** plan is found involving the **lesser** operation. Note that at the moment only the plans have been found - the step to recognising **max** and **min** has not been made yet, since these involve sets, and as yet no sets have been found.

Step 5. In a similar fashion, a standard (Pascal specific) plan, **iterative-readln**, is found. This is shown in Figure 8.6. Notice how all of these plans share common structure.

Step 6. Now overlays and temporal abstraction come into their own. Figure 8.7 shows two overlays **iterative-readln->readall**, and

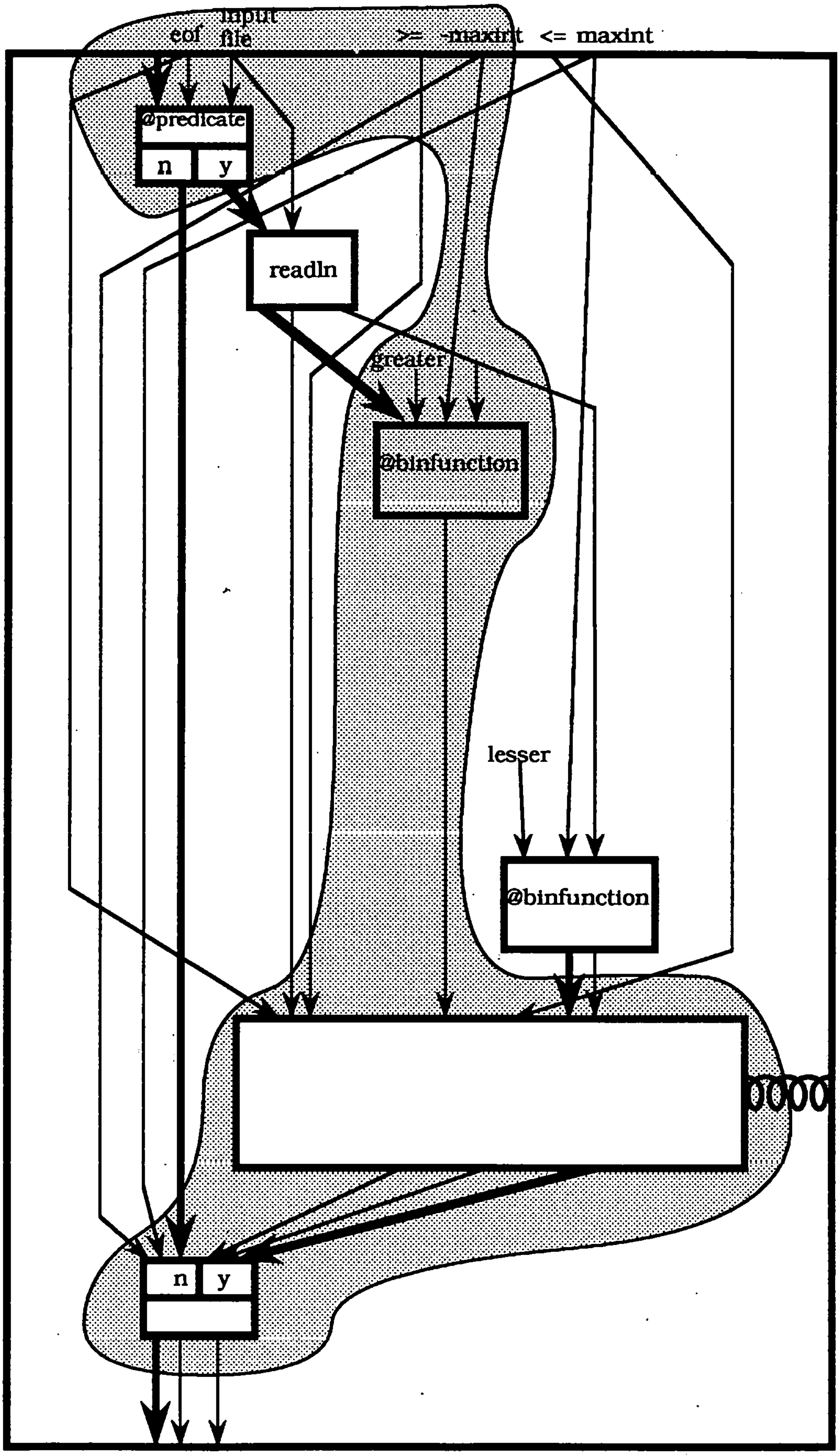


Figure 8.5
Showing Iterative Accumulation

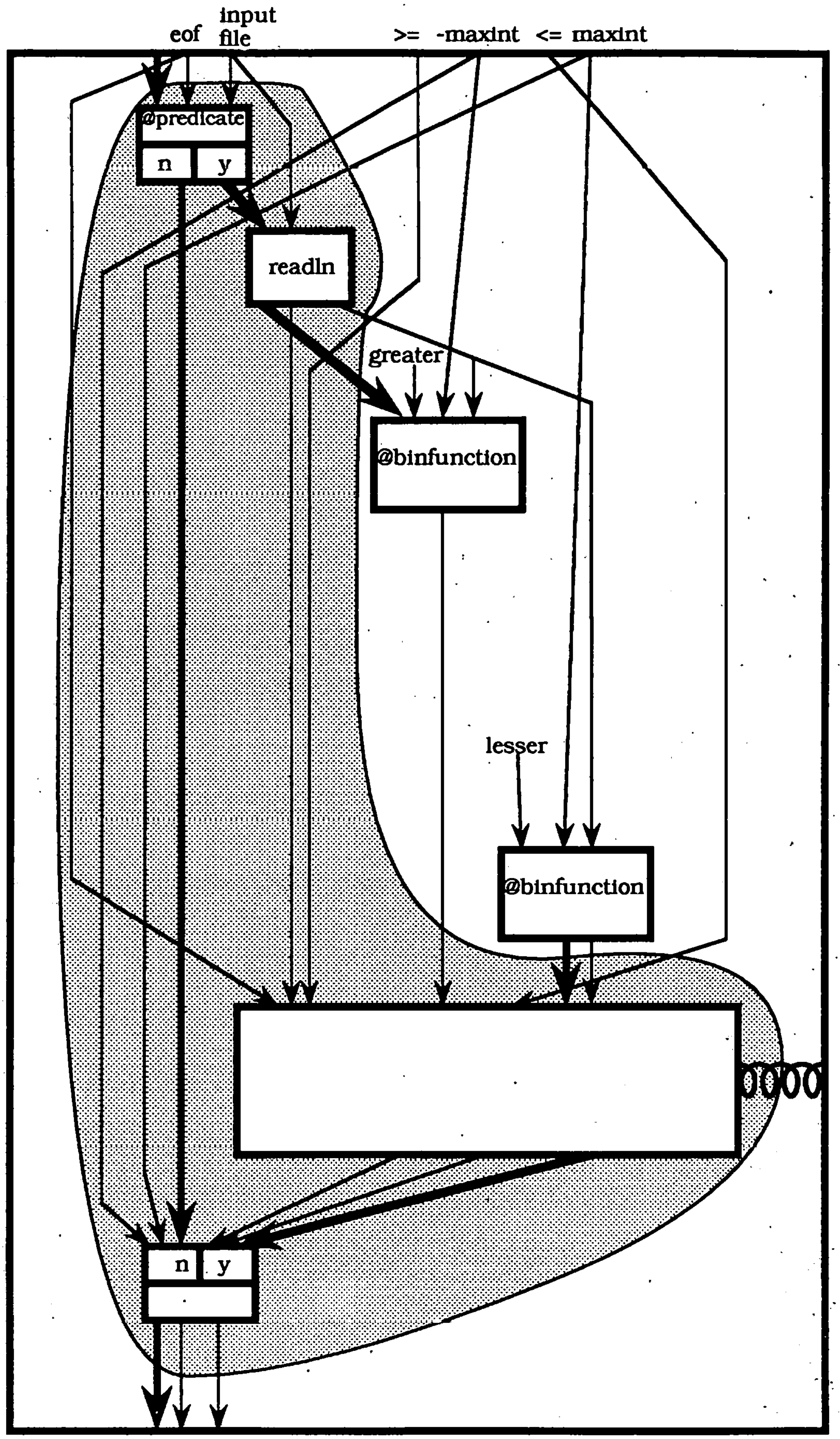
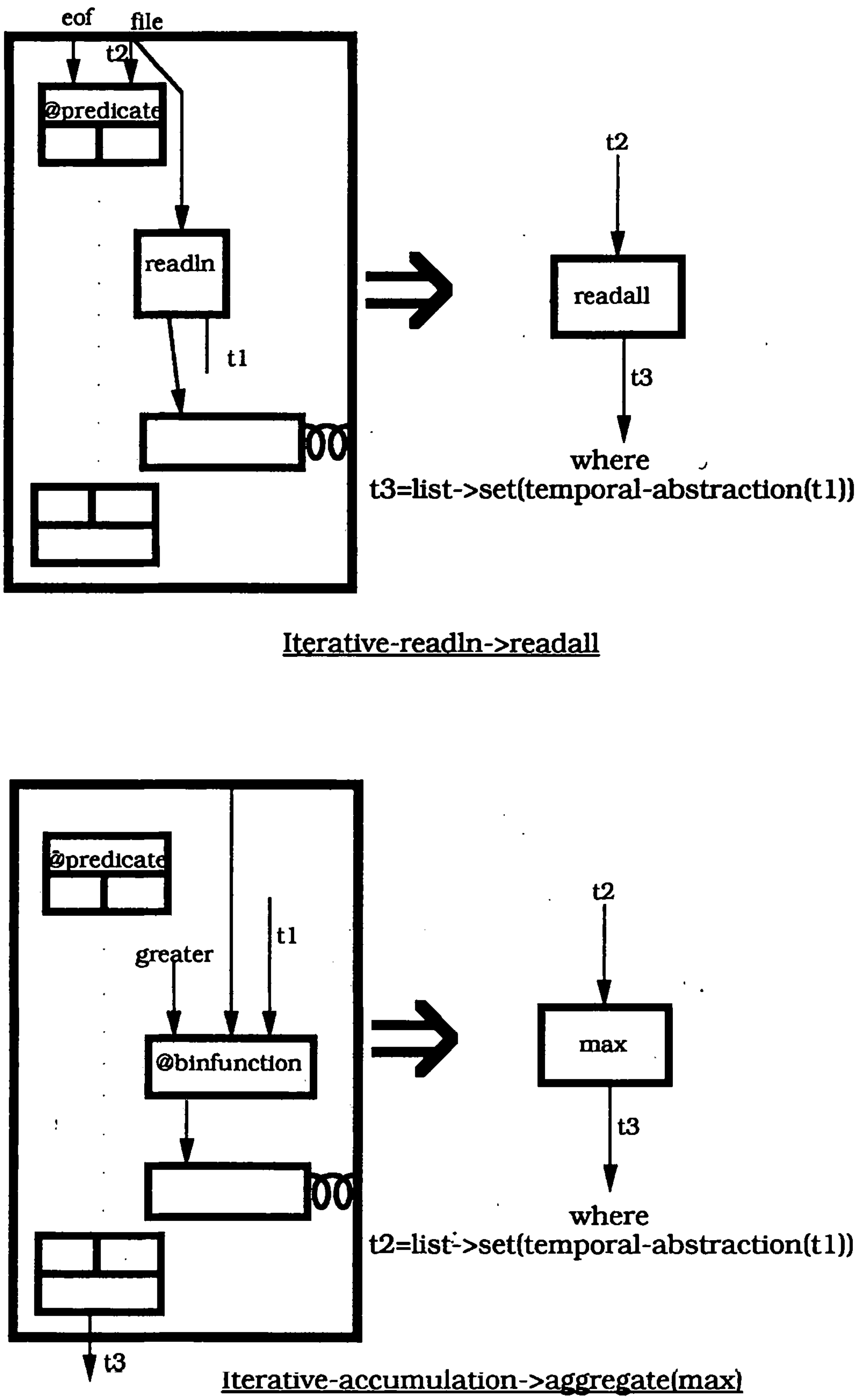


Figure 8.6
Showing Iterative Readln



Iterative-readln->readall

Iterative-accumulation->aggregate(max)

Figure 8.7
Iterative-readln->readall and
Iterative-Accumulation->aggregate(max)

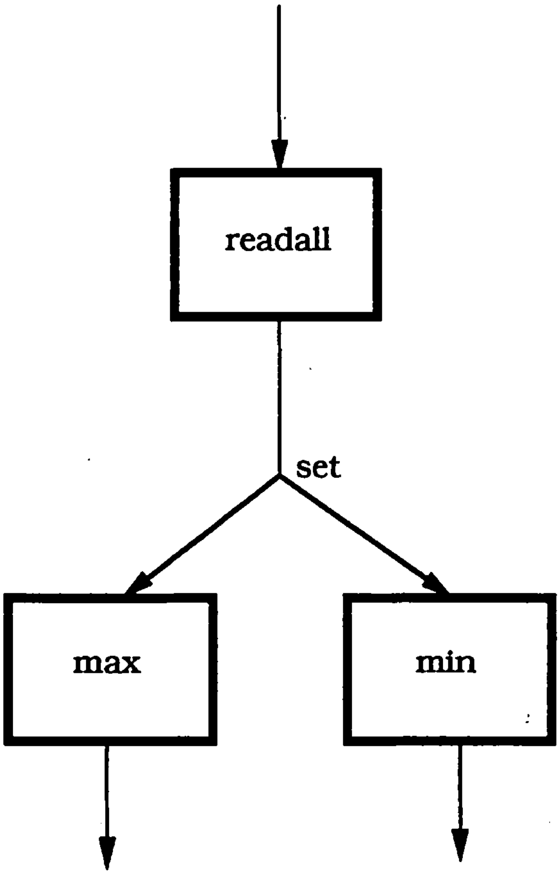


Figure 8.8
Final Analysis for Max-Min Loop

iterative-accumulation->aggregate(max)[1]. Suppose **iterative-readln->readall** is done first. As described in Chapter 7, this adds a **readall** patch to the chart, and creates two new tie-points, one corresponding to the temporal abstraction of the value output from the readln operation (so this tie-point represents a <u>list</u>), and the other representing this list viewed as a set. Assertions about these are made in the data-overlay database. Now the **iterative-accumulation->aggregate(max)** overlay is used. This adds a **max** patch to the chart. Now it tries to create a tie-point corresponding to the temporal abstraction of the input to the **greater** operation. This is the same tie-point as is output from the **readln**, so when it consults the data overlay database, it finds the temporal abstraction of this tie-point has already been created. So it uses the one that is already there. Then it tries to create a tie-point corresponding to the **list->set** view of this tie-point, and again it finds it is there already. Thus the **max** operation acquires as its input set the same set as is output from the **readall** operation. In a similar fashion the **min** operation is recognised, and it too connects to the same tie-point. So the final analysis of the loop is as shown in Figure 8.8.

The second program we will consider is the following:

[1] Strictly speaking there are two overlays involved here - one taking **iterative-accumulation** to **aggregate** and one taking **aggregate**, with **greater** as its input operation to **max**.

```
program sort(input,output);
type listelement = record
                   numb : integer;
                   next : ^listelement;
                   end
      plist = ^listelement;
var head, p : plist;
    n : integer;

procedure addtolist(n : integer; t : plist);
var  p : plist;
begin
    new(p);
    p^.numb:=n;
    if  t = nil then
         begin
              p^.next := head;
              head:=p;
         end
    else
         begin
              p^.next:=t^.next;
              t^.next:=p;
         end;
end;

procedure findplace(n: integer; var p: plist);
var  t : ^listelement;
     found : boolean;
begin
    if  head^.numb > n  then
         p:=nil;
    else
         begin
              p:=head;
              t:=p^.next;
              found:=false;
              while not found do
                   if  t <> nil then
                        if  t^.numb <= n then
                             begin
                                  p:=t;
                                  t:=t^.next;
                             end
                        else found:=true
                   else found:=true;
         end;
end;

begin
    head:=nil;
    while not eof do
         begin
              readln(n);
              if head<>nil then findplace(n,p)
              else p:=head;
              addtolist(n,p);
         end;
    p:=head;
    while p<>nil do
         begin
              writeln(p^.numb);
              p:=p^.next;
         end;
end.
```

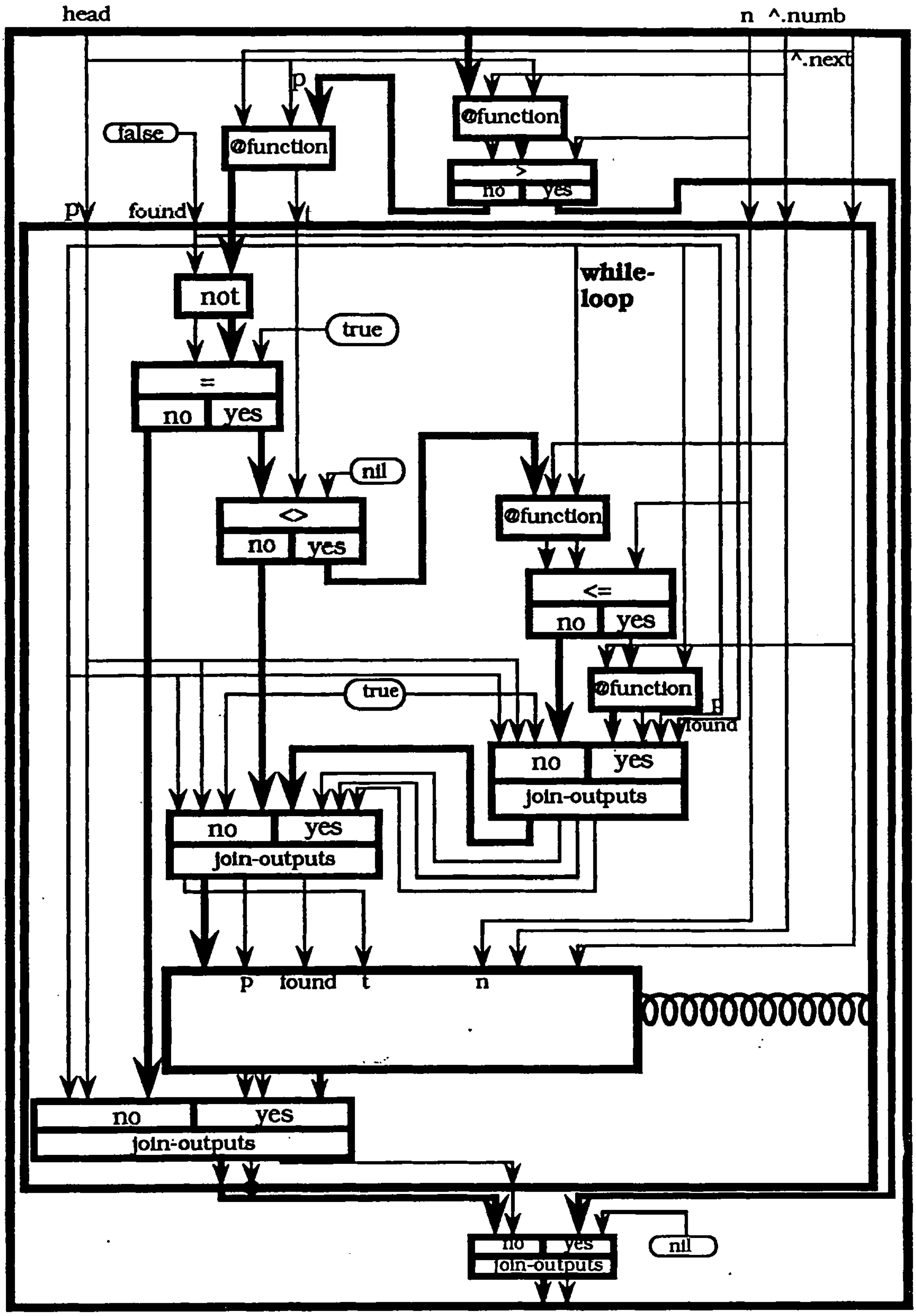


Figure 8.9
Findplace Surface Plan

This is a much more complicated program which reads in numbers from a file or terminal (one per line) until the end of the file, and outputs the numbers in ascending order. It does this by use of a linked list which holds the numbers, each new number being stored in the appropriate place in the list. The program contains a couple of user procedures - **findplace**, which finds the correct place in the sorted list where a new number is to go, and **addtolist**, which actually adds in a new number to the list. We will begin by considering the procedure **findplace**. Its surface plan is shown as Figure 8.9. In what follows we will not describe the understanding process in full detail in terms of patches added to the chart since there are far too many of these for the description to be helpful. Instead we will describe the process in terms of the main steps that the patches represent, and illustrate it by diagrams in which each NAPE represents a patch in the chart, but in which only some of the patches have been shown. The understanding (via plan recognition) process thus proceeds in several steps (not necessarily in the order given):

To start with the loop is analysed, as follows:

Stép 1 **nots** are removed, simply by changing the order of the yes/no labels on the appropriate tests and joins. There is only one case of this, and the resulting surface plan is shown as Figure 8.10. This is actually done by the rule, discussed earlier in Chapter 7.

Step 2 The staggered **joins** are turned into **n-join-output** joins, as discussed in Chapter 4. This results in Figure 8.11.

Step 3 IDS now finds an **iterative flag test** as discussed in Chapter 7. It can therefore remove the =true test (and corresponding joins), and move the recursive role to within the other two tests, by changing its controlling condition, and reconnecting appropriately.

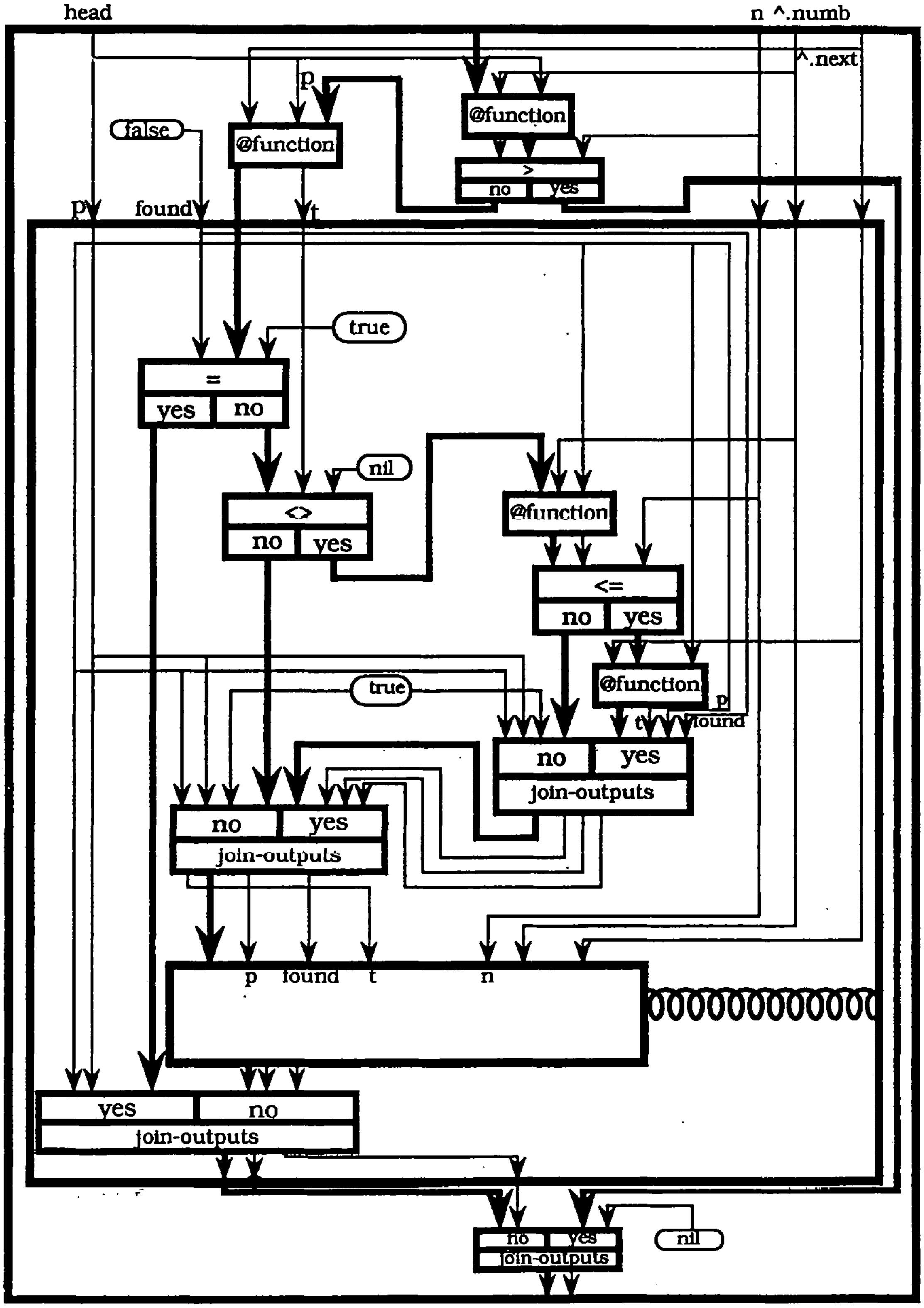


Figure 8.10
After Not Removal

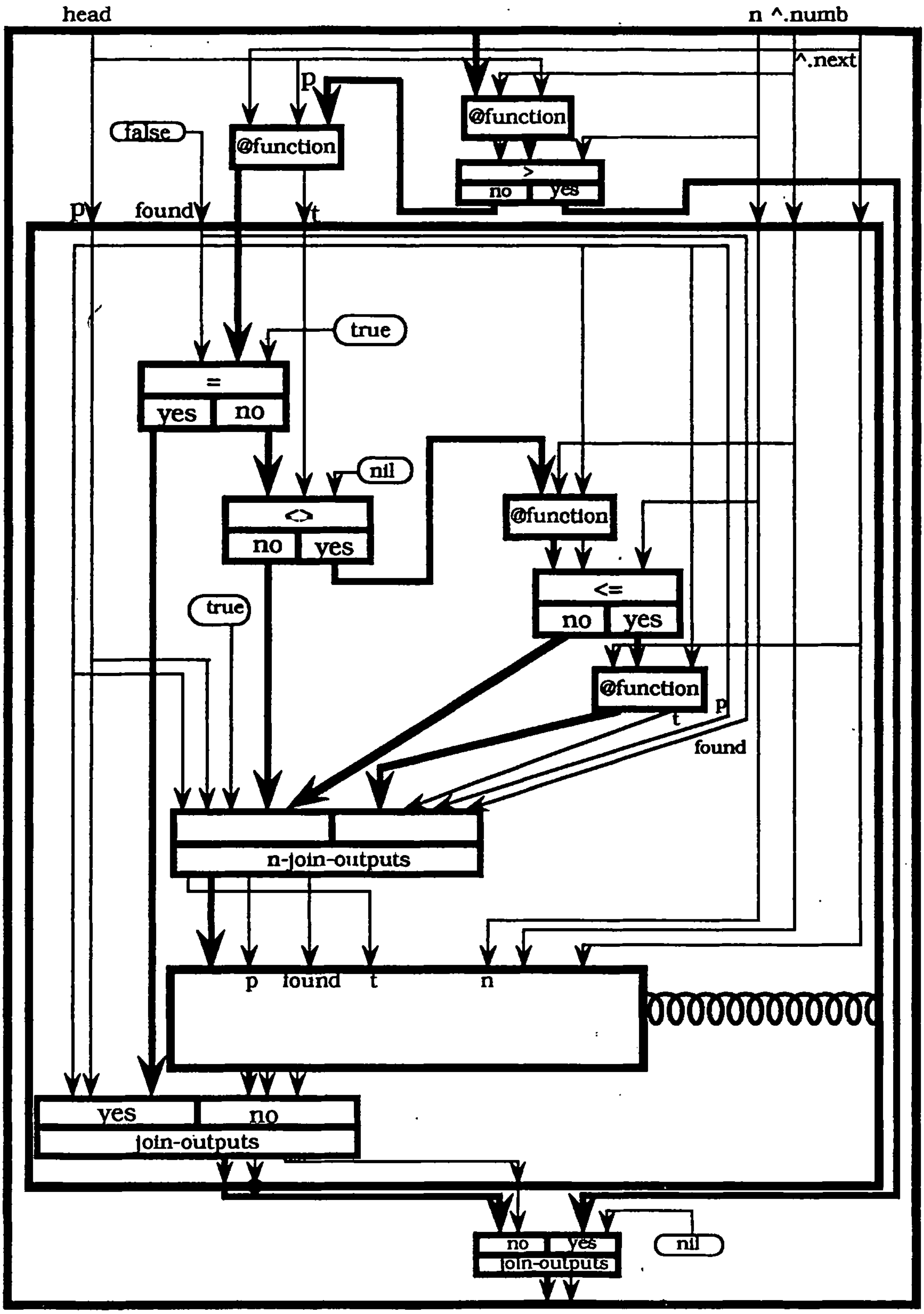


Figure 8.11
After n-join-output created

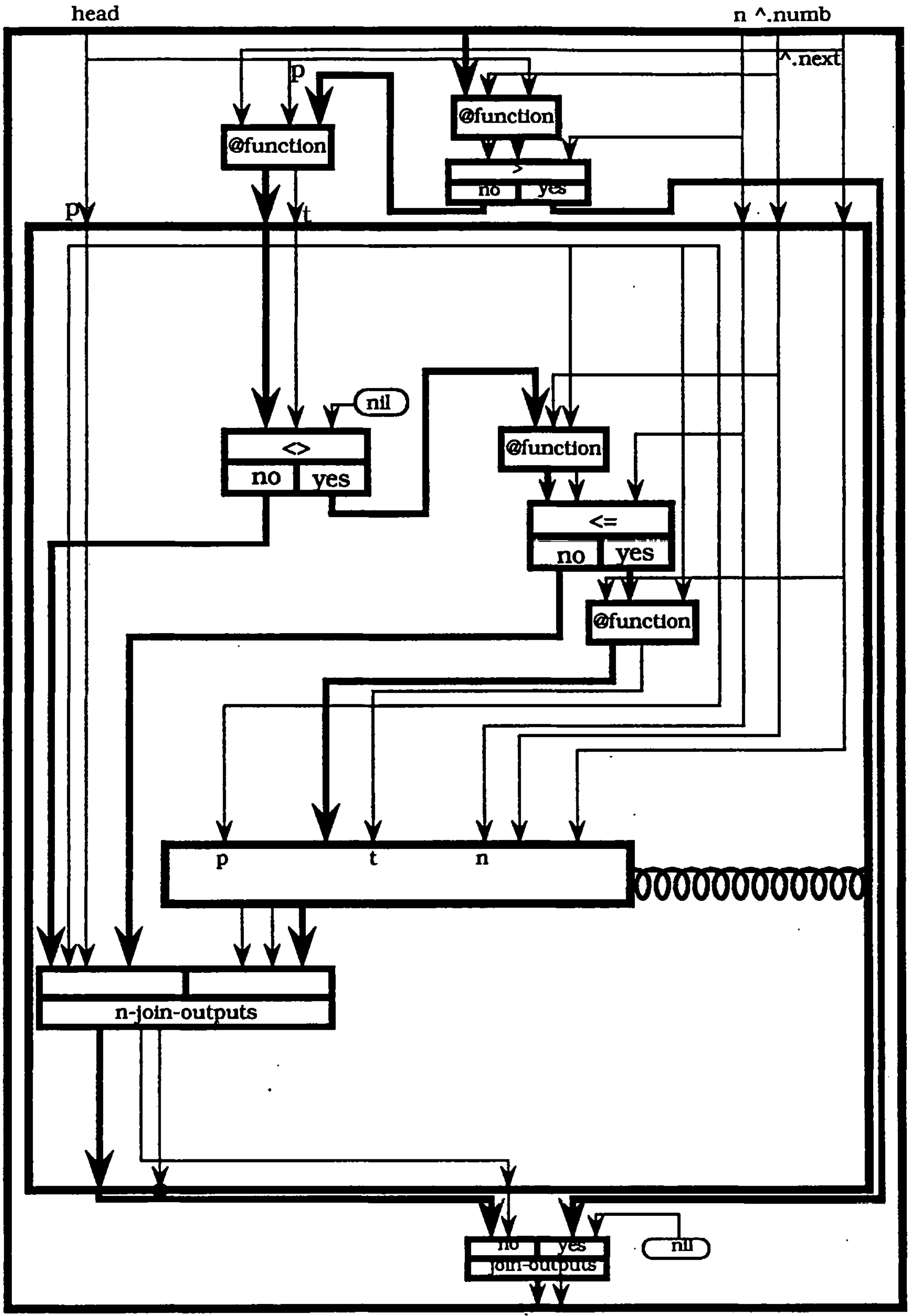


Figure 8.12
After Iterative Flag Test Removal

Note that all the other NAPEs (strictly speaking, patches) also have their controlling conditions altered by this. The resulting plan diagram is shown in Figure 8.12.[2]

It is worth commenting here that the fact that this last step has been necessary is really an artifact of Pascal. Pascal does not use the McCarthy form [] of the Boolean operators **and** and **or**, and evaluates both conditions C1 and C2 in Boolean expressions such as:

C1 **or** C2

The piece of code involved in the graph manipulations just described is actually trying to iterate down a list until either the end of the list is encountered, or some condition is met. The natural way to code this would have been something like (the while form of the loop will continue to be used here):

while not(at-end-of-list **or** condition-met) **do** ...

Unfortunately this could not be done in Pascal because the condition to be met involves dereferencing a pointer which may be nil. Even though the first part of the test would have detected this, Pascal would go on to try and evaluate the second part of the **or**, giving rise to a run-time error. In a language with the McCarthy form of the Boolean operators, as soon as the first part of the **or** evaluated to true, the expression would be considered to be true, and the second condition would not be evaluated. In a language like this (including LISP) the loop could have

[2] In an earlier version of this work (Lutz,1989) this was achieved by a series of four separate program transformations. However, our use of **n-join-output** nodes, rather than staggered single **join-output** nodes, and the use of controlling conditions has greatly simplified the process.

been written as above, and the corresponding surface plan would have been more or less what we have now arrived at after a graph transformation operation.

Step 4. (i) Put termination tests in standard form. Plans in the library are stored with the termination tests on loops being just that i.e. the tests succeed when the loop is to terminate. This step is actually rather similar to the **not** removals of step 1, and is actually handled by the rules discussed in Chapter 7, which have automatically given us patches for all the equivalent variant ways of expressing tests involving comparison operators.

(ii) Recognise complex predicates (actually this part of this step could have been done at any time but its description has been left till now). By this is meant the process of recognising that a condition like

$f(x) < n$

can be regarded as a complex predicate of **x** by first of all regarding the combination of a binary relation (in this case <) and a fixed value (in this case n) as a predicate g defined by

$g(y)$ = true if $y < n$, and $g(y)$=false otherwise

and then regarding the combination of a function (in this case f) and a predicate (in this case g defined above) as a 'compound' predicate h, defined by:

$h(y)=g(f(y))$

In this way both test boxes are replaced by **@predicate** boxes which take as input a predicate and an object. If such complex predicates have an obvious or easily understood name (such as =nil)

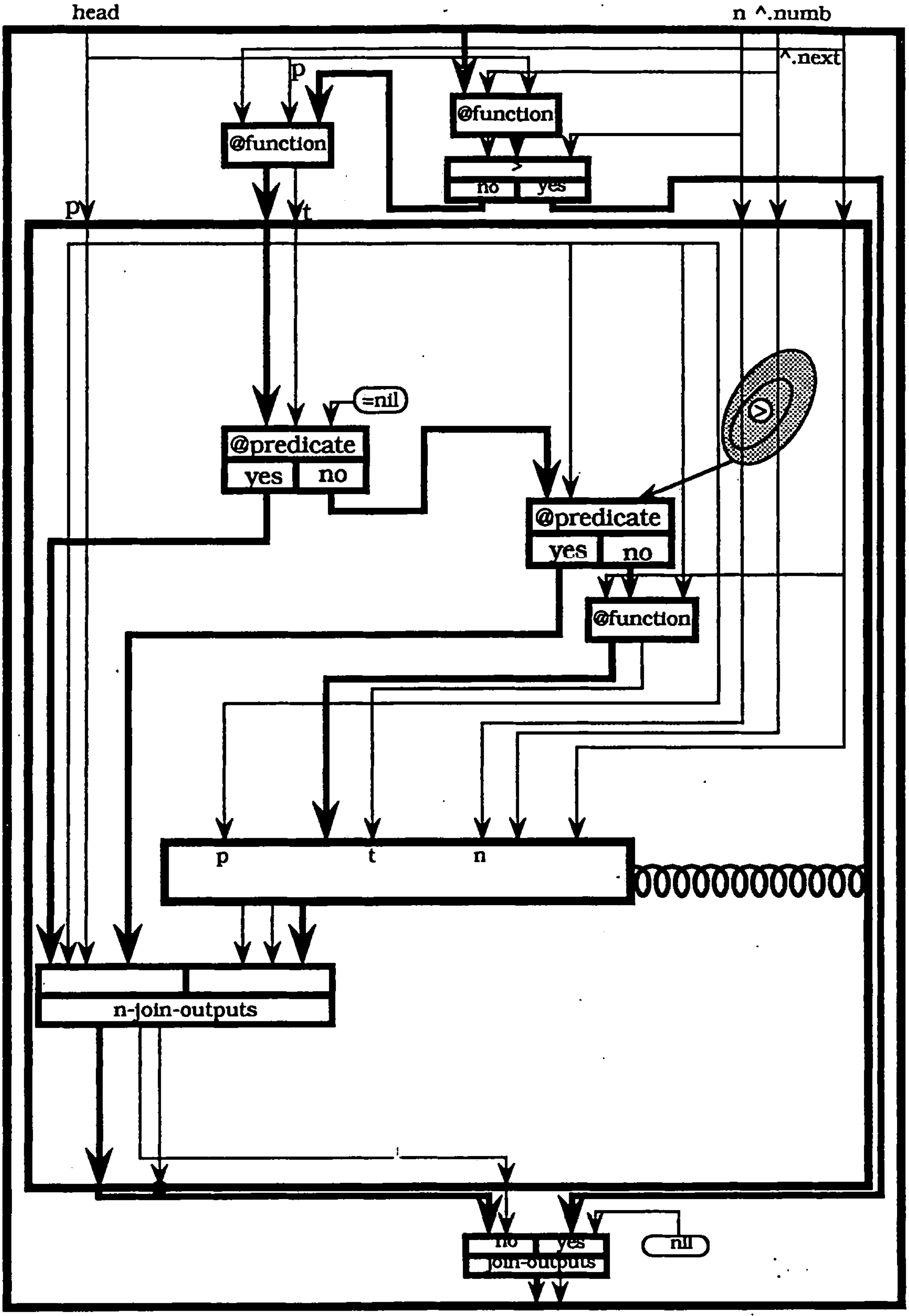


Figure 8.13
After Composite Object Grouping Etc.

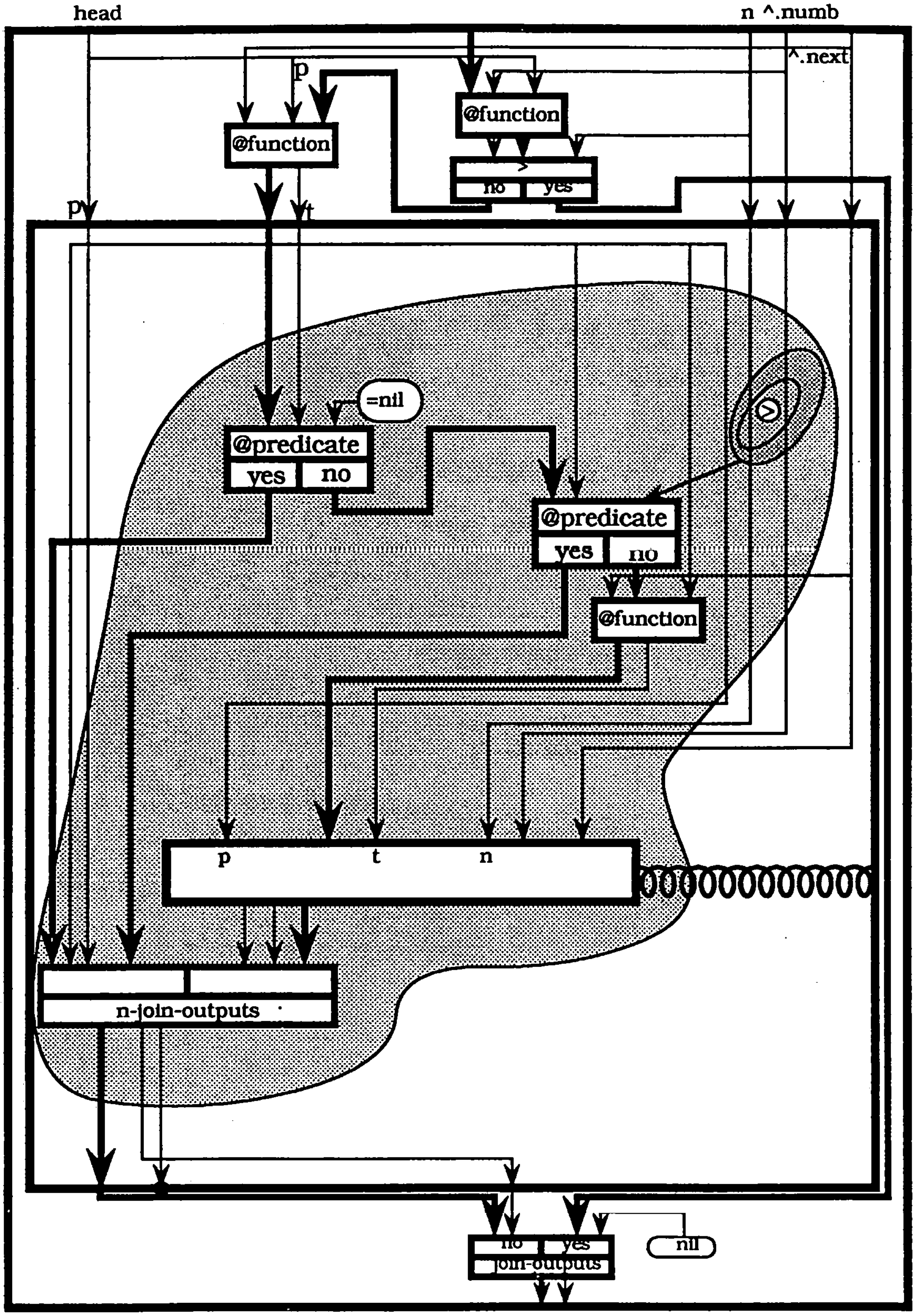


Figure 8.14
Terminated Iterative Generation and Search

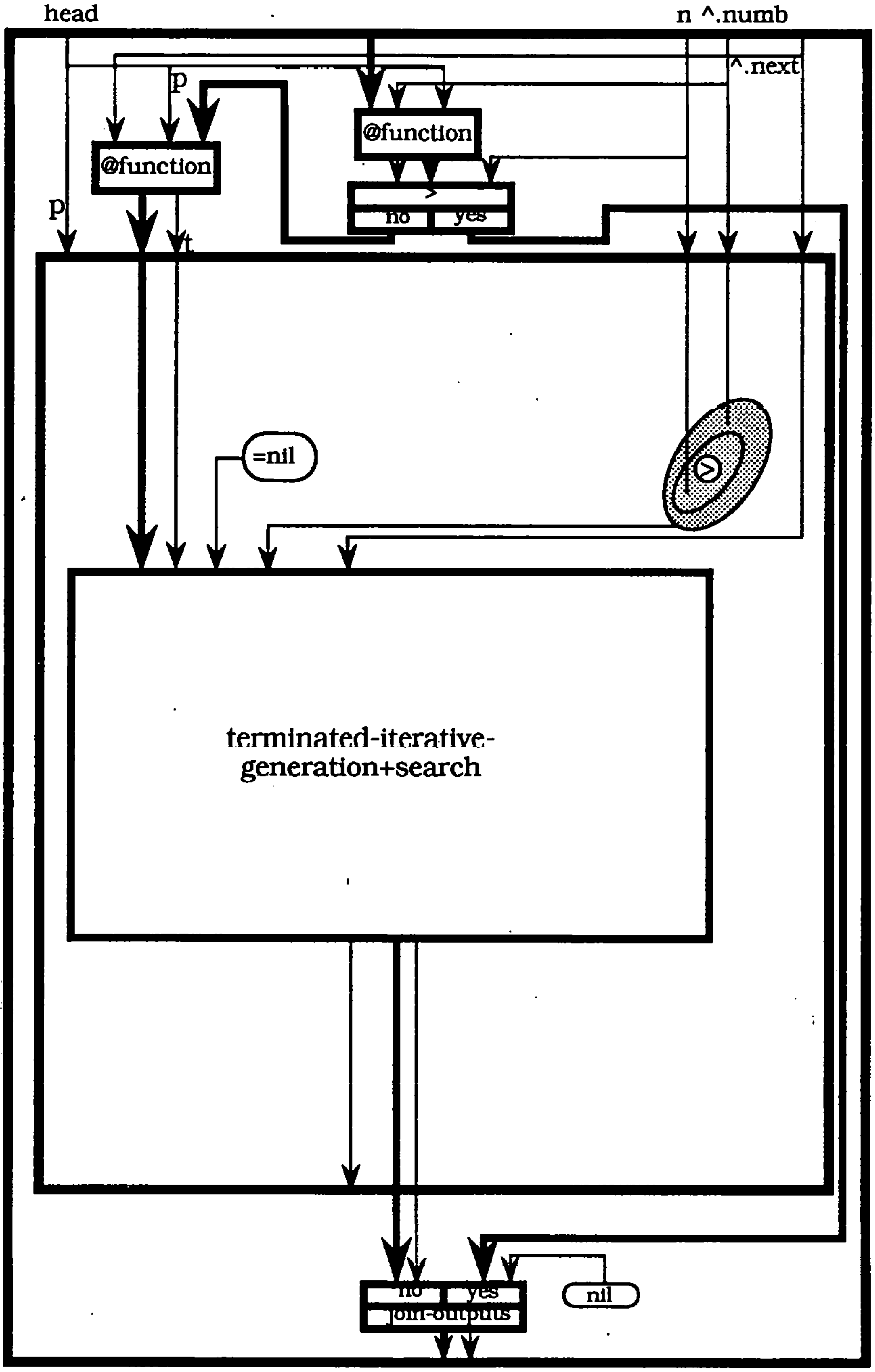


Figure 8.15
After Recognition of Terminated Iterative Generation and Search

then this has been used, otherwise the grouping is shown by shaded ovals round the appropriately grouped objects. Such shaded objects are then treated as as single objects which can be inputs to (or indeed outputs from) computations in plan diagrams, since in fact they represent the new tie-points introduced by data overlays. This is all done using overlays such as **@function+predicate->predicate** from Rich's library. The plan diagram resulting after this has been done is shown in Figure 8.13.

Step 5. The standard plan, **terminated-iterative-generation+search**, shown shaded in Figure 8.14, is now recognised, resulting in the plan diagram Figure 8.15. This plan represents the plan common to code which searches for some object, returning the object if found, but stopping the search when there are no more objects. In this case it returns the final object.

Now the analysis of the loop is finished, so its patches are copied back into the chart for the main-procedure. The analysis then continues as follows:

Step 6. The plan **@successor+terminated-iterative-generation+search**, shown shaded in Figure 8.16 is recognised. After applying an overlay **@successor+terminated-iterative-generation +search->terminated-trailing-generation+search** Figure 8.17 is then obtained.

Step 7. Up till now all that has really been recognised is a standard pattern of code. At this point however the notion of overlays really comes into its own, and enables IDS to start reasoning about the more abstract operations on more abstract data types that this pattern of code is actually implementing. In particular IDS recognises that the

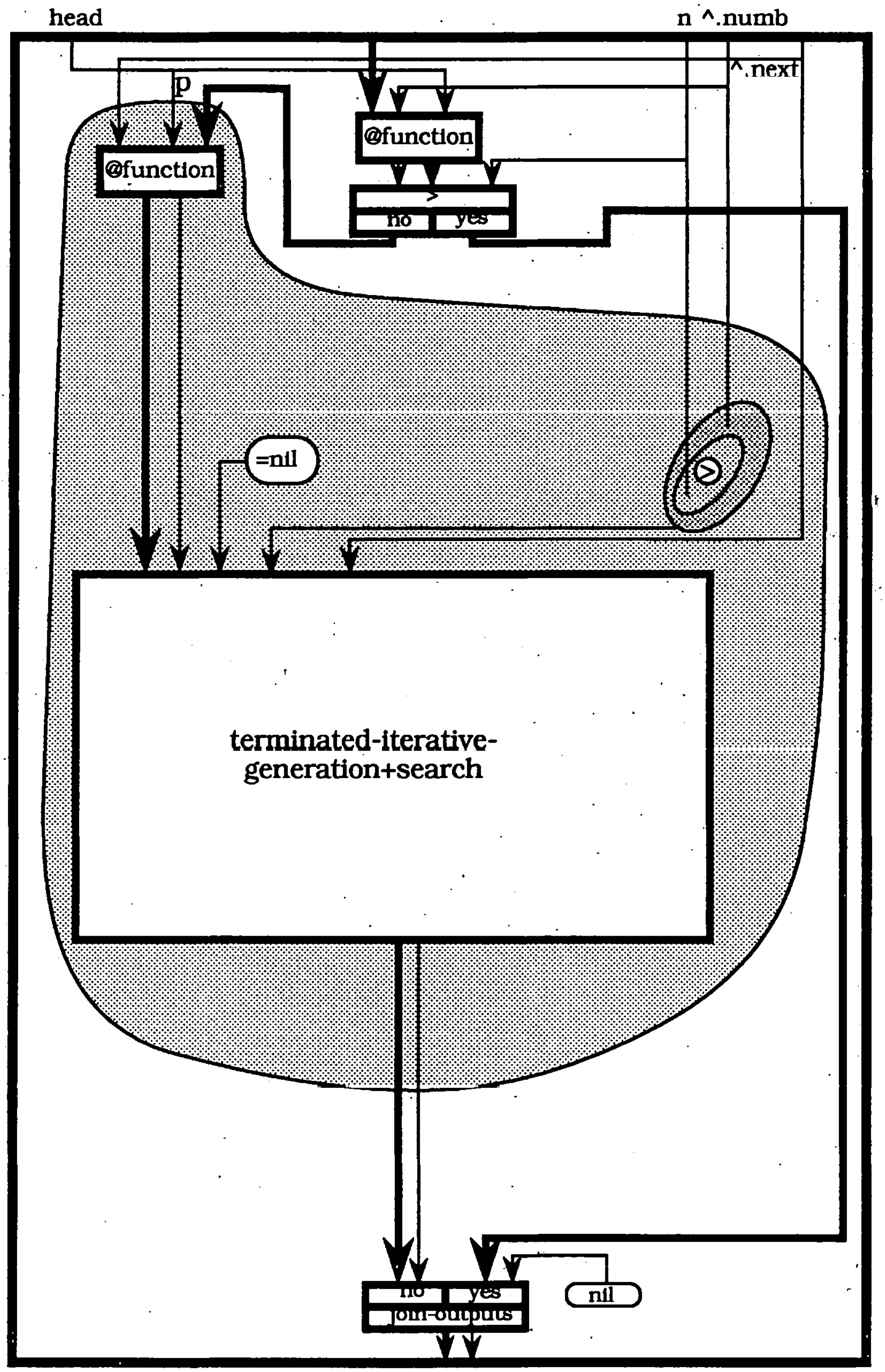


Figure 8.16
@sucessor+Terminated-Iterative-Generation+Search

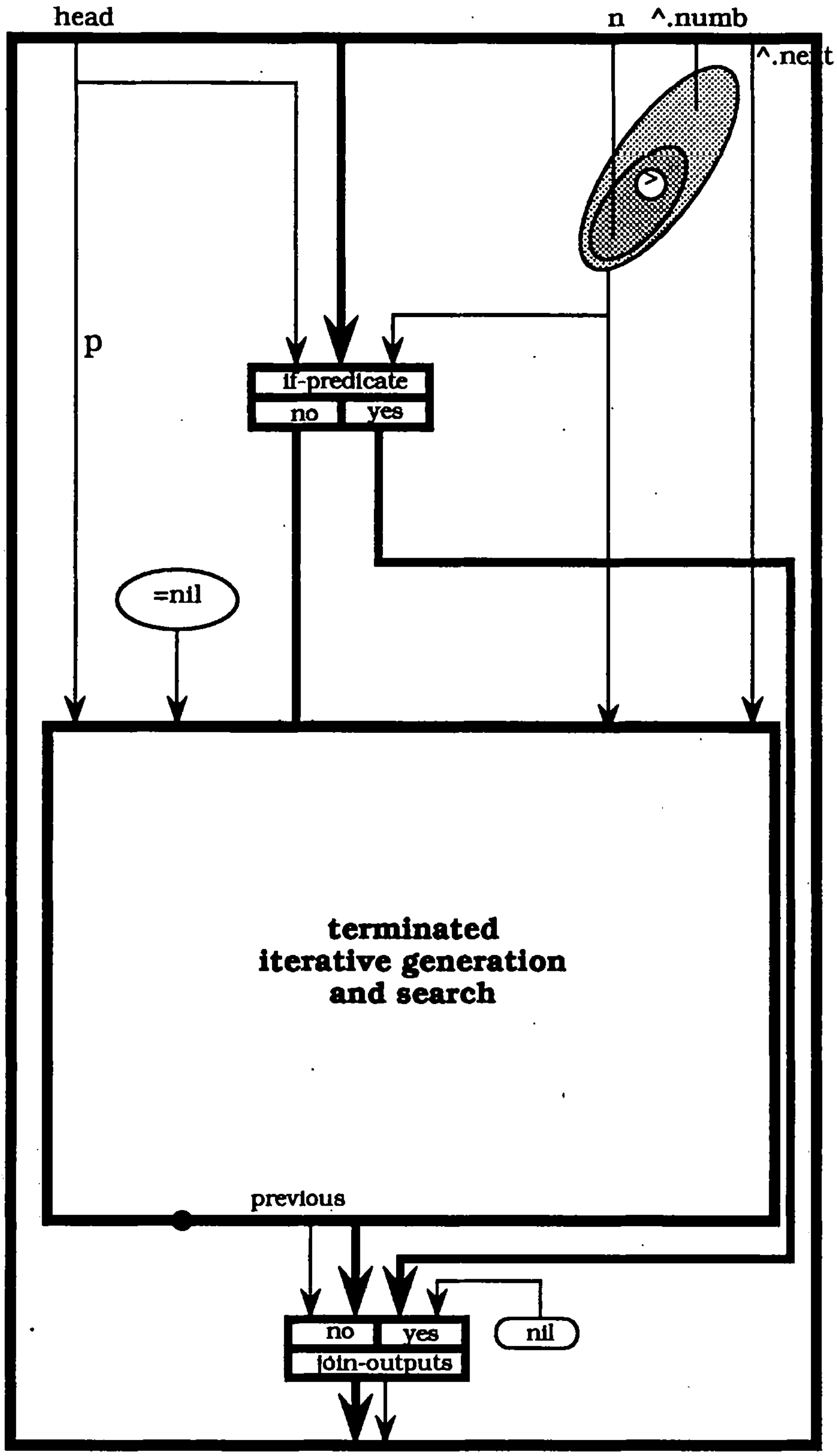


Figure 8.17
After Terminated Trailing Generation+Search Recognised

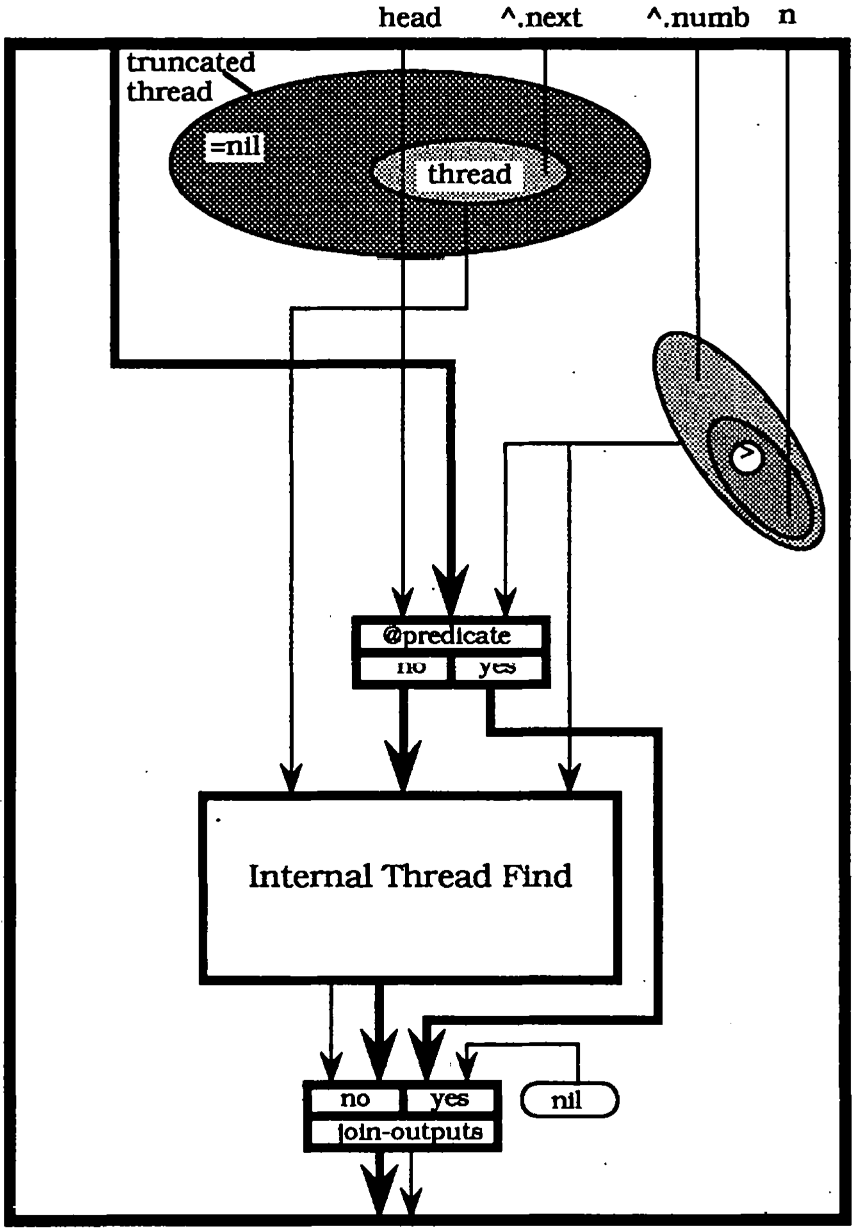


Figure 8.18
After Internal Thread Find Recognised

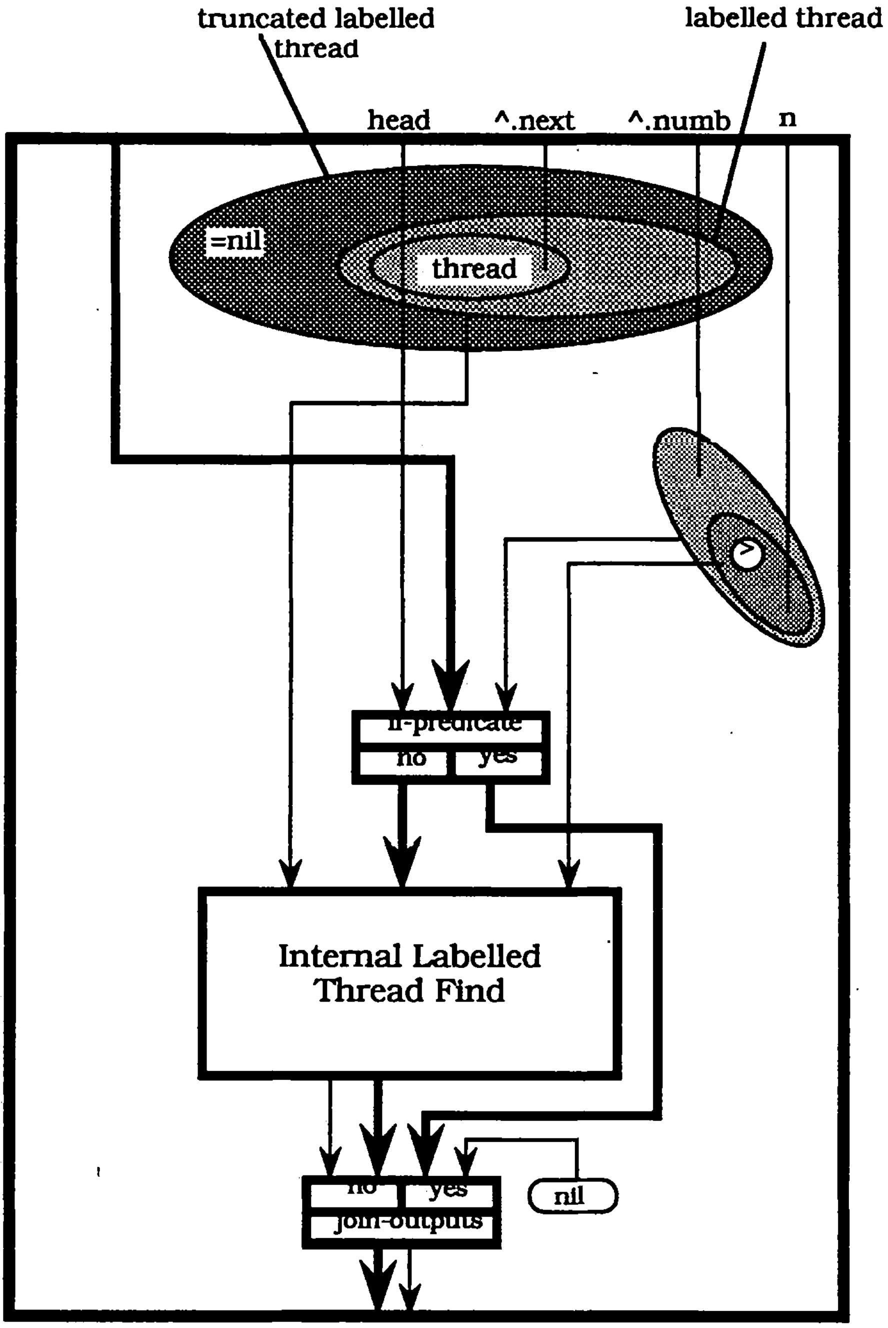


Figure 8.19
After Internal Labelled Thread Find Recognised

above plan can be viewed as implementing the **internal-thread-find** operation discussed earlier, with a **nil-terminated thread** as input. This **nil-terminated thread** essentially is that obtained by iterating down the data structure starting from head, and using ^.next as the successor function. In this way we arrive at Figure 8.18.

Step 10. Finally IDS is able to recognise that this pattern of code actually implements an **internal-labelled-thread-find** operation, resulting in Figure 8.19, and this is essentially as high-level as we go for this procedure.

In a similar fashion **addtolist**, whose surface plan is shown as Figure 8.20, is first of all seen to implement the plan shown in Figure 8.21, in a similar fashion to that described in Chapter 7. Then the combination of the topmost **newarg** (following the **new** operation) and the **spliceafter** is recognised as an **internal-labelled-thread-add-after**, and the two **newargs** are seen as implementing a **new-labelled-root** operation resulting in Figure 8.22. Notice how each of these two operations was split with part occurring before the test, and part after, and notice also how the topmost **newarg** is shared between them, thus demonstrating the power of structure sharing, and of the controlling condition manipulations. It is partly this ability of the plan diagram formalism (and of the chart parsing recogniser this project is using) which makes it so powerful.

Having analysed both procedures to the best of its ability IDS now turns its attention to the main program. Its surface plan (or at least most of it) is shown in Figure 8.23. To analyse this it substitutes its understanding of the user procedures **findplace** and **addtolist** in place of the boxes representing the calls to these procedures. Note that this is done with its high-level accounts of what the procedures do, so it is

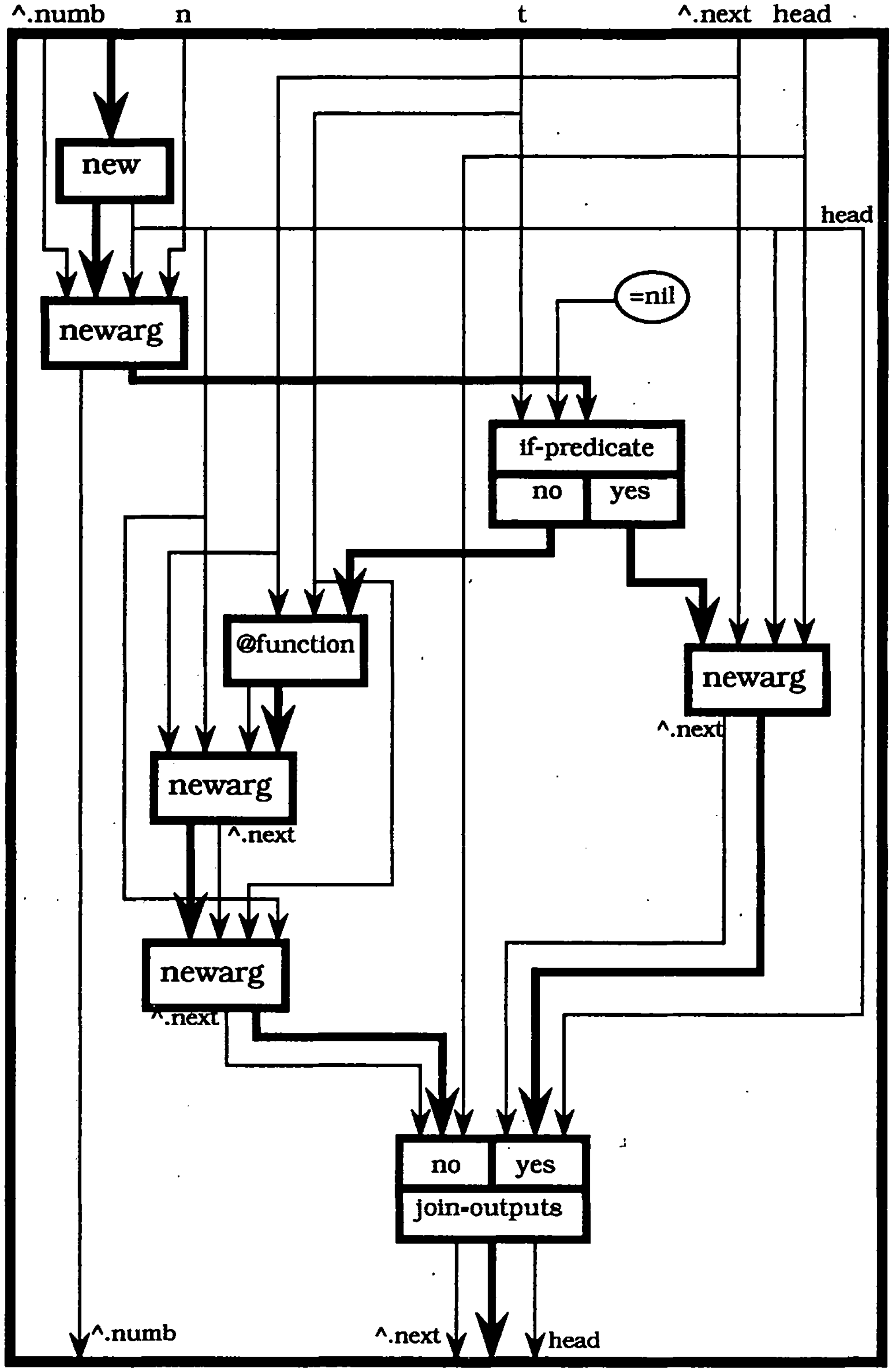


Figure 8.20
Addtolist Surface Plan

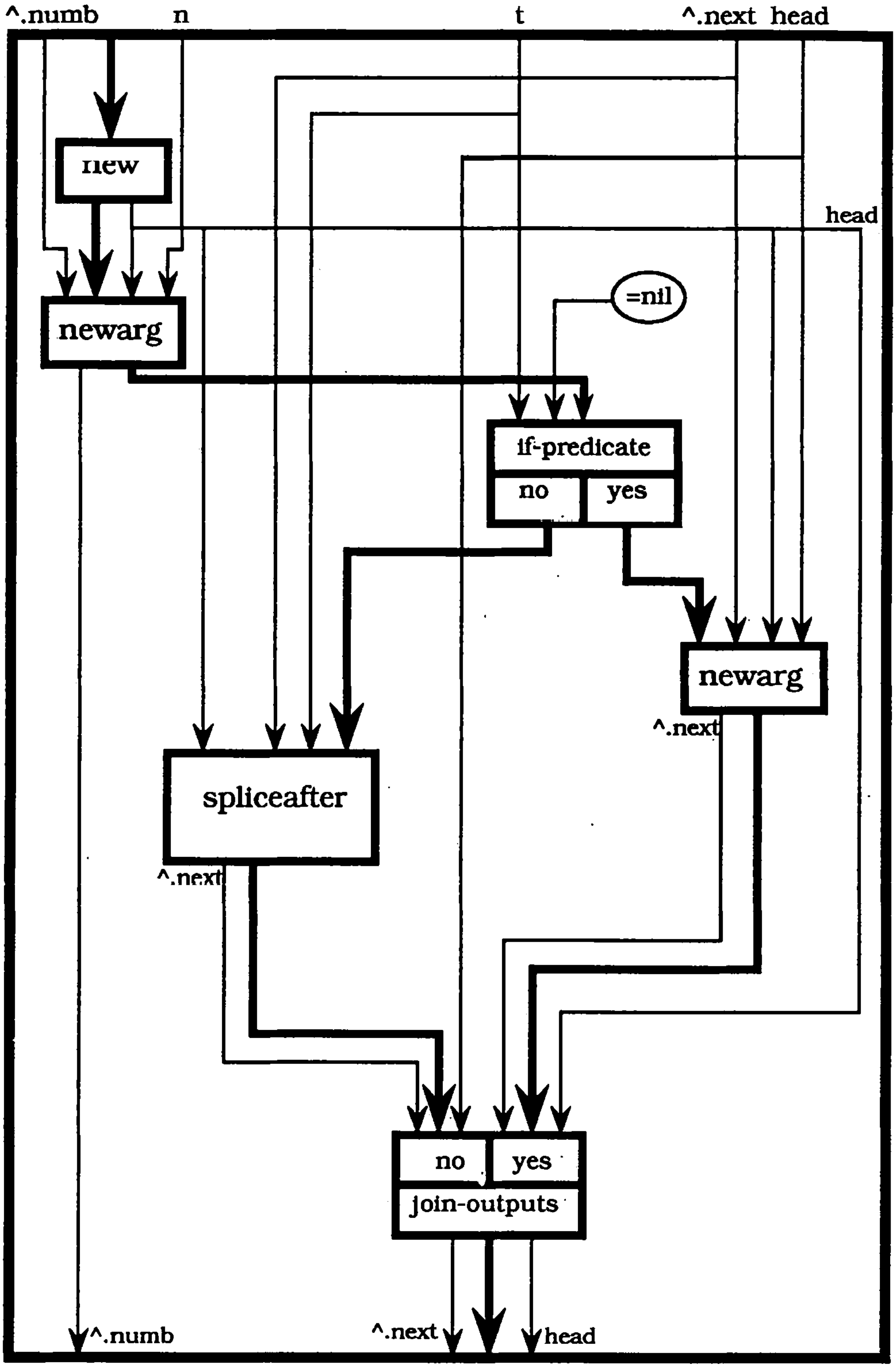


Figure 8.21
After Spliceafter Recognised

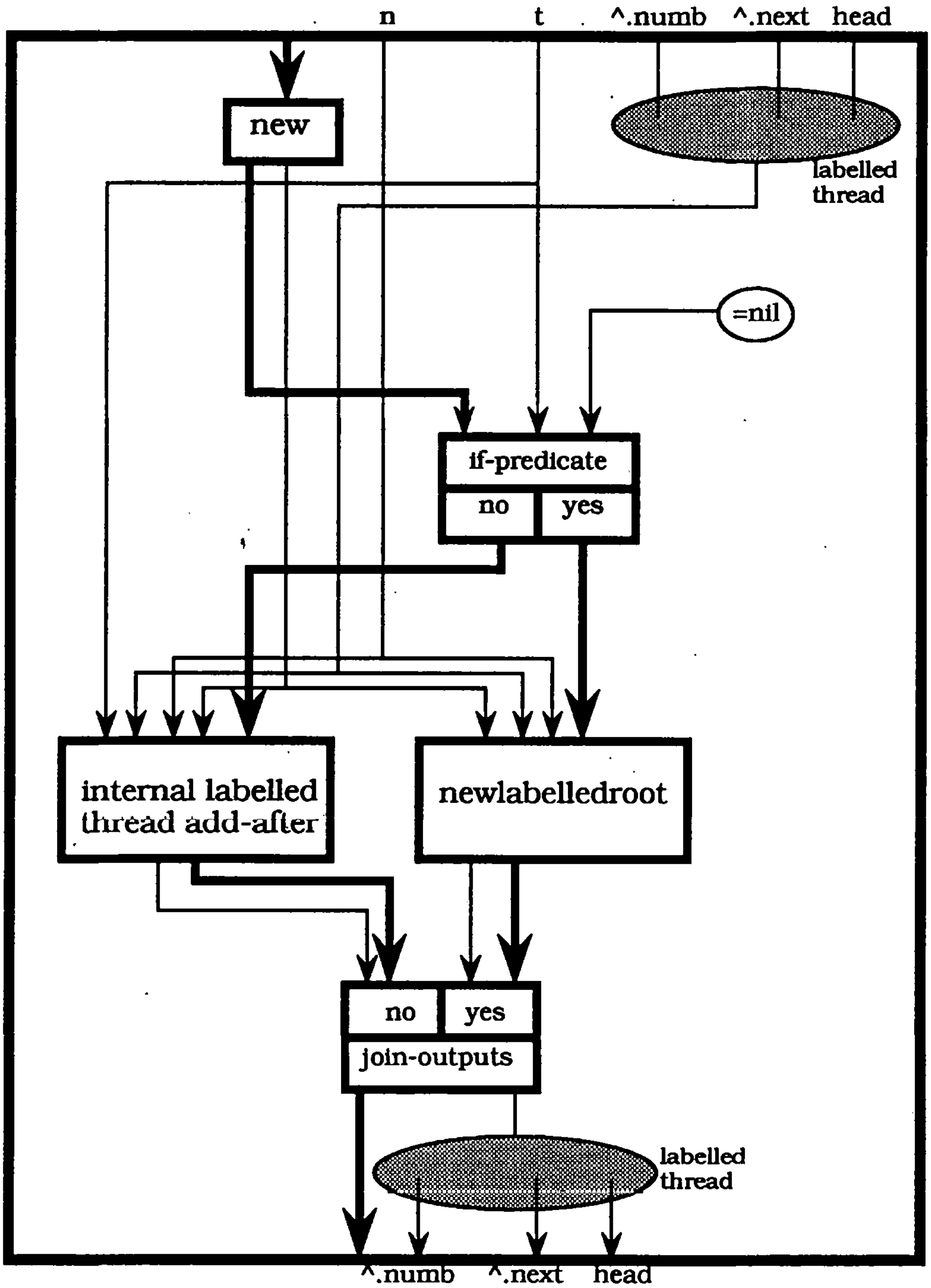


Figure 8.22
After Recognition of Internal-Labelled Thread Add-After and New Labelled Root

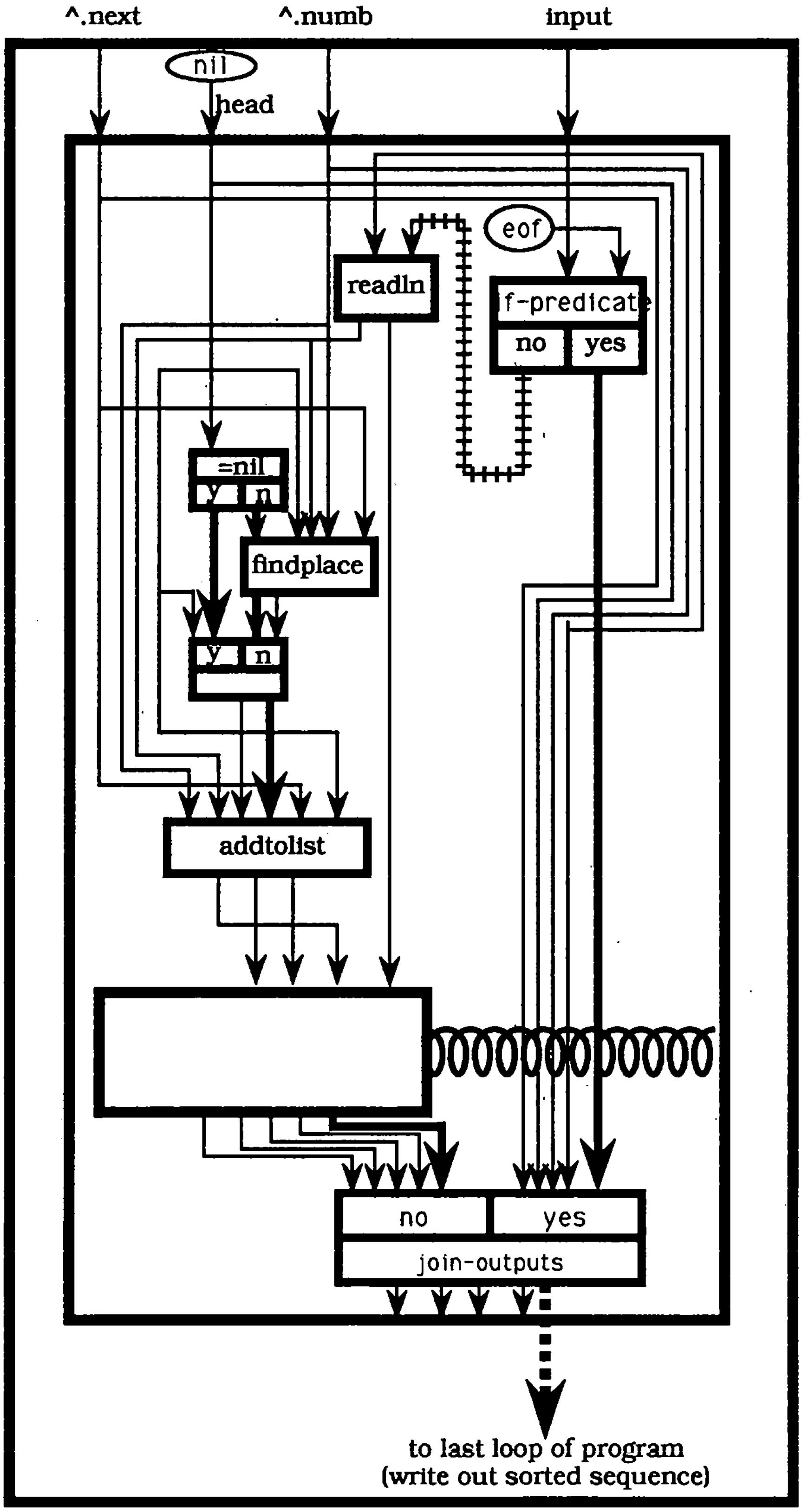


Figure 8.23
Main Program Surface Plan

a considerable improvement on just replacing procedure calls by the body of the procedure (with appropriate substitutions) as is done in Wills 1986]. When this replacement is done clearly part of the resulting diagram consists of the final diagrams for **findplace** and **addtolist** (appropriately joined). The resulting diagram (for this part of the main program's surface plan), after "replacing" cascaded joins by **n-join-outputs**, is shown in Figure 8.24.

Now, IDS has a plan in its library called **ordered-labelled-thread-insert**. This is the plan for inserting a new item in the appropriate place in an ordered list, and is shown in Figure 8.25. As implied by our discussion of structure sharing flowgraph grammars, rules which contain two nodes with the same label and the same inputs are collapsed, before being used for parsing. So the form of the rule applicable is that shown in Figure 8.26. Using this rule IDS can recognise that there is an **ordered-labelled-thread-insert** present in the program, since all the data flow requirements are met. Note that it has to match through the join to realise this. Unfortunately, IDS thinks the resulting plan is in a conditional control flow environment, since many of the operations in the plan (in particular, the **internal-thread-add-after**, and the **newroot** operations) are not in the control flow environments expected. Furthermore, the resulting controlling condition for the plan, is not reducible to true using simple propositional logic. This is because the variables in the controlling conditions representing the two "=nil" tests are different, and furthermore they do not have the same inputs, so can't be collapsed. Figure 8.27 shows the relevant parts of Figure 8.24. In order to realise that the resulting plan condition was in fact just true, IDS would have to deduce from the pre- and post conditions associated with the **internal-thread-find** that the output from the **internal-thread-find** could not be nil, and that the output from the

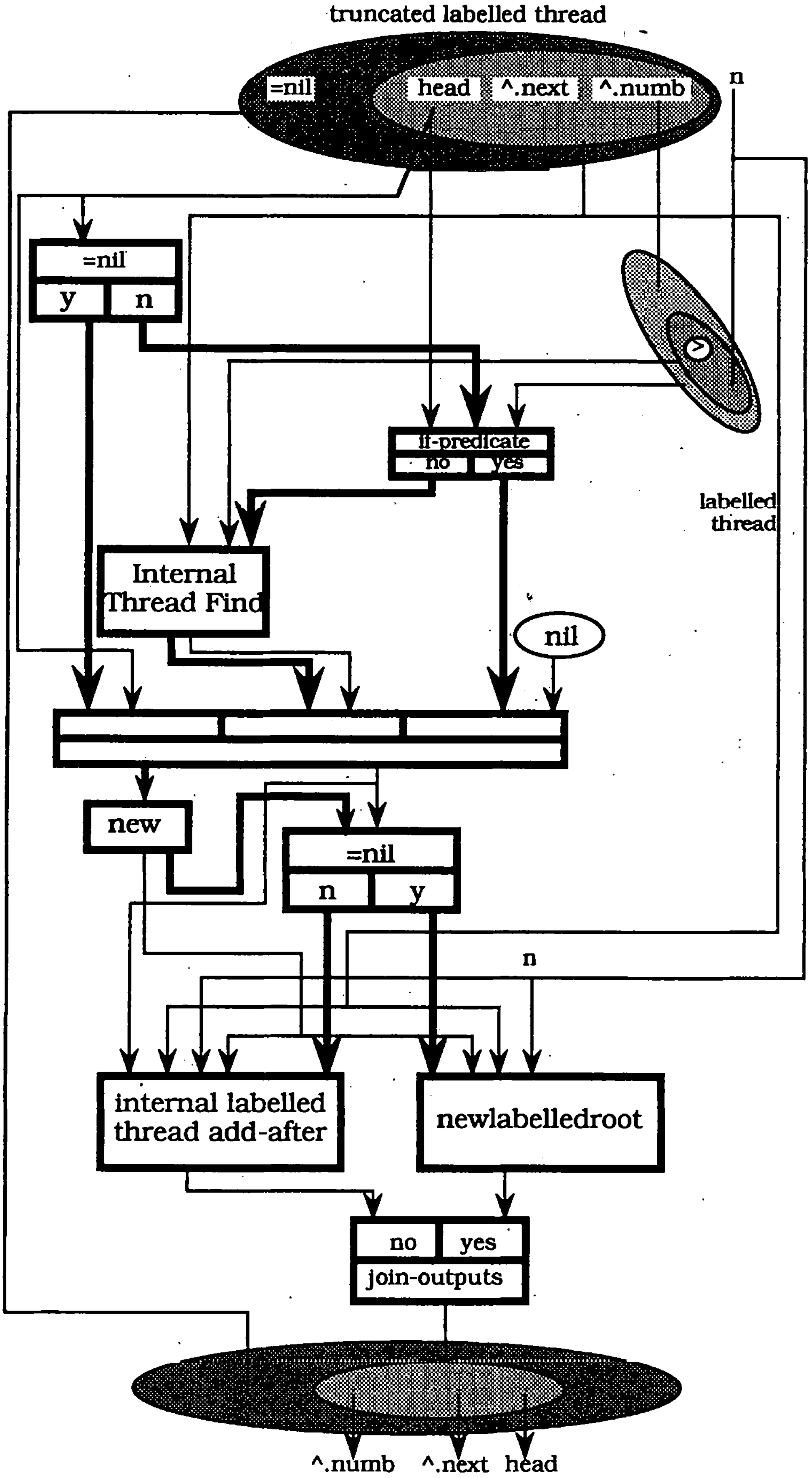


Figure 8.24
The FindPlace and Addtolist Part
of Main Program's Surface Plan

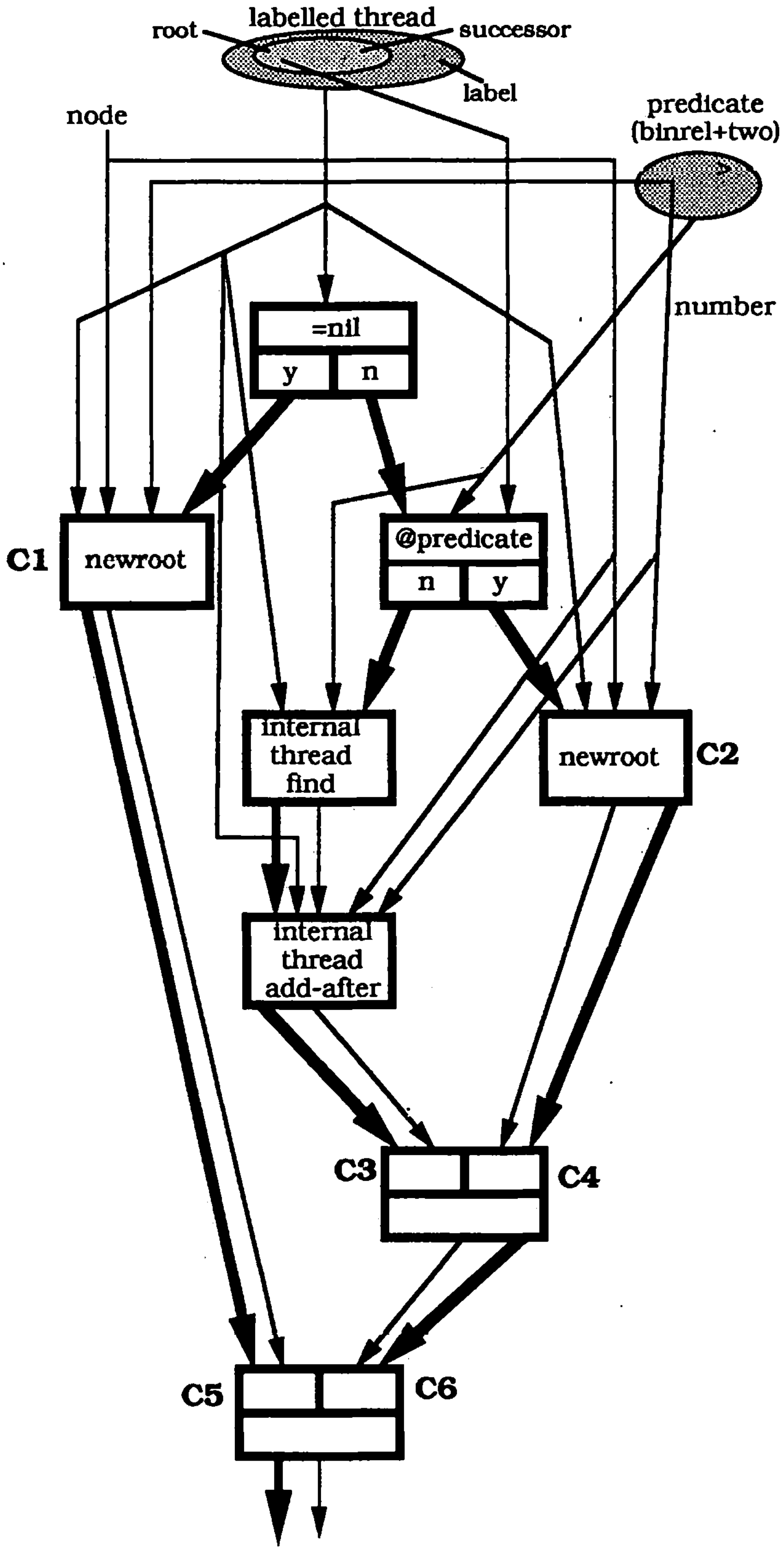


Figure 8.25
Ordered Labelled Thread Insert Plan

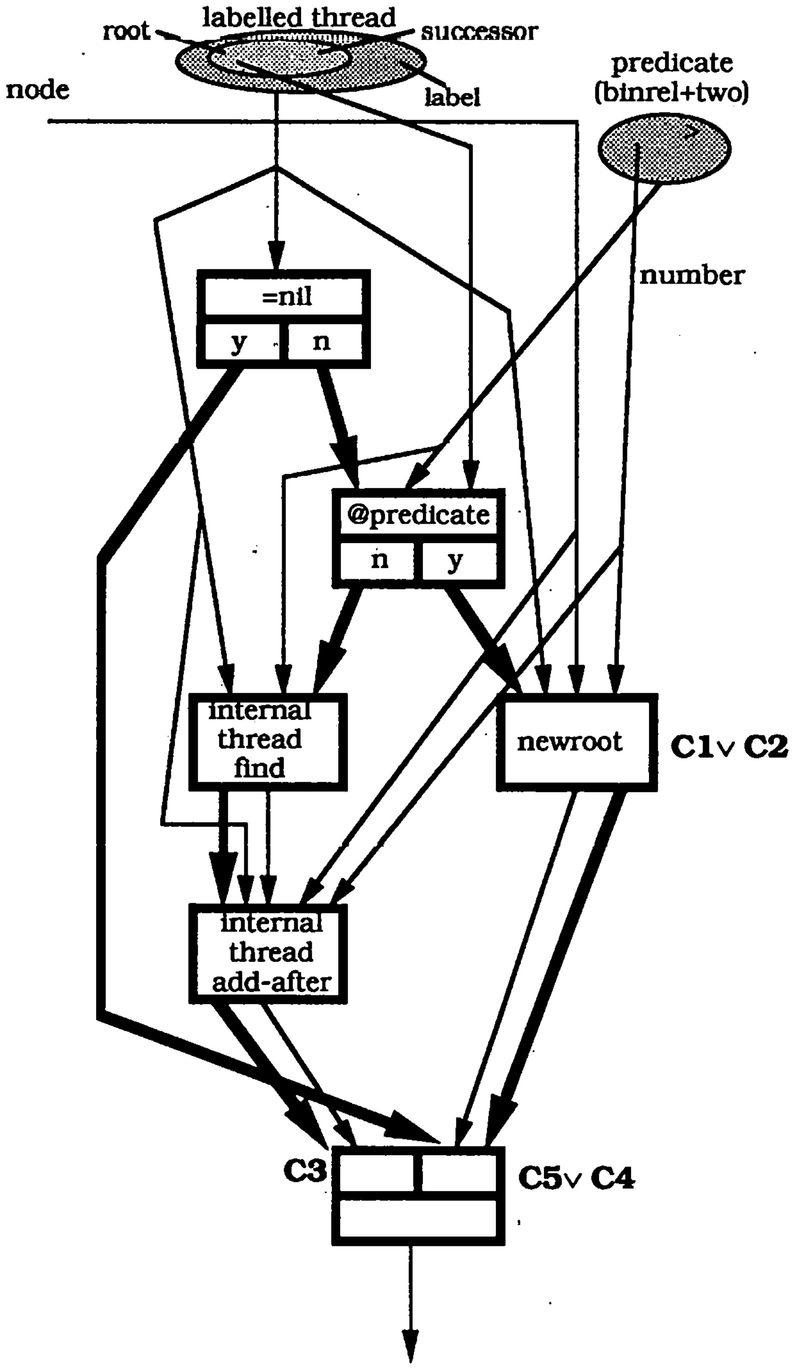


Figure 8.26
Collapsed ordered labelled thread insert plan

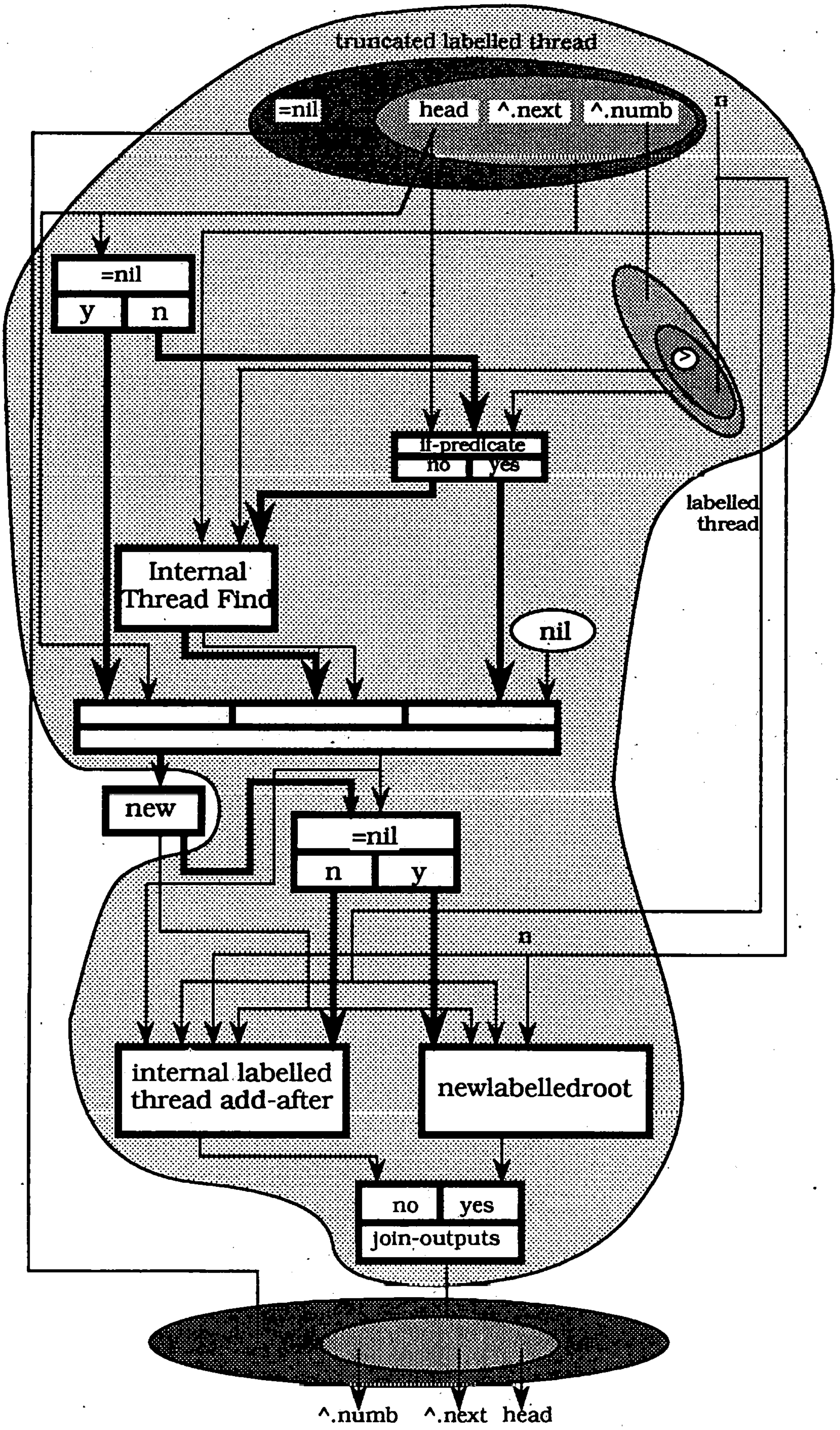


Figure 8.27
The FindPlace and Addtolist Partof Main Program's Surface Plan, Showing Ordered Labelled Thread Insert

other two branches of the conditional were always nil. Under these conditions it could assign appropriate controlling conditions to the **newroot** and **internal- thread-add-after** plans, which would result in the plan condition for the **ordered-labelled-thread-insert** being true. So this plan is recognised conditionally, and until a theorem prover is available this is the best it can do. The rest of the discussion will assume this has been done so the plan is in fact unconditional. Notice, though, how this plan actually comes from two procedure calls - parts of the cliche come from one procedure and other parts from the second. The resulting surface plan for the main program is shown as Figure 8.28. IDS now recognises two standard plans. The first of these is **terminated-iterative-read** (described above), and again this plan is recognised as implementing a **readall** operation which takes a file as input and produces a finite sequence as output. The other standard plan recognised is **iterative-ordered-labelled-thread-insert,** which IDS then uses an overlay to recognise as an implementation of a **sort** operation, temporally abstracting the items input to the **ordered-labelled-thread-insert** operation (which are the same items produced by the **iterative-read)** as the set input to the **sort**, and viewing the labelled thread output from the **iterative-ordered-labelled-thread** plan as the ordered sequence output by the **sort**. Similarly the final loop is seen to bc implementing a **writeall** operation, which takes a sequence and writes it to a file. Thus the whole program is finally represented by Figure 8.29.

As a final stage, IDS should check (and will once the theorem prover has been implemented) that all the preconditions for all the various plans and overlays involved in this analysis of the program are satisfied. If they are all satisfied then the program has been understood by IDS to the extent that its plan library allows. Of course adding more

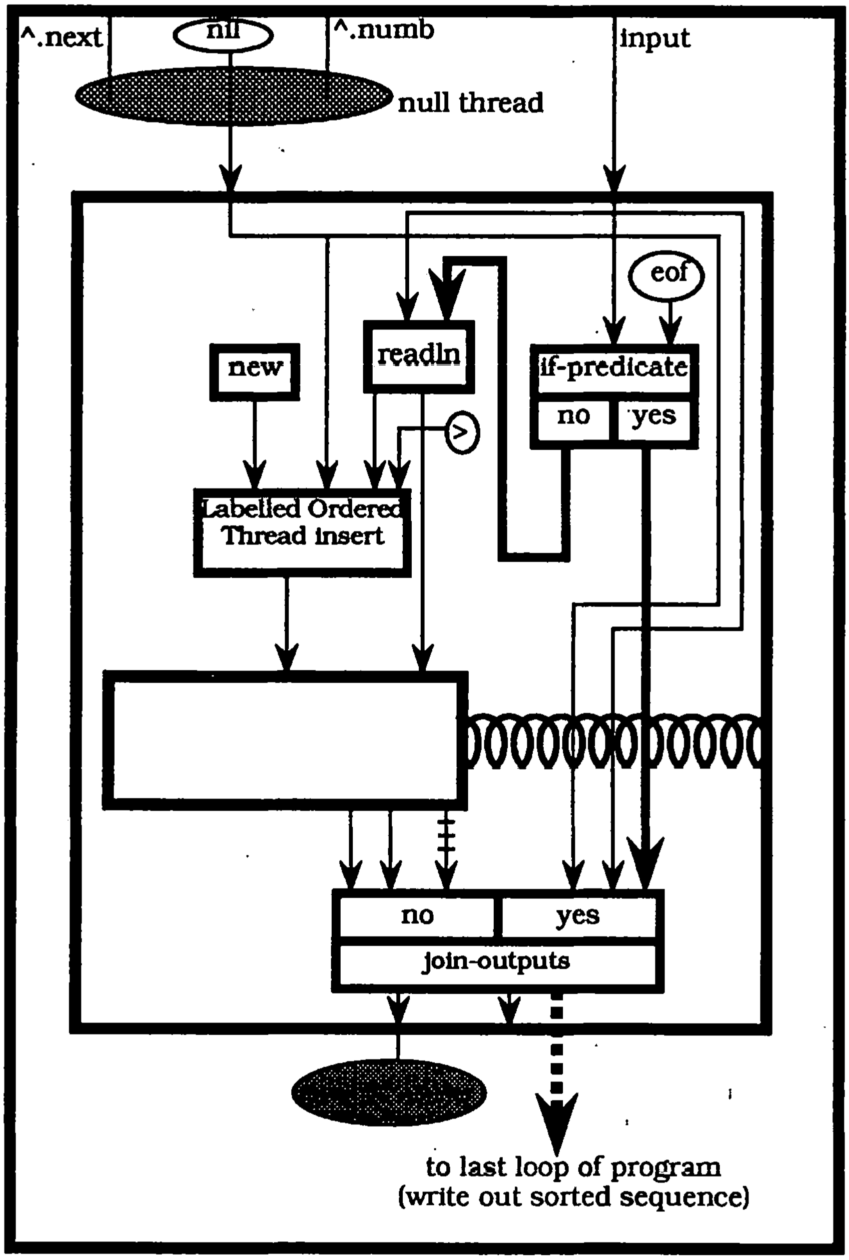


Figure 8.28
After Recognition of Ordered Labelled Thread Insert

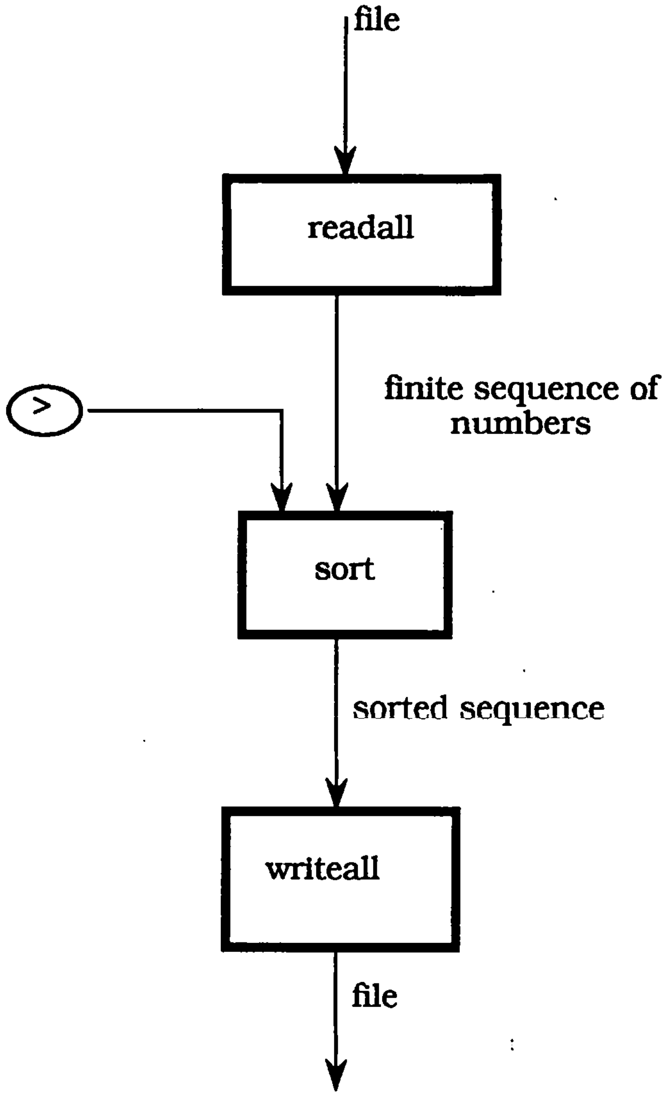


Figure 8.29
Final stage of Analysis

plans may enable it to give an even higher level description, and it is simply this ability to add plans which could enable IDS to be used in a tutoring environment since plans could always be added which encode knowledge about particular problems the student could be working on. In the case where a plan has been recognised, but its preconditions are violated in some way, then a bug (Type 1 or 3) has been located, and a debugging strategy must be invoked.

Chapter 9.

Towards a Debugging System

9.1 Debugging Programs Using Plan diagrams

As already mentioned in Chapter 6, the strategy IDS will use to debug programs is the following:

(a) Translate the program into its surface plan.

(b) Try to understand the program by recognising all occurrences of library plans, as described in Chapter 8. Make a note of any "near" matches.

(c) Symbolically evaluate any remaining (i.e. unrecognised) parts of the surface plan i.e. deduce properties of these parts of the program using the theorem prover.

(d) Check for broken preconditions of any of the recognised plans.

(e) Use near match information and broken precondition information to try and repair the program.

(f) Translate the debugged surface plan back into the source language.

This chapter will try to show how this process can be supported by use of the plan calculus and IDS's chart parser. It will also discuss the role to be played by the (as yet) unimplemented parts of the system. Note that the only bug types that can be found by this approach alone are type 1 (and type 3), and then only if the normal use heuristic is a valid assumption.

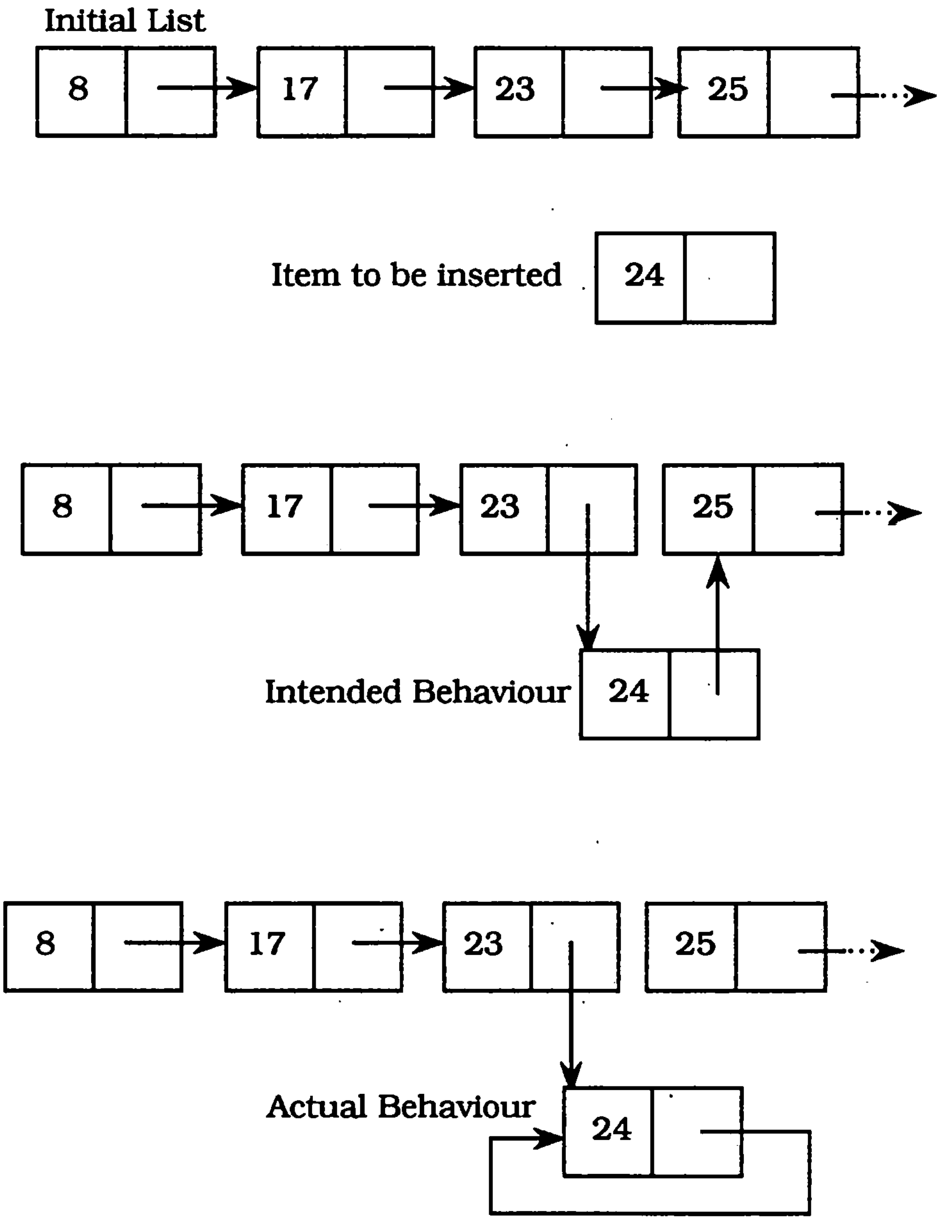


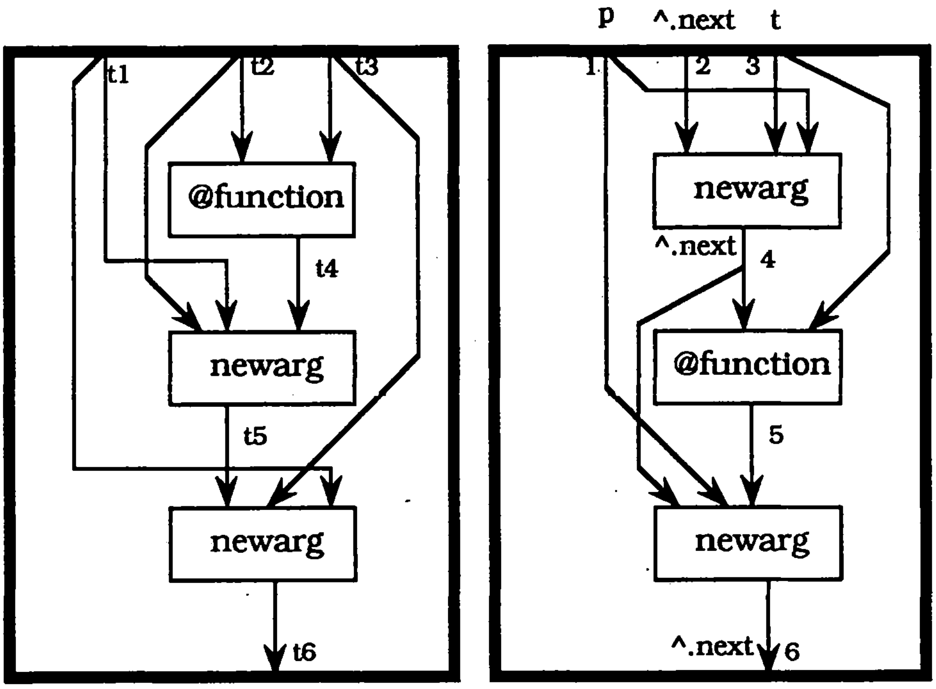
Figure 9.1
The Buggy Program's Behaviour

So suppose that, instead of being correct, the program just analysed had contained a bug, and in particular suppose that the procedure **addtolist** had been the following instead of the earlier correct version:

```
procedure addtolist(n : integer; t : plist);
var p : plist;
begin
     new(p);
     p^.numb:=n;
     if  t = nil then
            begin
                   p^.next := head;
                   head:=p;
            end
     else
            begin
                   t^.next:=p;
                   p^.next:=t^.next;
            end;
end;
```

The bug in this procedure is that the two lines responsible for 'splicing in' the new element into the list have been given in the wrong order, resulting in the kind of behaviour shown as Figure 9.1. How can IDS detect and correct this bug?

It starts by trying to understand the program as before, and of course its analysis of **findplace** is identical to that described earlier. However when it tries to analyse **addtolist** it recognises the **new-labelled-root** plan as before, but is unable to recognise the **internal-labelled-thread-add-after** plan. However, as IDS uses the generalisation of chart parsing to do plan recognition etc., it also builds up information about partial plans that it recognises. In particular it finds the partial **spliceafter** plan shown in Figure 9.2. The figure shows the partial match to **spliceafter** found by IDS, and what would be needed to make it a complete instance of **spliceafter**. This partial match is close enough for IDS to recognise it as a near miss. However at this point IDS will not categorise this as a bug - it will merely make a note of the near miss for future reference. So at this point the analysis of **addtolist** is Figure 9.3 rather than Figure 8.22 as earlier.



Correct Spliceafter

```
[spliceafter [?t1 ?t2 ?t3][?t6]] =>

[ [@function [?t2 ?t3][?t4]]
  [newarg [?t2 ?t1 ?t4][?t5]]
  [newarg [?t5 ?t3 ?t1][?t6]]
]
```

Buggy Spliceafter

```
t^.next:=p;
p^.next:= t^.next;
```

```
[ [@function [4 3][5]]
  [newarg [2 3 1][4]]
  [newarg [4 1 5][6]]
]
```

Matching process gives best partial match as:

```
[spliceafter [1 4 3][?t6]] =
  [ [@function [4 3][5]]
    [newarg [4 1 5][6]]
  ]
```

and needing to find an object matching

```
[newarg [6 3 1][?t6]]
```

to complete the match.

Figure 9.2
Partial Spliceafter Plan

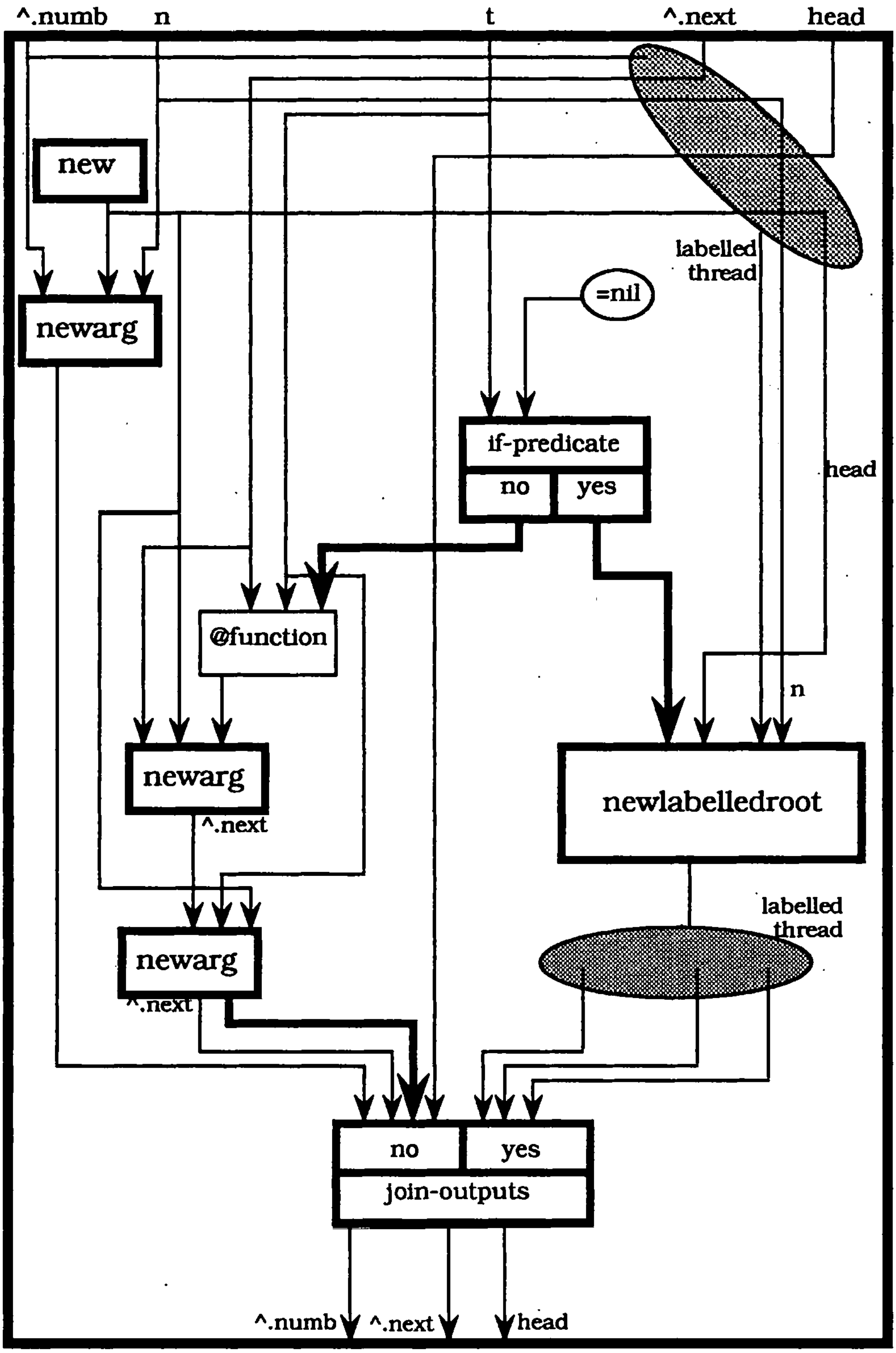


Figure 9.3
Partiallly Understood Addtolist

Now IDS begins analysis of the main program as before. When it starts analysing the concatenation of **findplace** and **addtolist** it finds that it cannot really go any further. So the situation it is in is that shown in Figure 9.4. So it will assume that it has finished its analysis, and start trying to verify all the preconditions for the various plans and overlays. As part of this process it needs to verify that the preconditions for **internal-labelled-thread-find** are satisfied. Now there are actually two places it has to check these preconditions - on entry to the loop, and on entry to its tail recursive component. This double checking corresponds to doing inductive reasoning. One of the preconditions that has to be satisfied is that the object input to the **internal-labelled-thread -ind** be a **thread**. In the outer part of the loop this is of course (trivially!) satisfied by the null **thread**. However the input to the **internal-labelled-thread-find** in the recursive part of the loop comes from **addtolist** in the outer part. So IDS has to check that ^.numb, ^.next, and head coming from **addtolist** form a **thread**. Now these values come from a **join-outputs** box, so it has to check that the values input on the success side, and on the failure side of the **join-outputs** form a **thread**. This is clearly true on the success side (they come from a **new-labelled-root** plan. However on the other side they come from the part of **addtolist** that IDS is unable to recognise. Therefore IDS symbolically evaluates the unrecognised part of the plan (the part that should be the **spliceafter** in this case) and deduces that

^.next(object1) = object1

where object1 is the input labelled p in the diagram (note that we are treating ^.next as the name of a composite function). This clearly contradicts the definition of a **thread** (part of which states that there are no cycles in a **thread**) and so IDS has located an internal inconsistency in its analysis of the program, and hence has located the

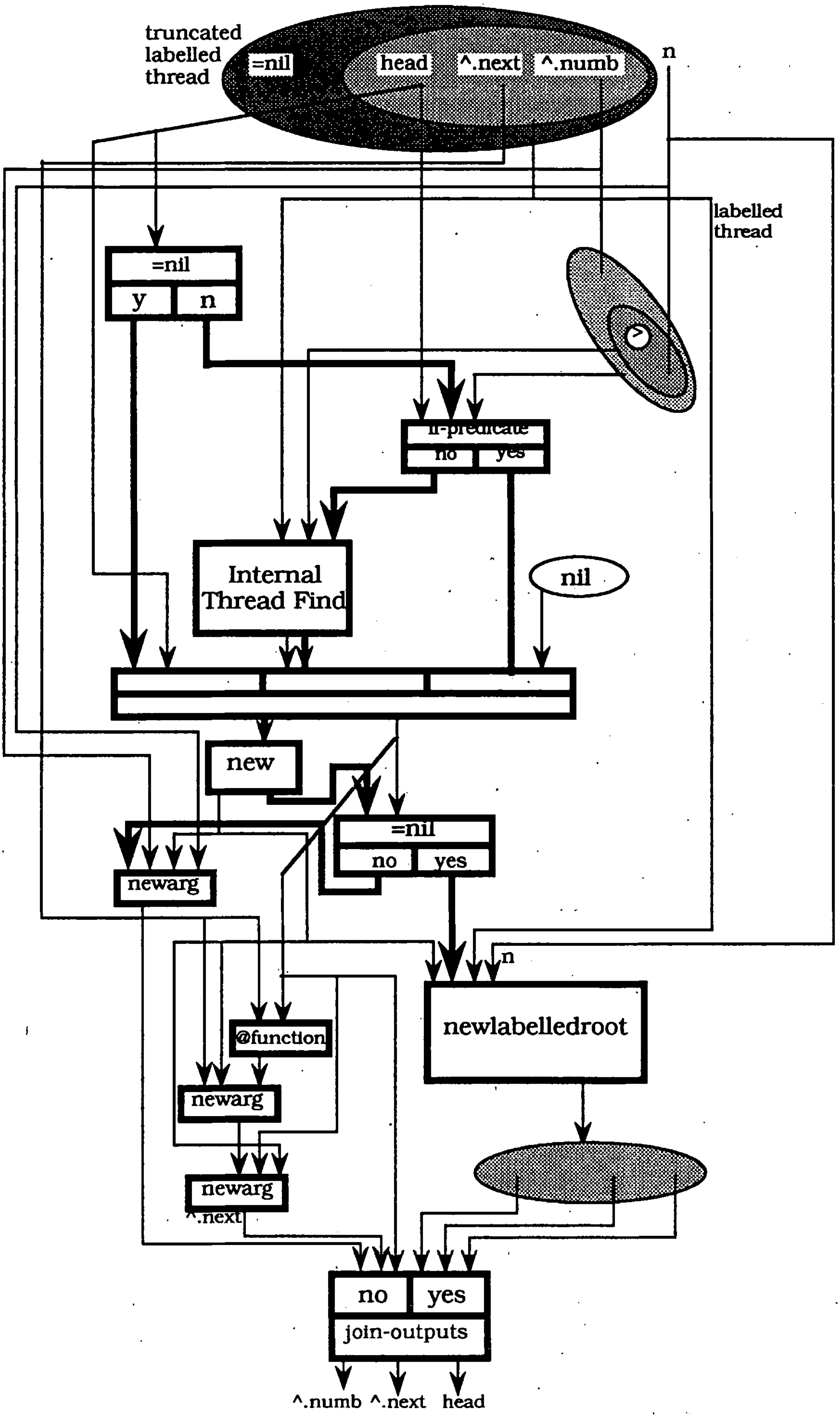


Figure 9.4
Findplace And Addtolist Part of Buggy Program

bug. Even in the case where IDS cannot now go on to suggest a "fix", pointing out where the bug lies could be of great use to the programmer. However in this case IDS should be able to actually suggest how to fix the bug. It can do this by noticing that the piece of the surface plan it can't recognise has (partly) been recognised as a partial **spliceafter** plan. In addition it notices that the best way to turn the unanalysed part of the plan into a complete **spliceafter** is to delete (refer to Figure 9.2 for the notation)

[newarg [2 3 1] [4]]

from the plan and replace it by a new **newarg** operation

[newarg [6 3 1] [7]]

where 7 is a new tie point. This would give rise to a **spliceafter** plan

[spliceafter [1 4 3] [7]].

The question then arises as to how this should be connected up to the surrounding graph in place of the original unrecognised parts of the plan. Although the question of how to do this in general is still one of the ongoing topics of research for this project, in this case it can do it by noticing that the new node 7 (the output of the plan) is replacing node 6. So it connects node 7 to everything node 6 was previously connected to. Similarly it notices that the only new input is node 4, replacing node 2. So it connects everything that was connected to node 2 to node 4. IDS then notices that, if it did this, the analysis can continue (in fact exactly as described earlier for the original correct program) and IDS would arrive at the same analysis of the program, with no broken preconditions. So it can ask the user if the program is indeed trying to sort the input numbers, and if so, can correct the plan accordingly. Note that by keeping track of the variable names associated with all the dataflows involved IDS should even be able to generate the correct code, with the right use of variable names. Note that at this stage in the development of the project no attempt will even be made to point out (or even recognise) the fact that the error was really just having two lines in the wrong order. But IDS should be able to point out the incorrect lines, and the correct versions of them. Note also that even if IDS had not had the plan for sorting in its library (and it would be unreasonable to expect it to have every possible plan for every possible program) it should still in fact be able to correct this bug, by noticing that once it has recognised the **spliceafter,** and hence the **labelled-thread-add-after** plan, it would find that the broken precondition was no longer broken. The difference would be that it would query the user about the program at a lower level - instead of asking if the program was trying to sort, it would have to ask if a particular piece of code was trying to splice a new item in at a particular place in a linked list structure. Notice that this process makes it essential to have some kind of reason maintenance system closely integrated with both the chart parser and the (unimplemented) theorem prover/symbolic evaluation module. This is because IDS will essentially hypothesise a repair based on the near-miss information. It then

(i) removes some of the original patches that were present in the chart, and

(ii) adds a complete plan (based on the near-miss information) and continues with its analysis.

If this leads to the precondition that was previously broken no longer being broken then IDS asks the user for final confirmation that it has found and repaired the bug. If the precondition is still broken, or another is broken in its stead as a result of removing or adding patches then IDS will need to retract the changes it has made to the chart (including removing any new patches added as a result of the chart parsing continuing after the near-miss was repaired). Additionally, IDS should try to avoid having to completely redo all the theorem proving it has done previously. To do both of these it really needs a reason maintenance system to keep track of which theorems are currently believed. Now, as discussed in Chapter 3, patches in the chart also represent theorems so the reason maintenance system can also be used to keep track of which patches are currently considered to be in the chart, and which are not. It would be an interesting research project in its own right to build such an integrated reason maintenance system.

The question of how to generate correct code in terms of variables the user will recognise is enormously complicated (but only slightly in the situation just described!) by the fact that the piece of amended code may have been through a long sequence of the program transformation operations. It is still an area of ongoing research as to how this can best be dealt with in general.

9.2 Duplicating PROUST

Although, this work is only now starting to get to the point where debugging can be tackled in earnest, an indication of how some of the capabilities of PROUST could be duplicated within the surface plan formalism discussed in this thesis can be obtained by considering the program below:

```
program averrain(input,output);
const
     sentinel=99999;
var
     rainfall, averrain, highrain, norain : real;
     totalrain, validrains, rainydays, count : real;
begin
     read(rainfall);
     while rainfall<0 do read(rainfall);
     write('the rainfall entered was  ', rainfall, '  inches');
     count:=0;
     highrain:=0;
     while rainfall<>sentinel do
            begin
                   if rainfall>highrain then highrain:=rainfall;
                   totalrain:=totalrain+rainfall;
                   count:=count+1;
                   averrain:=totalrain/count;
                   validrains:=count-norain;
                   rainydays:=validrains;
            end;
     writeln('the number of validrain days entered was  ',
validrains);
     writeln('the averrain was  ', averrain);
     writeln('the highrain is  ', highrain);
     writeln('the no. of rainydays in this period was  ',
rainydays);
end.
```

Clearly there are several things wrong with this program. Perhaps the major fault is that instead of being written as a single loop which processes a number, then reads the next, this program has been split into two loops, the first of which reads numbers from the input file until it reaches the first non-negative number. This number is then passed to the second while loop in the program, which since it does not itself read in any numbers, goes into an infinite loop (unless the first non-negative number happens to be the stop sentinel). However, there are other bugs in this program as well. For instance, two of the variables (norain and totalrain) used by the second loop are uninitialised. Finally, although not necessarily an error, there is code placed within the loop which for efficiency reasons would be better placed after it (e.g. computation of averrain, validrains, and rainydays). All of these can be detected using the plan representation already discussed. Figure 9.5 shows the surface plan corresponding to (a slightly simplified version of) the program above. It is quite clear from

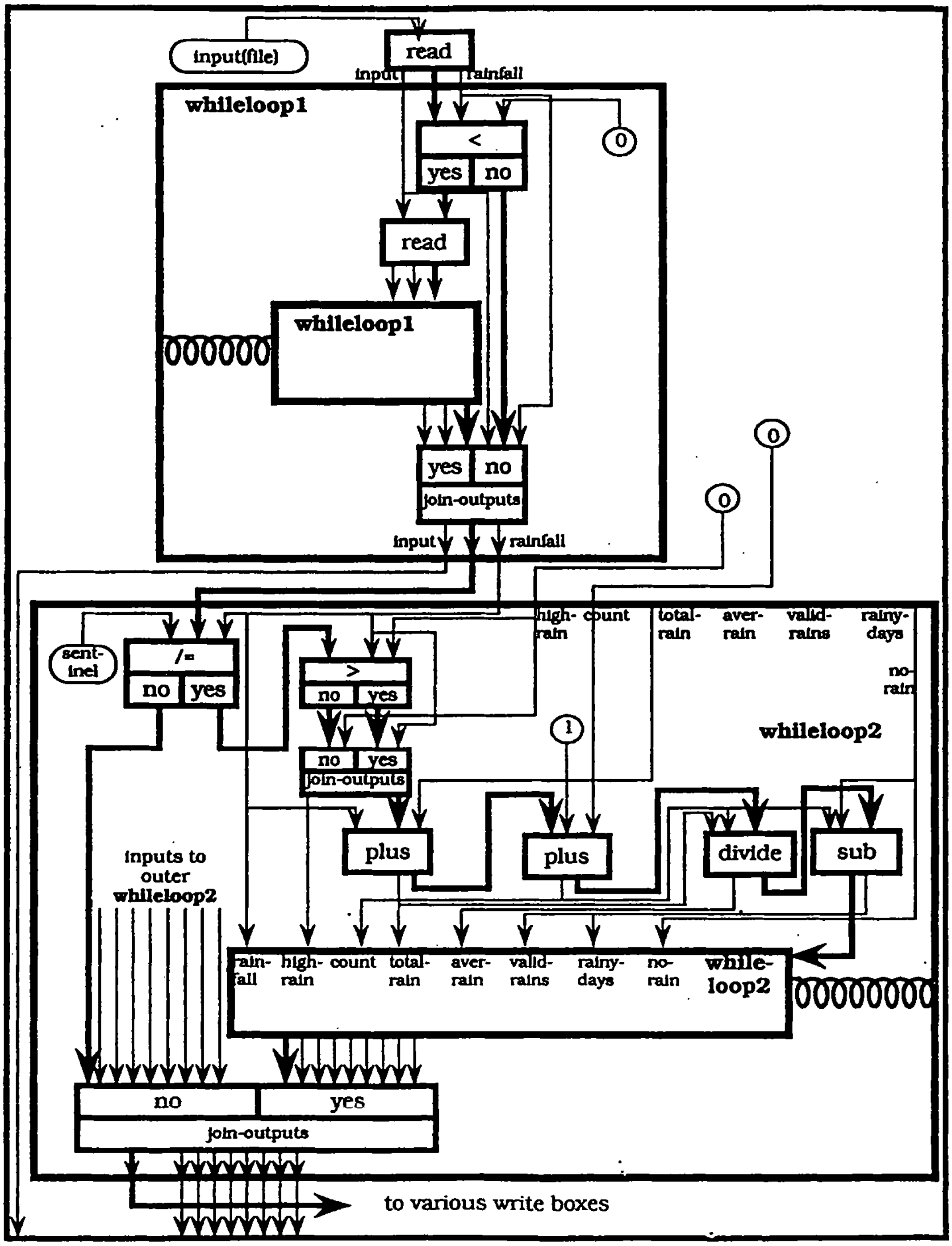


Figure 9.5
Buggy Rainfall Program Surface Plan

this that five of the inputs to the second while loop are undefined. In the case where the first positive number read by the program is the stop sentinel, all of these are connected straight through to the output of the whileloop. Therefore the whileloop is ill-defined in this case. Even if this first number is not the stop sentinel, it can be seen that totalrain and norain are used within the loop and hence these are cases of uninitialised variables. This is actually detected during the process of source code to surface plan translation. In addition averrain, validrains, and rainydays are inputs to the loop but are not used within it, merely updated. This is easily checked for, and the suggestion made that this code be moved outside the loop. The infinite loop is detected by noticing that the variable tested to see if the exit condition holds (rainfall) is passed unmodified through to the recursive call on the whileloop 2 segment. IDS can suggest correcting the program inserting code similar to that which produced the initial value passed to the loop. Certainly this works for this particular program, and part of the ongoing research on this project is into finding a suitable set of heuristic rules for this kind of bug. It may be that this is where IDS will be able to make use of the Yale group's bug categorisation.

PROUST also managed to recognise that this program was attempting to compute the number of positive numbers by subtracting the number of zeros from the number of non-negative numbers. It managed to do this by having a lot of problem-specific knowledge. Clearly in a general purpose system this kind of knowledge is not available. However if it finds out from the programmer (or elsewhere) that the program is trying to compute the number of positive numbers, and if we include in the plan library a plan representing the fact that in general one can compute the number of items in a set which satisfy some property by counting the items which do not satisfy the property and subtracting from the total number of items in the set, then the system would be able to suggest a correction to this part of the code in Program 2.

Chapter 10.

Conclusions and Future Work.

10.1 What has been achieved?

This thesis has been motivated by a wish to build an intelligent general purpose, domain-free debugging system for Pascal programs, and has presented the case that, in order to do this, program understanding is necessary. Looking at other systems for debugging programs, in particular Johnson's [1986] system PROUST, and Murray's [1986] system TALUS it is clear that the strengths of other systems come from an ability to do plan recognition and general purpose reasoning, and their weaknesses come from an inability to do one or other of these. The plan calculus offers the ability to combine both of these as it provides a framework for both general purpose reasoning using a theorem prover, and graph-based plan recognition, in a way that is probably as syntax independent and language independent as possible. However, when it came to actually trying to use the plan calculus, in particular to do the graph based plan recognition, it became clear that this is an enormously hard problem (reflecting the fact that programs are extremely hard objects to analyse).

As a result, the work described in this thesis is only now getting to the point where serious research into its applicability to program debugging can start. In particular, once an integrated theorem prover, along the lines of Rich's [1985] CAKE system, has been built then its potential can seriously be investigated. However, along the way, several obstacles to using the plan calculus have been overcome. This chapter will therefore begin by summarising what has been achieved.

The first achievement, and main one, is that we have developed a new polynomial time algorithm for doing bottom-up, or top-down, analysis of surface plans. This algorithm has been presented in the framework of restricted structure sharing flowgraph grammars that we have developed for this purpose.

Secondly, we have developed some theory, in particular that of deterministic operations in the plan calculus, and of generalised control flow environments, and their associated controlling conditions, that enables us to justify the use of restricted structure sharing flowgraph grammars as a suitable framework for doing plan recognition. As part of this it has also become clear that many of Rich's plans are expressed in too restrictive a form (in particular his use of cflow constraints everywhere), and we have developed a distinction between various types of control flow constraint that enable one to be more expressive, and only impose the restrictions that are strictly necessary. If this had not been done the parser would either not have found plans that really were present, or would have found plans not justified by the plan calculus, thus breaking the tight coupling of what the parser finds and what a theorem prover could have proved.

Thirdly, we have adapted the above algorithm to cope with various parts of the plan calculus that do not exactly fit into the framework we have developed. In particular, techniques have been developed that enable it to cope with plans involving data plans and data overlays.

Fourthly, we have identified various new plans that are essential if these modifications are to work properly and enable plans that we recognise to connect up properly for reasoning about side-effects.

Additionally, we have written a program (structured in a similar way to a recursive-descent parser for Pascal) which can translate Pascal programs into their surface plans, thus enabling them to be analysed within the framework of the plan calculus.

Finally, we have written a program for translating plans and overlays, expressed in Rich's compact notation, into suitable rules for the parser to use.

As we are ultimately interested in program debugging, it was also an interesting question as to what kinds of bugs could reasonably be expected to be found by a system with no knowledge of the programmer's intentions, and no knowledge of the domain, but possessing general programming knowledge coded as plans in the library. Accordingly, we have outlined a new bug classification in order to try state precisely which bugs can be located and (sometimes) fixed by a such a system which knows nothing about the purpose of a program. Additionally, we have presented some preliminary work on exactly how such a debugging system might go about locating and fixing such bugs.

10.2 Outstanding Problems

As stated, it has been a major aim of this work to try and ensure that there is a tight coupling between what the parser does, and what can be deduced from the plan calculus, and in this it has been partially successful. However, there are a variety of places where this goal, although intuitively satisfied, has not been theoretically demonstrated. In particular, the way in which data overlays and plans have been dealt with, although seeming to work, does not fit the grammatical formalisms we have developed, and as is the way with grammatical formalisms, it is always possible that some strange interaction might occur, although at the moment this seems unlikely.

Furthermore, it seems likely, even if we could more formally justify what has been done, perhaps by developing some even more general grammatical framework capable of expressing data plans and overlays, and their relationship to temporal plans and overlays more naturally, that we may have reached roughly the limits of what can be done by a pure graph based parser. To some extent we have already deviated from pure graph parsing in our use of plan conditions etc, but at least we have kept the added machinery to simple propositional logic (and even this introduces NP-completeness into what was previously a polynomial algorithm). However it seems likely that even to do the pure data flow parsing required, in the presence of data overlays and so on, may well require more general theorem proving abilities. For example, more reasoning about the entries in the data-overlay database will probably be required, as there can be complex relationships between the entries, which if not realised, can mean that one fails to detect plans that are present. For example, if one has in the data overlay data base the fact that one predicate p is the complement of q, then one also knows that q is the complement of p. At the moment this is dealt with by adding such an assertion at the time the first assertion is added, but, realising the relationships between p and q, where p=**binrel+two->predicate**(q,t), complement(p), and **binrel+two->predicate**(complement(q),t)) to take quite a simple example, is currently beyond the system, and can sometimes lead it into trouble, in the sense that it will not realise that new tie-point it is trying to create has already been created. This can mean that plans which do in fact connect up, appear not to. Now, it may turn out that we can express these relations via a grammar of some kind. If so this would be a very nice result. Otherwise, without a general theorem prover capable of proving that two apparently distinct tie-points are actually the same, many plans will not be recognised. In general it seems that there are quite a lot of places in the parsing process where a theorem prover might need to be invoked.

Another area where more work needs to be done is on the sort of problem encountered in the sorting program in Chapter 8, where a plan was recognised, but with a condition attached, which was in fact guaranteed to be true. Part of the problem may be soluble without a general purpose theorem prover if we introduce some sort of notion of conditional collapsing. This would be some sort of collapsing operation done only when matching through joins resulted in the realisation that, under the condition implied by the branch of the join matched through, two operations were collapsible, resulting in a more complex condition on the resulting collapsed operation.

A second area where more work needs to be done is in the area of program transformations, expressed as graph transformations. At the moment these are hedged about with criteria that must be satisfied before they are applicable. For example, **iterative-flag-test-removal**, discussed in Chapter 7, requires that there be no actions preceding the flag test in a loop to which we intend to apply the transformation. This actually requires the recogniser to search the graph to make sure this is true. Although not too hard to do, this does not seem to fit in with the spirit of context free-graph parsing, since we now essentially have a context-sensitive operation. Such operations may turn out to be necessary, but it would be nice if they could be avoided. It may turn out that in these situations the theorem prover would come into its own, by deducing the existence of actions in different control flow environments from the ones in which they actually turn up. If these deductions are added to the chart by the theorem prover, then the chart parser could recognise the plans. However, it is likely that this will be too inefficient, so probably some graph parsing approach will be needed. An alternative solution to this problem would be to adopt something like the PROUST [Johnson 1986] approach which stores lots of variants of each plan, but the likely combinatorial explosion, especially if there are thousands of plans, make this seem unlikely as a satisfactory approach.

10.3 Future Research

Of course much remains to be done to complete this work. In particular it needs the addition of the (by now often mentioned!) powerful theorem prover and reason maintenance system, and much more work needs to be done on heuristics for how to reconnect replacement bug-fixes to the surrounding graph. It would be interesting to attempt to use multiple near-miss information as a guide in this process. For instance, if we know there is a bug at some point in the program, and there is a near-miss to some plan A at that point, and part of what is wrong with A is that it requires a plan B, but cannot find one, and there is also a near-miss to a plan B at that point then this mutual information ought to help both confirm the hypothesis that a B is required, but ought also to aid the system in working out how to reconnect a repair to B to the surrounding graph.

It would also be interesting to look at the various bug classifications, and repair strategies, developed by other researchers to see if these are meaningful in flowgraph terms. If so many of these could perhaps be adopted. In particular it would be interesting to try and build a tutoring system (perhaps rather like a combination of a high-level PROUST and a high-level TALUS) based on this work.

In connection with this suggestion, it should be noted that the system is now almost at a point where it could be given (by a tutor) a solution to a problem students are supposed to be working on. The system could then analyse this in terms of its plan library, and extract a high-level description of what the program does, in terms of high-level operations, and their inter-relationships. This could just be added to the system as a rule, like any other in its plan library. When the system tries to understand student programs, it could then use a combination of top-down and bottom-up recognition to try and recognise the rule. Because of the system's (potential) ability to invoke a theorem prover when confronted with novel code, it should be able to gain some of the strengths of TALUS in being able to deal with programs containing such code. On the other hand, its ability to recognise plans should enable it to deal with much larger and more complex problems than those TALUS can deal with. Furthermore, it should potentially be much more able to deal with programs that work by side-effect, and it doesn't take students long before they start using such programming techniques, once they are introduced to almost any kind of data structuring, unless they are severely restricted by the language.

However, tutoring novices is likely to necessitate something like PROUST's approach. As discussed earlier in this thesis, the approach to debugging that we have adopted requires what we called the Normal Use Heuristic. This was:

> If a programmer uses a standard plan in a program then, as a first hypothesis, assume they are using it deliberately in order to achieve the results of operations commonly implemented by that plan, and that therefore the preconditions for these operations should be met.

For more experienced programmers this is probably quite a reasonable assumption. However, for complete novices this is almost certainly a really wild assumption. Apart from the fact that novices produce such bizarre code [Johnson et al. 1983, Soloway et al. 1981] that sometimes, without a deep understanding of the misconceptions underlying the production of such code, it is almost impossible to recognise the intent behind it, thcy also often misunderstand what a **common programming** technique actually does, so plans turn up in totally inappropriate places. In this kind of situation, there is probably very little alternative to some sort of bug catalogue, with appropriate "canned" repairs, and advice to the students. It is not yet clear how many of the bugs in PROUST's bug catalogue can easily be expressed in flowgraph terms. If it turns out that they can, then the chart parser can simply use these as yet another set of rules to parse with.

Another interesting direction to go from here would be to try and build a system capable of finding, and possibly repairing bugs of type 2 and 4. This could involve interacting with the programmer, and perhaps linking the system to an existing (non-intelligent) interactive debugger to enable the system to set breakpoints, examine variables etc. This could perhaps sometimes replace theorem proving in an attempt to locate errors.

Yet another area that would be interesting to apply this work to is that of tutoring digital circuit design. Here the structure sharing flowgraph formalisms fit very closely onto the domain. Furthermore, it may be possible to develop something like a plan calculus for this domain, with overlays stating (say) that an addition circuit corresponds to a high-level operation like adding two numbers, with data-overlays stating that the connection between the (say) 8 input lines representing one of the numbers to be added is given by something like an **8-bits->integer** overlay.

10.4 Conclusions

Despite the above mentioned problems, the plan calculus seems to offer more power to analyse and reason about programs than any other knowledge representation technique currently available. The ability not only to reason about side effects, but to reason about programs at different levels e.g. to switch from record cells to sets, or to trees, and yet still be able to realise how changes to the structures involved at one level of description affect the other level of description, is unique, and yet fundamental to the way in which experienced programmers seem to think about programs. It is hoped that this thesis takes us a little closer to actually being to use the plan calculus as the basis for automated tools to aid programmers in the task of debugging, and ultimately in all aspects of the programming process.

References

Adam A. and Laurent J. (1980) LAURA A System to Debug Student Programs Artificial Intelligence 15, pp. 75-122.

Aho, A. and Ullman, J.D. (1977) Principles of Compiler Design, Addison-Wesley, Reading, Mass.

Barstow D.R. (1979) Knowledge-Based Program Construction North Holland. New York.

Bobrow, D.G. and Winograd, T. (1977a) An Overview of KRL, Cognitive Science 1, pp. 3-46.

Bobrow, D.G. and Winograd, T. (1977b) Experience with KRL-0: One Cycle of a Knowledge Representation Language, Proc. 5th Int. Joint Conference on AI, MIT, (vol. 1), pp.213-222.

Brotsky, D.C. (1984) An Algorithm for Parsing Flow Graphs. Technical Report AI-TR-704 MIT Artificial Intelligence Laboratory.

Chow A.L., Rudmik R. (1982) The Design of a Data Flow Analyser Proc SIGPLAN '82 Sympoium on Compiler Construction. Boston Mass. USA.

Delaney, W.A. (1966) Predicting the Costs of Computer Programs. Data Processing Magazine 32.

Della Vigna, P. and Ghezzi, C. (1978) Context Free Graph Grammars. Information and Control 37, pp. 207-233.

Domingue, J. (1987) ITSY, An Automated Programming Advisor. Tec. Report No. 22, Human Cognition Research Laboratory, Open University, Milton Keynes, U.K.

Ducasse, M. and Emde, A.M. (1988) A Review of Automated Debugging Systems: Knowledge, Strategies, and Techniques. Proc. 10th International Conference on Software Engineering, Singapore, April 1988.

Earley J. (1970) An Efficient Context-Free Parsing Algorithm. CACM 13(2) pp.94-102.

Ehrig H. (1979) Introduction to the Algebraic Theory of Graph Grammars (A Survey). Graph Grammars and their Application to Computer Science and Biology. (eds. Claus, V., Ehrig, H. and Rozenberg, G.) Lecture Notes in Computer Science, Springer-Verlag.

Ehrlich K., Soloway E. (1982) An Empirical Investigation of the Tacit Plan Knowledge in Programming Research Rep. No. 236 Yale Univ. Dept. Comp. Sci.

Eisenstadt M., Laubsch J. (1981) Domain Specific Debugging Aids for Novice Programmers Proc. 7th Int. Joint Conf. on Artificial Intelligence (IJCAI-81). Vancouver BC, Canada.

Eisenstadt M., Laubsch J. (1982) Using Temporal Abstraction to Understand Recursive Programs Involving Side Effects Proc. American Assoc. of Artificial Intelligence (AAAI-82).

Eisenstadt M., Laubsch J., Kahney H. (1981) Creating Pleasant Programming Environments for Cognitive Science Students Proc. 3rd Annual Cognitive Science Conference, Berkeley CA, USA.

Elsom-Cook, M. (1985) Design Considerations of an Intelligent Tutoring System for Programming Languages. Ph.D. Thesis, University of Warwick.

Faust G.G. (1981) Semiautomatic Translation of Cobol into Hibol MIT Laboratory for Computer Science MIT/LCS/TR-256.

Feder, J. (1971) Plex Languages. Information Sciences, Vol. 3 (1971) pp. 225-241

Floyd R.W. (1971) Toward Interactive Design of Correct Programs IFIP 1971.

Fu, K.S. (1974) Syntactic Methods in Pattern Recognition, New York: Academic Press.

Gerhart S.L. (1975) Knowledge About Programs: A Model and Case Study Proc. Int. Conf. on Reliable Software 1975.

Goldstein I.P. (1974) Understanding Simple Picture Programs PhD. Thesis MIT AI Lab. Technical Report 294.

Goldstein I.P., Roberts R.B. (1977) NUDGE, A Knowledge-Based Scheduling Program Proc. 5th Int. Joint Conf. on Artificial Intelligence (IJCAI-77). Cambridge Mass., USA.

Gonzalez, R.C. and Thomason, M.G. (1978) Syntactic Pattern Recognition: An Introduction. Addison-Wesley.

Green, C. (1969). Theorem Proving by Resolution as a Basis for Question-Answering Systems. Machine Intelligence 4, Michie, D. and Meltzer, B. (eds.) Edinburgh University Press.

Hasemer, T. (1983) An Empirically-Based Debugging System for Novice Programmers. Tech. Report No. 6, Human Cognition Research Laboratory, Open University, Milton Keynes, U.K.

Hayes, P. (1979) The Logic of Frames. Frame Conceptions and Text Understanding, Metzing, D. (ed.), de Gruyter, pp. 46-61.

Hecht M.S. (1977) Flow Analysis of Computer Programs Elsevier North-Holland Inc. New York.

Hewitt C., Smith B. (1975) Towards a Programming Apprentice IEEE Trans. on Software Engineering 1, 1 pp. 26-45.

Johnson, W.L. (1986) Intention-Based Diagnosis of Novice Programming Errors. Pitman(London).

Johnson, W.L., Soloway, E., Cutler, B., and Draper, S. (1983) Bug Catalogue I Technical Report YaleU/CSD/RR #286 Dept. Comp. Sci. Yale University.

King J.C. (1976) Symbolic Execution and Program Testing CACM 19:7 July 1976

Laubsch J., Eisenstadt M. (1980) Towards an Automated Debugging Assistant for Novice Programmers Proc. Artificial Intelligence and Simulated Behaviour Conference (AISB-80) Amsterdam.

Lukey F.J. (1978) Understanding and Debugging Simple Computer Programs PhD Thesis, University of Sussex.

Lutz, R.K. (1984a) Towards an Intelligent Debugging System for Pascal Programs: A Research Proposal. Open University Human Cognition Research Laboratory Technical Report No. 8 April 1984.

Lutz, R.K. (1984b) Program Debugging by Near-Miss Recognition and Evaluation. Proc. ECAI 1984.

Lutz, R.K. (1986) Diagram Parsing - A New Technique for Artificial Intelligence. CSRP-054, School of Cognitive and Computing Sciences, University of Sussex.

Lutz, R.K. (1989a) Chart Parsing of Flowgraphs. Proc. 11th Joint Int. Conf. on AI. Detroit, USA.

Lutz, R.K. (1989b) Debugging Pascal Programs Using a Flowgraph Chart Parser. Proc. 2nd Scandinavian conference on AI, Tampere, Finland.

Lutz, R.K. (1991) Plan Diagrams as the Basis for Understanding and Debugging Pascal Programs. in Eisenstadt, M., Rajan, T., and Keane, M. (Eds.) Novice Programming Environments. London: Lawrence Erlbaum Associates (in press).

Manna Z. (1974) Mathematical Theory of Computation McGraw-Hill.

Manna Z., Waldinger R. (1979) Synthesis: Dreams => Programs IEEE Trans. on Software Engineering SE-5, 4.

McCarthy, J. and Hayes, P. Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence 4, Michie, D. and Meltzer, B. (eds.) Edinburgh University Press.

McGregor, J.J. (1982) Backtrack Search Algorithms and the Maximal Common Subgraph Problem. Software-Practice and Experience 12, pp. 23-24 (1982).

Miller, M.L. and Goldstein, I.P. (1977) Overview of a Linguistic Theory of Design. MIT Artificial Intelligence Laboratory AI Memo No. 383A.

Murray, W.R. (1986) Automatic Program Debugging for Intelligent Tutoring Systems. Doctoral Dissertation, Artificial Intelligence Laboratory, The University of Texas at Austin. June 1986.

Persch G., Winterstein G. (1978) Symbolic Interpretation and Tracing of PASCAL Programs Proc. 3rd Int. Conf. Software Eng. 1978.

Pfaltz, J.L., and Rosenfeld, A. (1969) Web Grammars. Proc. IJCAI 1, pp. 609-619.

Quillian, M.R. (1968) Semantic Memory. Semantic Information Processing, Minsky, M. (ed.), MIT Press, Cambridge, MA.

Renner, S. (1982) Location of Logical Errors on Pascal Programs with an Appendix on Implemantation Problems in Waterloo PROLOG/C. Knowledge Based Programming Assistant Project, University of Illinois at Urbana-Champaign Technical Report UIUCDCS-F-82-896.

Rich C. (1981) Inspection Methods in Programming MIT Artificial Intelligence Laboratory AI-TR-604.

Rich, C. (1985) The layered architecture of a system for reasoning about programs. Proceedings IJCAI-85, Los Angeles, CA. pp. 540-546.

Rich, C., Schrobe H. (1978) Initial Report on a Lisp Programmer's Apprentice IEEE Trans. on Software Eng. SE-4:6, pp. 450-467.

Rosenfeld, A. and Milgram, D.L. (1972) Web Automata and Web Grammars. Machine Intelligence 7 pp.307-324 (eds. Meltzer, B. and Michie, D.) Edinburgh University Press.

Ruth G.R. (1973) Analysis of Algorithm Implementations PhD Thesis MIT.

Ruth G.R. (1976) Intelligent Program Analysis Artificial Intelligence 7, pp. 65-85.

Schneiderman, B. (1980) Software Psychology. Human Factors and Information Systems. Winthrop Publishers, Inc.

Schneiderman, B. and Mayer, R. (1979) Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results. International Journal of Computer and Information Sciences, 8(3) pp. 9-23.

Schrobe H.E. (1979) Dependency Directed Reasoning for Complex Program Understanding MIT Artificial Intelligence Laboratory AI-TR-503.

Schrobe H.E., Waters R.C., Sussman G.J. (1979) A Hypothetical Monologue Illustrating the Knowledge Underlying Program Analysis MIT Artificial Intelligence Laboratory AI Memo 507.

Shapiro, D.G. (1978) Sniffer: A System that Understands Bugs. MIT Artificial Intelligence Laboratory. AI Memo 459.

Shapiro, E. (1982) Algorithmic Program Debugging. MIT Press, Cambridge, Mass.

Soloway E., Bonar J., Woolf B., Barth P., Rubin E., Ehrlich K. (1981) Cognition and Programming: Why Your Students Write Those Crazy

Programs Proc. National Educational Computing Conference (NECC-81) pp. 206-219.

Soloway E., Ehrlich K., Bonar J. (1982) Cognitive Strategies and Looping Constructs: An Empirical Study Research Rep. Yale Univ. Dept. Comp. Sci.

Soloway E., Ehrlich K., Bonar J., Greenspan J. (1982) What Do Novices Know About Programming Research Rep. No. 218 Yale Univ. Dept. Comp. Sci.

Soloway E., Rubin E., Woolf B., Bonar J., Johnson W.L. (1982) MENO-II: An AI-Based Programming Tutor Research Rep. No. 258 Yale Univ. Dept. Comp. Sci.

Spohrer, J.C., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., Johnson, L., and Soloway, E. (1985) Bug Catalogue: II, III, IV. Tech. Rep. YaleU/CSD/RR #386. Dept. Comp. Sci. Yale University.

Sussman, G.J. (1978) Slices at the Boundary Between Analysis and Synthesis. Artificial Intelligence and Pattern Recognition in Computer Aided Design (ed. Latombe) North-Holland.

Thompson H. and Ritchie, G. (1984) Implementing Natural Language Parsers. Artificial Intelligence: Tools, Techniques, and Applications pp.245-300 (eds. O'Shea, T. and Eisenstadt, M.) Harper and Row.

Waters R.C. (1978) Automatic Analysis of the Logical Structure of Programs MIT Artificial Intelligence Laboratory AI-TR-492.

Waters R.C. (1979) A Method for Analysing Loop Programs IEEE Trans. on Software Eng. SE-5:3, pp. 237-247.

Waters R.C. (1982) The Programmer's Apprentice: Knowledge Based Program Editing IEEE Trans. on Software Eng. SE-8:1, pp. 1-12.

Weiser M. Programmers Use Slices When Debugging (1982) CACM 25, 7.

Wertz, H. (1987) Automatic Correction and Improvement of Programs. Ellis Horwood Series in Artificial Intelligence.

Wills, L.M. (1986) Automated Program Recognition. MSc Thesis MIT Electrical Engineering and Computer Science.

Wills, L.M. (1990) Automated Program Recognition: A Feasibility Demonstration. Artificial Intelligence 45, pp. 113-171.

Winograd T. (1973) Breaking the Complexity Barrier (Again). Proc. ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973.

Youngs, E.A. (1974) Human Errors in Programming. Int. J. Man-Machine Studies 6, pp. 361-376.