AVID: A SYSTEM FOR THE INTERACTIVE DEVELOPMENT OF VERIFIABLY CORRECT PROGRAMS

Dean Blackmar Krafft
Ph.D. Thesis

TR 81-467
September 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

AVID: A SYSTEM FOR THE INTERACTIVE DEVELOPMENT OF VERIFIABLY CORRECT PROGRAMS

Dean Blackmar Krafft, Ph.D.
Cornell University 1981

The AVID system is designed to Aid Verification through the techniques of Interactive program Development. AVID continues the work in programming logics begun at Cornell University in 1975. It provides a syntax-directed editing environment for the development by stepwise refinement of programs and proofs in the PL/CV2 programming logic. AVID is another step in the continuing effort to provide methods and software tools for developing correct programs.

AVID contains a number of important contributions to the area of program/proof development. To allow the full power of the AVID verification facilities to be applied to programs developed by stepwise refinement, we created a new program construct, called an _ATTAIN block_, that formalizes the concept of a refinement level. This construct allows the independent verification of different refinement levels, and thus of partially developed programs. Using this construct, AVID can guarantee that the refinement in a top-down development actually implements its high-level specification.

AVID is the first system to support the interactive display of logical dependency in proofs. We have developed a new algorithm, built on the _congruence closure_ method for deciding the theory of equality, that efficiently determines logical dependency within this theory. This algorithm is independent of the AVID system, and has potential applications wherever the congruence closure method is used.

AVID also contains some significant contributions to the area of

syntax-directed editor design. AVID makes use of a standard, powerful, and widely available screen-oriented editor as its user interface. The system design gives a strategy for the incorporation of other powerful editors into syntax-directed development systems. AVID is also the first system to make extensive use of _derived nonterminals_ to avoid redundant specification and to guide the user in developing his proof.

Perhaps AVID's most important contribution is its demonstration of the feasibility of a system to support and enforce the development by stepwise refinement of provably correct programs. For programs that _must_ be correct, this approach may be one of the most promising.

One final contribution of the AVID project is the system itself as a base for future research. The modular design of the AVID system and its facilities for the high-level description of AVID language constructs make the system easy to modify and to build on. There are already several projects planning to use AVID in this fashion.

AVID has been implemented on a DEC VAX 11/780 under the Berkeley UNIX operating system. The current version of the system, which supports the development and verification of the predicate calculus portion of PL/CV2, consists of approximately 20,000 lines of C language source code. When running on the VAX, the current system requires 230K bytes of memory. A version of the system suitable for distribution is expected to be available by January 1982.

ACKNOWLEDGEMENTS

It is impossible for me to fully express the gratitude I feel toward all the people who helped to produce this thesis. In particular, without the constant encouragement and boundless enthusiasm of my advisor, Robert Constable, I would have ground to a complete halt long ago. His excellent advice and unfailing support served to guide this project around the inevitable rocks and reefs to its final conclusion. He has also been a good friend and enjoyable companion in a large variety of circumstances. Both professionally and personally I am deeply indebted to him.

I would also like to thank Alan Demers for his advice and assistance. Without his brilliant insights, the algorithm presented in Chapter 5 would almost certainly not exist. He has also extensively reviewed the thesis, in several versions, and has made numerous helpful suggestions.

David Gries has made a great many stylistic and substantive contributions to this work. His unremitting attacks on the words "simply" and "actually" and his elucidation of the difference between "which" and "that" have helped to make this thesis much pleasanter reading.

In developing the AVID system, I have benefited greatly from many discussions and suggestions. For these, I would like to thank Scott Johnson, Dan Zlatin, Dick Conway, Jim Archer, Tim Teitelbaum, Tom Reps, Mike O'Donnell and many, many others.

I would particularly like to thank Ryan Stansifer for his efforts in translating the PL/CV2 verifier from PL/I to C and in implementing the template description language processor. I would also like to thank all those who worked on the PL/CV project, without which AVID would have been impossible.

At a more personal level, my years in Ithaca would have been much duller and grayer without the friendship of Jim and Loren Archer. They have both helped to keep me sane throughout the last five years.

To my parents, and indeed to my entire family, I owe much of what I am. This work is their work as well.

Finally, I would like to thank my wife, Kathy, for her support, patience, and love. "The sum that two married people owe to one another defies calculation", Goethe says, "It is an infinite debt, which can only be discharged through all eternity".

TABLE OF CONTENTS

CHAPTER 1
Introduction

1.1 What is AVID?

The AVID system is designed to Aid Verification through the techniques of Interactive program Development. It is another step in the continuing effort to provide methods and software tools for developing correct programs. AVID combines the techniques of interactive program development by stepwise refinement with those of logical proof and verification. It provides a new, unique environment for the development of provably correct programs.

The roots of the AVID system lie in two separate reaearch projects at Cornell University. The logical and mathematical foundation of AVID is the PL/CV programming logic [Constable and O'Donnell 78]. In this logical system, formal proofs of programs can be written and mechanically checked. Over the period 1976-1980, a computer system was developed to perform this checking. This system, known as PL/CV2, is described in [Johnson 81] and [Constable et al 81].

The user interface and proof development style of AVID are most directly inspired by the Cornell Program Synthesizer [Teitelbaum and Reps 80]. This system, in turn, builds on the the techniques of program development by stepwise refinement [Conway and Gries 79] and the notion of a template driven structure editor [Hansen 71].

In AVID, the results of these two projects have been synthesized and extended. Like the Synthesizer, it provides a friendly user environment supporting the top-down development of programs. Like the PL/CV2 verifier, it checks the total correctness of the program against a logical specification. In combining the functions of both systems, however, AVID can provide facilities that neither system could provide separately. Unlike PL/CV2, AVID allows verification of partially developed programs and the display of logical dependency in the proof. Unlike the Synthesizer, AVID can check that a refinement actually implements its high-level specification. This synergistic interaction has resulted in a new and interesting program development tool.

The AVID system differs significantly from previous "interactive" program verification systems. In these systems, typified by the Stanford Pascal Verifier [Luckham et al 79], the input is a program annotated with specifications and loop invariants. From this, the system generates a collection of formulas known as verification conditions. The original program is valid if and only if the verification conditions are valid. In the next step, the verification conditions are processed by an automatic theorem prover that transforms them into equivalent but possibly simpler formulas. Any verification conditions that cannot be simplified to **true** are then presented to the user. The user "interaction" now takes place. The user must look at these simplified formulae, divorced from the original program text, and decide whether they are incorrect, whether further attempts should be made to prove them correct, and whether they are "obviously" correct and can therefore be ignored.

The interaction in AVID is totally different. The user is interactively constructing a program and, simultaneously, its proof of correctness. He is always dealing directly with the proof itself, in a complete and well-defined logic, and never with a collection of random lemmas whose logical derivation is shrouded in the mystery of the simplifier.

1.2 AVID's contributions

AVID has made a number of important contributions to the area of program/proof development. AVID is the first interactive program development system based on a solid and well defined programming logic. Many systems allow interactive verification (e.g. [Boyer and Moore 79], the Stanford Pascal Verifier [Luckham et al 79], and AFFIRM [Thompson et al 79]), but these systems do not allow interactive development. At least one system, the SID/Gypsy system [Moriconi 77], supports the interactive development of large programs, although its approach is much different from that of AVID. However, SID/Gypsy is not based on a well-defined programming logic. AVID is also the first system to allow the interactive display and development of natural deduction proofs. Although there are systems to allow interactive development of proofs (e.g. GOAL [Buines-Rozas 79] and LCF [Gordon et al 79]), displaying the state of proof development in these systems is either difficult or nearly impossible. AVID introduces a new programming language construct that allows the independent verification of refinement levels. It is the first system to support the interactive display of logical dependency in proofs. To this end, we have developed a new algorithm, built on the *congruence closure* method for deciding the theory of equality, that efficiently determines logical dependency within this theory.

AVID has also made some significant contributions to the area of syntax-directed editor design. Unlike systems such as MENTOR [Donzeau et al 80], the Cornell Program Synthesizer [Teitelbaum and Reps 80], and COPE [Archer 81], AVID makes use of a standard, powerful, and widely available screen-oriented editor as its user interface. It gives a strategy for the incorporation of other powerful editors into syntax-directed development systems. It is the first system to make extensive use of *derived nonterminals* (described in section 2.3) to avoid redundant specification and to guide the user in developing his proof.

Perhaps AVID's most important contribution is its demonstration of the feasibility of a system to support and enforce the development by stepwise refinement of provably correct programs. For programs that *must* be correct, this approach may be one of the most promising.

To allow the full power of the AVID verification facilities to be applied to programs developed by stepwise refinement, we have created a new program construct, called an *ATTAIN block*, that formalizes the concept of a refinement level. We present this construct and discuss the problems of stepwise refinement in chapter 3.

As compared to the batch PL/CV2 verifier, AVID has a number of advantages. Several annoying problems relating to the input syntax of PL/CV2 have been eliminated. The use of *derived nonterminals* allows the user to avoid redundant specification required by the batch system. AVID has all the advantages over batch PL/CV2 that the Cornell Program Synthesizer has over batch PL/CS. These include immediate response to errors, the power of a screen oriented editor, and the ability to develop a program through direct refinement techniques. Unlike batch PL/CV2, AVID allows the verification of partially developed programs and the interactive display of logical dependency.

One final contribution made by AVID is the system itself as a base for future research. The modular design of the AVID system and its facilities for the high-level description of AVID language constructs make the system easy to modify and to build on. There are already several projects planning to use AVID in this fashion.

Summing up, we feel that AVID has made a number of significant contributions to the areas of interactive development systems and verification. The system serves as an important stepping stone to the eventual goal of a truly practical system for the interactive development of provably correct programs.

1.3 An Overview of the Thesis

This thesis discusses the most interesting features of the design and implementation of the AVID interactive proof development system. It makes no attempt to describe the PL/CV2 programming language accepted by the current implementation of AVID, nor does it describe all the commands and features of AVID. The reader is referred to [Constable et al 81] and [Krafft 81] for more information on these matters.

In chapter 2, we present the basic design principles of the AVID system, together with a description of the various parts of AVID. This should give the reader an understanding of how the system works. In chapter 3, we discuss the problem of developing correct programs and verifying their correctness in AVID. In chapter 4, we discuss the problems of incremental change, which are just now becoming understood. Chapter 5 presents a new efficient algorithm, which allows the AVID system to determine logical dependency while using the congruence closure decision procedure for equality. In chapter 6, we give a high level overview of the implementation of AVID. We present some of the internal structures of the system, and some of the software tools used in its implementation, the intention being to give a system designer who is considering adapting AVID to some other problem some notion of the underlying system structure. Chapter 7 describes the current status of the AVID system, discusses future directions for the AVID project, and summarizes the results of this work. Finally, the Appendix presents a small example from an AVID development session.

CHAPTER 2
Interactive Proof Synthesis

2.1 Introduction

The purpose of any programming environment is to aid in the rapid development of correct programs. In this chapter, we describe the major features of the AVID proof development system. In section 2.2, we present the basic design principles underlying the interactive proof development user interface. In the following sections we discuss the major proof-synthesis features of AVID. Finally, we review our basic design principles, contrasting them with the actual implementation.

Both the characterization of the problems of interactive programming environments and the specific solutions used in AVID are based on earlier work in the field. Below, we briefly describe some of the relevant features of these earlier systems. For greater detail, the reader should consult the referenced papers.

In designing the EMILY text editing system, [Hansen 71] was one of the first to attempt to analyze the basic user engineering principles of an interactive program-development environment. EMILY provides a screen-oriented editor based on a template-selection system in some ways similar to that used in AVID. The system makes use of a BNF-like abstract syntax to specify the language being edited. As we discuss in chapter 6, AVID makes use of such a language description, but the concept has been considerably extended to support sophisticated error checking and the generation of verification code.

More recently, the MENTOR system [Donzeau et al 80] studied the problems of programming environments supporting structured editing. MEN-

TOR provides a general purpose tree manipulation language, in which the user can write programs to modify and process a tree-structured version of his program. AVID, on the other hand, provides a set of high-level commands, which are not user extensible. We felt that a general purpose manipulation language was complex and unnecessary in this application, but this might be an interesting area for future investigation.

The MENTOR system does not provide a screen editor for program structures. The authors cite two reasons for this: portability, and the difficulty of maintaining two separate representations, one for the internal abstract-syntax tree, the other for the screen display.

We postpone discussion of the problem of portability until chapter 6. On the second problem, the authors of MENTOR state that, "The difficulties [of maintaining two representations] may not be insurmountable, but it is not clear that the end result would be worth the effort." Since the AVID system keeps two such representations, we believe that the difficulties are certainly not insurmountable. In this author's opinion, for reasons that this chapter and the next should make very clear, the effort involved in providing screen display was definitely worthwhile.

The most immediate ancestor of the AVID approach to programming environments is clearly the Cornell Program Synthesizer (CPS) [Teitelbaum and Reps 80]. Like CPS, the AVID system makes use of explicit commands to select a production from the language grammar and display it on the screen. Unlike CPS, AVID makes use of a very sophisticated text editor to make modifications to the program structure and text. In AVID we have also introduced several new and improved concepts in syntax-directed editing.

The COPE system [Archer 81], recently developed at Cornell, offers a slightly different approach to the problems of the interactive programming environment. Rather than use commands to choose program structures, the user enters very abbreviated structure-specifications, and the system makes its best guess at filling them in. Because of the complexity of AVID templates, we do not feel that this approach is suited to AVID. As the COPE system becomes available, however, it will be interesting to compare its ease of use with AVID and CPS.

In this chapter, and throughout this thesis, we focus only on the part of the programming process that concerns the actual development and verification of programs. We do not consider the problems of program execution, testing, and debugging in the interactive programming environment. Program execution is simply outside the scope of the current AVID system, although such a facility could probably be integrated with it fairly easily.

2.2 Design Principles for Interactive Programming Environments

In this section we present a set of design principles for interactive program development systems. This list of principles is not intended to be complete, but they have all been basic to the AVID design. For each principle, we give a short justification of its importance. Throughout the rest of this chapter, we will refer back to these principles as we discuss the specifics of the AVID user interface.

Our design principles are similar to those of a number of existing systems. In particular, we are indebted to the previous design work of [Teitelbaum et al 81] and [Hansen 71]. In the last section of this chapter, we will examine how well AVID has met these design principles.

Principle 2.1. Frequent user actions should be simple to perform.

According to this principle, the system should be specialized, in that it should allow single short commands for all common actions. When these commands perform some reasonably complex action, then this principle becomes the synthesis of two of Hansen's three basic user engineering principles: "Minimize memorization" and "Optimize operations". This principle is clearly basic to any good user interface.

Principle 2.2. The set of user commands should be _complete_, in that it should be possible to perform any desired action in some reasonable way.

While it would probably always be possible for the user to delete his existing program and reenter the desired program, this does not seem like a reasonable way to make a change. The idea behind this principle is that conceptually simple changes should be simple to make.

There is some conflict between this principle and several other principles listed here. For example, with regard to the principle above, if single commands can perform complex actions, then the system designer must be careful that he has not rendered some meaningful intermediate state difficult or impossible to reach. At the end of this chapter we will make a few further comments on this conflict.

Principle 2.3. The user interface should be _consistent_.

To quote [Teitelbaum et al 81]: "A consistent approach is one in which the entire user interface is based on a few fundamental rules and tenets. Mastering these few ideas enables the user to generate most operations required to use the system. [Hansen 71] refers to this principle as 'predictable behavior'." Consistency is important both because it helps to minimize memorization, and because it helps to eliminate a source of error.

Principle 2.4. All user actions should be _reversible_.

If the user makes an error, this design principle guarantees that it will not be catastrophic. Even (especially) when the user action has caused significant and extensive change, the user should be able to undo the change in a single step.

Principle 2.5. The environment should provide _immediate visual response_ to user actions.

The programmer is most aware of the problems and requirements of a specific statement at the time he is actually developing it. Even a short time later, he must re-acquaint himself with the reasoning involved before he can correct errors. Thus, immediate error-feedback will save work.

In the case of more fundamental errors, such as those involving unverifiable proof assertions, there may be a basic failure in problem analysis. The earlier the system is able to warn the programmer of this problem, the more quickly he will be able to correct his misunderstanding. Thus wasted effort will be avoided.

Immediate notification of errors aids the programmer in understanding the specific source of the error. It will be clear to him that his most recent action was the cause of the error.

Principle 2.6. An interactive programming environment should support the notion of _abstraction_ during program or proof development.

During the top-down development of a program, the programmer must be able to focus his attention on the specific problem level being considered. In doing this, the programmer should ignore irrelevant portions of the program.

An interactive system should therefore provide mechanisms to allow the suppression of currently irrelevant detail. Only material needed at the current level of abstraction should be presented.

Principle 2.7. An interactive programming environment should, where possible, guide the programmer in the correct development of his program (and proof).

This design goal is particularly important when the system is used for proof development. By displaying the structure of the proof rule, the system guarantees that the programmer is aware of all the details involved in the correctness proof of the particular construct. If this goal is fully met, the system will display the exact logical assertion to be proved and all the assertions that the construct makes available as hypotheses to that proof. This greatly aids the programmer in quickly developing the exact proof required.

2.3 The Structure Editor

In AVID, the programmer develops program-proofs using a screen-oriented structure editor. The structures are based on the formal grammar of the language supported by AVID, i.e. PL/CV2. We will make use of the concept of a formal grammar in describing how the user develops an AVID program.

A context-free grammar consists of a set of terminal symbols, a set of nonterminal symbols, a set of productions, and a start symbol. The programmer uses the AVID structure editor to construct a derivation tree from the start symbol of the grammar. When he is done, this derivation tree will represent a complete, syntactically correct program.

The start symbol of a grammar is a distinguished nonterminal symbol from which the derivation must begin. In AVID, the nonterminal **<object>**

is used as the start symbol. When the system begins execution, the **<object>** nonterminal is displayed alone on the screen.

To begin building the derivation tree, i.e. constructing the program, the programmer selects a production from the set of productions in the grammar such that the left hand side of the production consists of the start symbol nonterminal. The programmer does this by issuing a command that selects such a production. As a result, the abstract syntax tree (ast) for the selected construct is inserted into the program derivation tree.

The ast is a tree whose root is the name of the production and whose leaves are the nonterminals of the right hand side of the production. Figure 2.1 shows a simple abstract syntax tree for a single program construct.

The AVID system is capable of displaying these trees as normal, well-structured program text. The display form of a single construct is called a template. The template displays all the information contained in the production associated with the construct, including formatting information and syntactic terminal symbols. Thus, while the ast for a construct contains all the information needed to interpret the construct, the template is designed to provide a friendly display format for the construct. The template for the construct shown in Figure 2.1 is given in Figure 2.2 exactly as it would be displayed on the screen.

In AVID, the notion of a nonterminal has been slightly extended from the standard BNF definition. There are four types of nonterminals. All but the last correspond to syntactic constructs generally available in extended BNF grammar descriptions. The four types are described below.

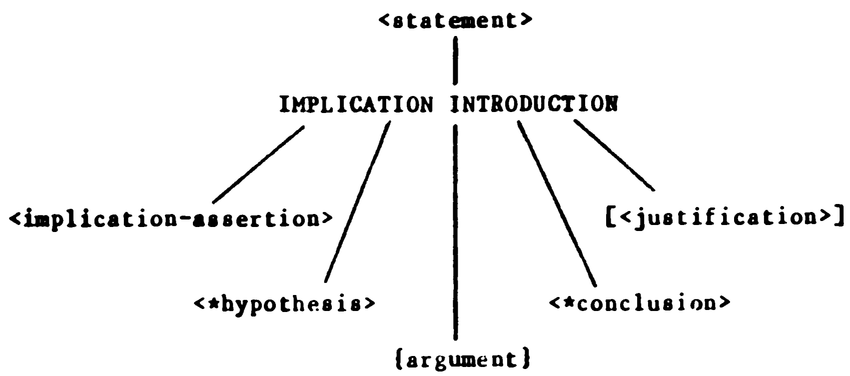


Figure 2.1 Abstract Syntax Tree for an Implication Introduction

```
<implication-assertion> BY IMPIN,
    PROOF;
        ASSUME <*hypothesis>;
        {argument}
        <*conclusion> [<justification>];
    QED;
```

Figure 2.2 Implication Introduction Template

(1) *Nonterminals* are the ordinary nonterminals of the grammar. They can be *refined*, i.e. productions can be applied to them, by single *asts* or single terminal phrases. They are displayed in angle brackets.

(2) *Optional nonterminals* are nonterminals that can be refined only to the null element of the grammar or to a single fixed template. They are only displayed by the system when the single template is "turned on". They are displayed in this thesis in square brackets.

(3) *Iterating nonterminals* are nonterminals that can be repeated zero or more times to form a list of instances of the nonterminal. The programmer can always create a new instance of such a non-terminal

adjacent to any refined existing instance. These nonterminals are displayed in curly brackets.

(4) Derived nonterminals are nonterminals that are automatically derived by the AVID system from information available to it. The programmer can treat these as syntactic terminals, which will change depending on the information available. They cannot be directly refined by the programmer. They are displayed in angle brackets, with the first character of the nonterminal description being an asterisk.

The AVID system also distinguishes two types of terminal symbols:

(1) Syntactic terminals are symbols that have no particular information value to the system. They appear only in the displayed templates to aid the programmer in interpreting the display. These symbols are generally upper-case text or punctuation symbols.

(2) Phrases are text strings the programmer types in as the last step in deriving the program structure. These strings in fact have significant structure of their own and are parsed by an LR(1) parser within AVID. Examples of phrases include expressions, assignment statements, lists of variables in declaration statements, and logical assertions.

Once the programmer as made an initial choice of construct, he has displayed before him a template containing, in general, various nonterminal and terminal symbols. The programmer can now position the cursor at any unrefined nonterminal, i.e. any nonterminal that is currently a leaf in the derivation tree, and issue a command to select a production from the language grammar. This causes the template for the language construct to replace the nonterminal on the screen. The programmer can then position the cursor.at any other nonterminal position, possibly in the template he just inserted, and repeat the process. If a nonterminal can be legally refined to some phrase, the programmer merely types it in. It is immediately parsed and checked for errors. Any error message is displayed as soon as the programmer indicates that he has completed the phrase.

The AVID structure editor allows the user to create directly the structure of his program. Using single commands, the user selects complete language constructs corresponding to conceptual program-action units. Rather than merely typing in text for later parsing, he is guiding the system in a derivation of his program. When he enters text, it is immediately parsed and checked. In section 2.5 we will look at some of the specific facilities available in AVID to manipulate program structure and text. In the following section we will show how the templates can be used to guide program and proof development.

2.4 Guiding Proof Development

In programming environments designed to support the development of ordinary programs, the programming language constructs are usually fairly simple. The advantages of having the system provide the exact language construct on command, as is done in CPS, are only moderate. In the COPE program development system, for example, the user is normally expected to provide the complete structure for any program construct he uses, although he may abbreviate the syntactic terminals. In the AVID system, however, the language constructs are those of the programming language together with their proofs of correctness. These constructs can be quite complex. For example, the minimal specification of a proof of correctness and termination in a PL/CV2 DO WHILE statement takes fourteen lines. Simply remembering the general proof requirements, to say nothing of the specific syntax, is relatively difficult. Thus, the templates provided by the AVID structure editor offer a unique and important guide for the developer of program proofs. They serve to minimize user memorization of language constructs.

The AVID template for a PL/CV2 DO WHILE with its corresponding ATTAIN construct is shown in Figure 2.3. It provides the user with an immediate outline of the correctness proof of the statement.

The reader should observe the extensive use of derived nonterminals in the template. When the user has filled in the simple nonterminals with which the derived nonterminals are associated, the derived nonterminals are automatically refined. For example, when the user provides the termination expression for the loop, the AVID system will automatically copy it as indicated in the figure, modifying it where necessary.

This notion of derived nonterminals, first extensively developed in AVID, has a twofold benefit for the programmer. In the PL/CV2 batch verification system, all assertions shown as derived nonterminals in Figure 2.3 had to be entered by the programmer. The AVID system has eliminated the need for this redundant specification. The programmer need only enter the required minimum of actual program text, thus significantly reducing programmer effort.

A much more important benefit is gained from the use of derived nonterminals. In a proof, the programmer must have the specific goal of the proof immediately available. Without it, he will constantly be unsure of the exact formula he is attempting to obtain. The derived

```
ATTAIN <assertion>
    SOME <*id> FIXED . (<*id> >= 0 & <*termination-exp>);
    <invariant-assertion>
    ¬(<*bexp>) => <*assertion>;
    DO WHILE (<bexp>);
        ASSUME <*invariant-assertion> & ¬(<*bexp>) => <*assertion>;
        ARBITRARY <id> FIXED WHERE <termination-exp>;
        ¬(<*termination-exp-on-zero>);
        {statement}
        <*termination-exp-on-id-minus-1>;
        <*invariant-assertion>;
        ¬(<*bexp>) => <*assertion>;
        END;
    <*assertion>;
    QED;
```

Figure 2.3 The PL/CV2 DO WHILE Template

non-terminals provide these goals. As an example, suppose in the DO WHILE loop of Figure 2.3 the programmer has refined the boolean expression <bexp> to

```
x > 0
```

and the ATTAIN assertion to

```
all y fixed where 1<=y<=maxindex . max>=a(y) .
```

The template will then appear as shown in Figure 2.4. The programmer is now aware that, if it is not immediately obvious, he must provide a proof at the end of the loop that

```
¬(x>0) => (all y fixed where 1<=y<=maxindex . max>=a(y)) .
```

If he does so using the optional justification nonterminal, issuing a command to select the implication introduction template perhaps (shown in Figure 2.2), the system will automatically break the assertion into

```
ATTAIN all y fixed where 1<=y<=maxindex . max>=a(y)
    SOME <*id> FIXED . (<*id> >= 0 & <*termination-exp>);
    <invariant-assertion>
    ¬(x>0) => (all y fixed where 1<=y<=maxindex . max>=a(y))
    DO WHILE (x>0);
        ASSUME <*invariant-assertion> &
            ¬(x>0) => (all y fixed where 1<=y<=maxindex . max>=a(y))
        ARBITRARY <id> FIXED WHERE <termination-exp>;
        ¬(<*termination-exp-on-zero>);
        {statement}
        <*termination-exp-on-id-minus-1>;
        <*invariant-assertion>;
        ¬(x>0) => (all y fixed where 1<=y<=maxindex . max>=a(y))
        END;
    QED;
```

Figure 2.4 A Partially Refined Template

hypothesis and conclusion, and the lower level proof goal will again be automatically displayed as in the program fragment shown in Figure 2.5

The emphasis on guiding proof development is unique to the AVID system. As made clear in section 2.2, we feel this is a important feature of an interactive programming environment. Throughout the design of the AVID system, we have attempted to provide immediate and thorough guidance for the program and proof developer.

2.5 Manipulating Text and Structure

In this section we will give a brief overview of the program and proof manipulation features of AVID. For a detailed and exhaustive description of these features, the reader is referred to the *AVID User's Manual* [Krafft 81].

The user interface to the AVID system is based on a powerful and versatile screen-oriented text editor. The editor interface chosen was

```
<*termination-exp-on-id-minus-1>;
<*invariant-assertion>;
¬(x>0) => (all y fixed where 1<=y<=maxindex . max>=a(y)) BY IMPIN,
    PROOF;
        ASSUME ¬(x>0);
        {argument}
        all y fixed where 1<=y<=maxindex . max>=a(y);
    QED;
END;
```

Figure 2.5 Program Fragment Demonstrating Automatic Goal Display

that of vi, a visual text editor written at Berkeley and described in [Joy 80]. The use of vi-style commands throughout the system provides a consistent user interface, which is already familiar to a large number of users of the Berkeley UNIX systems.

When we first considered adopting the vi user interface, we were concerned about how well an editing system designed to work with arbitrary text files could deal with a highly structured object. Although there have been claims that normal screen editors, augmented with some structure commands, can deal with structured program editing (see, for example, [Wood 81], which makes such claims for the Z editor), there seemed to be several problems with this approach. Since these editors do not have available a complete syntax for the language, a number of our design principles, such as immediate visual response to errors and being able to guide proof development, seemed to be impossible to achieve.

We decided instead to reinterpret the vi commands to deal with structures. vi was designed to work with characters and lines. In AVID, the basic units are phrases and pieces of abstract syntax trees. We chose to let the vi line-oriented commands affect asts, and to use the character-oriented commands within phrases. Since phrases are actually just character strings, only the line-oriented commands were changed.

To give an example, in *vi* the command **dd** serves to delete a line of text. In AVID, the **dd** command deletes the current subtree. Within a line of text, the *vi* command **d$** deletes the rest of the line. In AVID, the same command deletes characters to the end of the current phrase.

We found that almost all *vi* commands could be reinterpreted in this fashion; thus, the change in conceptual outlook required to go from the *vi* text editor to AVID is very small. Almost all the commands and facilities to which the programmer is accustomed in his powerful text editor are available in AVID.

It also seems clear that almost any editor that currently deals with both lines and characters can be reinterpreted in this fashion. We made use of the *vi* editor because it included the powerful features we wanted and because it was readily available. Other screen-oriented editors may be better suited to the task of structured program editing.

The version of the *vi* editor in the AVID system supports a broad range of powerful editing commands. These include the ability to give single one-word commands to select templates for complete program-language constructs, the ability to do standard screen editing on the program phrases, the ability to clip portions of the program tree into named buffers and to copy them elsewhere, and the ability to store and retrieve arbitrary size program segments from disk.

The operations of inserting, deleting and copying program text are implemented in AVID by the same mechanism as that used to write files to disk. In AVID, structures are rendered completely context free at the time they are removed from the derivation tree. To accomplish this, they are converted to a sequence of commands, cursor movements and text phrases. The format of this sequence is presented in section 6.5. When the sequence is later inserted, the commands and phrases are interpreted exactly as though the programmer had entered them himself at that point.

The AVID system also supports a new and interesting text manipulation facility. Since the logical assertions of PL/CV2 can be long and complex, the language provides two mechanisms to reduce the amount of assertion entry required. The language has a parameterized macro facility, the DEFINE construct, and it allows labelling of arbitrary assertions in the proof text. Macro references and labels can then be used wherever assertions are permitted (macros may also be used in place of expressions).

Frequently, the programmer will wish to obtain an assertion that is only a minor variation on some previously defined assertion. In batch PL/CV2, the programmer could only specify a label or macro at some point in his proof, obtaining a reference to the exact assertion. In AVID, the programmer can use the **expand** command to instantiate the underlying assertion for any label or macro reference at some point in the proof.

For example, if the following macros had been declared in the system:

```
FOR x FIXED DEFINE hole(x) = 1 <= x & x <= 3;
FOR (x, y, z) FIXED DEFINE distinct(x, y, z) =
        hole(x) & hole(y) & hole(z) & x ~= y & y ~= z & x ~= z;
```

then executing an **expand** command on the following expression:

```
hole(w) & distinct(a, b, c);
```

would result in:

```
1 <= w & w <= 3 & 1 <= a & a <= 3 & 1 <= b & b <= 3 &
        1 <= c & c <= 3 & a ~= b & b ~= c & a ~= c
```

The programmer can now easily modify the resulting assertion using the editing facilities of the system.

At this point, some mention should be made of the cursor movement features of AVID. As in CPS, the basic cursor movement is tree structured. The cursor moves among nodes (nonterminals) in the derivation tree. At any node, the programmer can issue commands or, where appropriate, enter phrases. In AVID, however, vi context searching and line-oriented movement commands are also supported. This is possible because AVID uses a different representation for the print form of the program than that used by CPS. The user can scroll up and down in the program, search for specific string occurrences, and even move directly to the seventeenth line from the beginning of the displayed program text. As a result, the programmer is not confined to a single display, but can move quickly to any point in the program.

2.6 Proof Display

The reader has already seen examples of the basic display form of AVID proofs, for example in Figure 2.4. It should be clear that these proofs can be large. Without some means of controlling proof display, the programmer will be constantly distracted by irrelevant proof detail.

There are two mechanisms by which proof (and program) detail can be suppressed. One is automatic; it is basically the system's best guess as to what detail is currently relevant. The other is manual; it is used to override the automatic system when it has not made the proper choices. Currently, the AVID system supports only the manual suppression of proof text. The major question involved in automatic suppression is whether the benefit of always displaying the correct level of abstraction outweighs the cost of a constantly changing display. At some time in the future, we hope to have both the manual and the automatic facility available to allow comparison. In either case, the display indicates that part of the program has been suppressed by replacing it by three dots (the elision characters), as in

```
PROOF;
...
QED;
```

The proposed algorithm for the automatic display mechanism attempts to guarantee that the system always displays the proof at a level of detail that is consistent with the problem the prover is attempting to solve. Thus, essential information must be immediately available and highly visible, and irrelevant detail must be suppressed, so that the prover is not distracted from the task at hand.

This automatic algorithm is quite simple. In terms of the derivation tree, it assumes that, for proofs, only siblings at the current level and elements on the path to the root need be displayed. Thus, the detail of proofs of assertions that are siblings to the current assertion are suppressed. For example, in the Implication Introduction proof of Figure 2.5, once the cursor is moved away from the nonterminal

{argument}

that entire subproof will be elided, as shown in Figure 2.6. Note the presence of the asterisks (*) on the line containing the elision characters. This indicates that the elided text contains unrefined nonterminals. If the text contained an error, the second asterisk would be replaced by a number sign (#). The system uses other special symbols to

```
     <*termination-exp-on-id-minus-1>;
     <*invariant-assertion>;
     ¬(x>0) => (all y fixed where 1<=y<=maxindex . max>=a(y)) BY IMPIN.
**       ...
     END;
```

Figure 2.6 Example of Automatic Subproof Suppression

provide verification information. These will be described in sections 3.3 and 3.4.

If the programmer does not agree with the system's choice of text to suppress, he can individually "turn on" or "turn off" sections of text. He can also completely disable the automatic mechanism and make his own choices about the text to suppress.

For program statements, the system will automatically suppress statements only when they are within the ATTAIN construct. There is still some discussion about whether the system should allow the user to manually suppress other program statements. The COPE system [Archer 81] allows arbitrary suppression, while the CPS system does not.

[Mikelsons 81] discusses a general automatic suppression mechanism for text display. It is somewhat more general than the algorithm we have described, and in a modified form it might be suitable for inclusion in the AVID system.

The problems of suppressing program text will be discussed further in section 3.2 where we introduce the ATTAIN construct.

2.7 A Review of the AVID Design Principles

We feel that we have followed most of the design principles quite closely, although there is always room for improvement. In the paragraphs below, we consider individually the various design principles.

The first principle, (2.1), held that frequent user actions should be simple to perform. The most frequent actions: selecting templates, moving the cursor, and entering text are all quite simple to invoke. To draw a more exact conclusion on this issue will require accumulating more user experience than we have at present.

The second design principle, (2.2), required the set of commands be complete. In this we have not been as successful as we might like. The subtree clipping and insertion features allow fairly arbitrary changes, but changing a "DO WHILE" construct into a "DO UNTIL" construct is fairly complex, perhaps more so than the conceptual change warrants. The user must clip out the phrases, delete the old template, insert the new template, and reinsert the phrases. In using simple commands to provide templates to guide the user in developing the proof, we have made it sometimes difficult to change the program structure. We could introduce program transformation commands, as has been suggested for the CPS system to solve the same problem, but the situation in AVID is much more complex than in CPS. To take just one example, the proofs of termination for the "DO WHILE" and "DO UNTIL" constructs are different. Should the transformation attempt to transform the phrases involved in the termination proof, or should it assume that the phrases are correct for the new template? For this and many similar problems, there does not seem to be a good solution.

The third design principle, (2.3), demands that AVID provide a consistent user interface. By making use of the <u>vi</u> front end for all AVID commands, we have essentially pushed this question off onto the design of <u>vi</u>. We are not prepared here to argue the strengths and weaknesses of <u>vi</u> itself, but we do feel it provides a fairly uniform, consistent interface.

The problem of reversibility (design principle 2.4) has been only partially solved. A single-step **undo** command is available, but it has not been implemented for all commands. An **undo** command that allows backing up arbitrarily far, together with a **redo** command to undo the effect of **undos** is clearly a more desirable solution. This approach has been implemented in the COPE system, although it has a fairly high cost in storage and efficiency. This is one area of AVID that is open to clear improvement.

In the matter of providing immediate visual response (design principle 2.5) we feel AVID has been very successful. Any change resulting in a syntactic error results in an immediate display. As we will discuss further in chapter 3, verification of partial proofs allows immediate feedback during the entire development process. We feel that the immediacy of the AVID system is one of its greatest strengths.

Levels of abstraction (design principle 2.6) have been implemented by means of a manual text suppression mechanism. In section 2.6 above, we presented a proposal for an automatic text suppression mechanism. At some point we hope to test this.

Our final design principle, (2.7), held that AVID should guide the programmer in developing his proof. The mechanisms in AVID to provide this guidance are discussed extensively in the next chapter.

CHAPTER 3

Developing Correct Programs

3.1 Structured Programming

During the last decade there has been a growing realization that there are a set of specific progrsm development techniques whose application results in the faster development of better programs. Tsken together, these techniques are usually referred to as the "Structured Programming Methodology". Some of these modern programming techniques, such as structured walkthroughs, are outside the scope of an interactive program development system, but some of the most important techniques are directly relevant to it.

One particular approach to program development has been fundamental to the design of the AVID system. In this approach the programmer develops his programs by the process of stepwise refinement. The underlying notion of this process is thst the programmer should solve small independent problems one at a time rather than lsrge problems all at once. At each level of development the programmer splits the current problem into a few simpler problems, such that the solutions to the simpler problems can be put together into a solution of the larger one. By making use of this technique, the programmer can isolate his concern to the specific problem he is currently analyzing. He does not yet need to concern himself with the solutions to the subproblems, and he is not in general interested in the detsils of the solutions to other problems at the same level. It should be clesr that the use of stepwise refinement in developing programs requires that the program be developed top-down, at least to some extent. at least to some extent

We will not attempt to present here the arguments in favor of structured programming. We refer the reader to [Dijkstra 76], [Dahl et al 72], and [Conway and Gries 79] for several clear and compelling presentations of these arguments.

The designers of a program development system can take one of two approaches to the relevance of structured programming to their work. They may view that it is not their business to enforce programming practices on the users of their system, and therefore provide an environment not particularly supportive of any one method of program development. This is the approach taken by COPE [Archer 81]. Alternatively, the system designers can attempt to craft their system so that it specifically supports modern programming practice. This is the approach taken by the AVID system. It has been designed to encourage strongly, and even to enforce, the use of stepwise refinement, isolation of concerns, and top-down development. In fact, as discussed in Requirement 2.2, we feel that any interactive development system must provide display facilities that support modern practices. Since both PL/CS [Bishop and Conway 77] and the Cornell Program Synthesizer [Teitelbaum and Reps 80] have taken the same approach, AVID is true to its heritage.

In choosing the more restrictive approach in designing AVID, we have been motivated partially by the knowledge that because AVID supports a programming logic rather than merely a programming language, it provides a unique opportunity for encouraging the correct development of programs. We have already discussed (in section 2.4) some of the advantages of proof and program display in guiding proof development. In this chapter we introduce several unique features of the AVID system and show how they interact to provide an environment that is specially suited to encouraging the development of verifiably correct programs.

3.2 ATTAIN blocks

In designing the PL/CV2-based AVID language, it became clear that an additional programming language construct was required. We needed some way to formalize the notion of a refinement step in the program. The notion of a refinement step as a programming language construct was first introduced by [Conway and Gries 79]. Their informal construct was the "statement-comment". It was originally merely a special use of the standard comment facility of the PL/C programming language. The construct is shown in Figure 3.1. The problem to be solved by the refinement is stated within a comment in a high-level language (English), and the code to implement the refinement appears indented below this comment.

In both PL/CS and the Cornell Program Synthesizer the "statement-comment" was introduced as an actual program construct rather than as a special use of the comment. The construct suffered from two related disadvantages. There was no way for the compiler to check that the refinement actually implemented the high-level specification, and there was no guarantee that some low-level refinement would not make an unexpected global change. Since the Cornell Program Synthesizer allows the programmer to suppress the display of the refinement portion of a statement-comment, this problem could result in very misleading displays.

The fact that the AVID system supports a programming logic rather than merely a programming language gave us an opportunity to avoid the problems encountered with statement-comments. By using predicate calculus assertions rather than English language comments, the system could check the accuracy of the refinement against the high-level specifica-

```
/* <high-level specification of refinement step> */
    {refinement code}
```

Figure 3.1 Statement-Comment

tion. The new construct, the ATTAIN block (shown in Figure 3.2), thus formalizes the notion of a statement-comment.

As discussed in section 2.6, the AVID system can suppress the display of the refinement of an ATTAIN block. When using the Cornell Program Synthesizer, such suppression can result in hiding program errors. With the PL/CV2 programming logic, however, the AVID system can flag the ATTAIN assertion as erroneous whenever it is incorrectly refined. Thus, the AVID system can guarantee that the displayed program text exactly reflects the behavior of the complete program.

The next few paragraphs discuss the specific clauses of the ATTAIN construct; examples of it will be given in the following section.

The optional ASSUME statement allows the correctness of the refinement to be determined independently from that of the program containing it. This will be discussed in detail in the following section.

The optional WITH <list-of-equality-statements> allows the programmer to give names to initial values of variables that will be changed within the ATTAIN block. These names cannot be used in the program itself, only in assertions that reason about the program.

The ATTAIN statement is the heart of the ATTAIN construct. It specifies the assertion that the refinement is to achieve. As we will see in the following section, it allows verification of partially developed programs. It also allows the AVID system to check that the refinement accurately implements the goal of the block.

```
[ASSUME <assertion>;]
[WITH <list-of-equality-statements>]
ATTAIN <assertion> [, CHANGING <variable-list>];
    <attain-refinement>
    <*assertion>;
    QED;
```

Figure 3.2 ATTAIN construct

The optional CHANGING <variable-list> portion of the construct allows the programmer to specify the program variables that will be modified by the refinement block. This is used to control assertion accessibility when the block is unrefined. The use of the CHANGING clause will be discussed in the next section. For a complete discussion of accessibility in PL/CV2, the reader is referred to [Johnson 81] pp. 101-106.

The <attain-refinement> nonterminal can either be refined to some statement for which a goal assertion must be given, e.g. a DO WHILE, or it can be refined to a block of program statements that form a logical refinement step.

As the reader may have already observed, the ATTAIN construct has some strong similarities to the PROCEDURE construct of PL/CV2. The CHANGING list of the ATTAIN block can be thought of as the read/write parameters of a procedure call. The ASSUME and ATTAIN assertions are exactly those that would appear for a procedure. As in a procedure, one of the purposes of an ATTAIN block is to isolate the code within it from the rest of the program. A procedure or an ATTAIN block can only modify the state of the program in a carefully controlled fashion. In some ways, then, they serve similar purposes.

It should also be clear, however, that procedures and ATTAIN blocks have some important differences. Procedures can be invoked recursively;

ATTAIN blocks cannot. The ATTAIN construct can also be invoked at only one point in the program. This allows the ATTAIN block's ASSUME statement to be optional, since this information can be derived from the list of accessible assertions at the point at which the block occurs (although this will restrict the verification information that can be obtained, as we will see in the next section). Thus, both constructs serve useful and independent purposes within the language.

3.3 Partial Verification

In designing the AVID system, we knew that by basing it on PL/CV2 we already had a good programming logic in which to describe verifiably correct programs. The emphasis in the AVID system was in developing correct programs. To do this, we felt it was essential to provide some mechanism for determining the correctness of partially developed programs.

In a system that requires the development of a complete procedure or theorem before meaningful verification information can be obtained, it is possible for the programmer to waste much effort. When verification reveals a conceptual error in program design, the programmer may be forced to make extensive changes. These changes could have been avoided if the system had performed verification one refinement level at a time.

AVID appears to have taken a unique approach to the development of verifiably correct programs. Of existing systems, the Stanford PASCAL Verifier [Luckham et al 79] requires that the programmer present as input a complete PASCAL program. The programmer must then interactively prove all assertions that the theorem prover cannot simplify to true. These assertions are presented completely divorced from the original program text. It is clear that the emphasis is not on program development.

The AFFIRM system [Thompson et al 79] takes a similar approach to program verification. The user provides a complete PASCAL-like program together with user-defined data types specified by algebraic axioms. Verification conditions are generated, and the programmer then interactively searches for a proof of these conditions. The emphasis of the system is on the development of proof search strategies, not the development of programs. The LCF system [Gordon et al 79] also emphasizes the process of searching for a proof, rather than that of developing a program.

The SID/Gypsy system of [Moriconi 77] allows the interactive development of systems of procedures, but its emphasis is on the interrelation of procedures. It is not particularly suited to the interactive development of the procedures themselves.

The PL/CV2 system [Constable and Johnson 79] does allow the programmer to directly relate errors in his proof to the position in the program text at which they occur, but it also requires that the programmer provide a complete theorem or procedure for batch verification.

The only system of which the author is aware that supports the actual process of developing verifiably correct programs is the PRL system [Bates 79]. As of this writing, however, the formal definition of this system is just being completed, and an implementation is still several years away.

The ATTAIN construct is essential to the verification of partially developed programs. It serves to isolate one particular refinement step from the remainder of the program. Consider the (rather artificial) program fragment shown in Figure 3.3. It consists of a single ATTAIN

```
(01) /* Find i such that a(i) = maximum(x,y) assuming there is one */
(02) /* Let max = a(i) */
(03) ASSUME SOME i FIXED WHERE 1<=i<=n .(a(i)=x & x>=y | a(i)=y & y>=x);
(04) ATTAIN 1<=i<=n & a(i)=max & ((max=x & max>=y) | (max=y & max>=x)),
(05)         CHANGING max, i;
(06)     /* Guarantee max = maximum(x,y) */
(07)     ATTAIN ((max=x & max>=y) | (max=y & max>=x)),
(08)             CHANGING max;
(09)         <attain-refinement>
(10)         ((max=x & max>=y) | (max=y & max>=x)),
(11)         QED;
(12)     /* Find i such that a(i) = max */
(13)     ATTAIN 1<=i<=n & a(i)=max,
(14)             CHANGING i;
(15)         <attain-refinement>
(16)         1<=i<=n & a(i)=max;
(17)         QED;
(18)     1<=i<=n & a(i)=max & ((max=x & max>=y) | (max=y & max>=x));
(19)     QED;
```

Figure 3.3 Program fragment containing ATTAIN blocks

refinement block containing two unrefined ATTAIN blocks (at 06-11 and 12-17). Since the ATTAIN beginning at line 01 has been fully and correctly decomposed, the verification process will not flag any errors at that line. Since both of the contained ATTAIN blocks are unrefined, the verification process will flag these (at lines 07 and 13) as being unverifiable. If the programmer had suppressed the display of irrelevant sub-blocks, the AVID display would be as shown in Figure 3.4. In this display the "..." is used to indicate that program text is being suppressed. The "??" flags at the left margin indicate unverifiable assertions. In this case both of these are the unrefined ATTAIN blocks. Note that the outer ATTAIN block (at line 03) is not flagged; since it has been correctly decomposed, it is marked as verifiable. The "*?"

```
    /* Find i such that a(i) = maximum(x,y) assuming there is one */
    /* Let max = a(i) */
    ASSUME SOME i FIXED WHERE 1<=i<=n .(a(i)=x & x>=y | a(i)=y & y>=x);
    ATTAIN 1<=i<=n & a(i)=max & ((max=x & max>=y) | (max=y & max>=x)),
            CHANGING max, i;
        /* Guarantee max = maximum(x,y) */
??      ATTAIN ((max=x & max>=y) | (max=y & max>=x)),
                CHANGING max;
*?          ...
        /* Find i such that a(i) = max */
??      ATTAIN 1<=i<=n & a(i)=max,
                CHANGING i;
*?          ...
        1<=i<=n & a(i)=max & ((max=x & max>=y) | (max=y & max>=x));
        QED;
```

Figure 3.4 AVID display showing suppression and error flags

flags are used to mark elided lines (those consisting of "...") to show that they contain unrefined nonterminals as well as unverifiable assertions. Note that the assertions shown at lines 10 and 16 are unverifiable. If the elided text had contained only unrefined nonterminals with no unverifiable assertions, the flag would have been "**".

The importance of the ATTAIN clause portion of the block should by this time be clear. The CHANGING clause, however, also affects directly the verification of partially developed programs. The variables listed in the CHANGING clause are assumed to be modified by the ATTAIN block. Thus, assertions that refer to those variables and that were accessible prior to the block will be inaccessible following the block. When the block is fully refined, the list of variables in the CHANGING clause is checked against the actual list of variables that may have been changed in the block (see [Johnson 81] pp. 101-106 for a discussion of what "may have changed" really means). If the lists differ, the CHANGING clause is flagged as being unverifiable. In the example, the CHANGING clause at line 05 is not flagged, since the two variables "max" and "i" are each noted as being changed in one of the two inner ATTAIN blocks.

We should discuss on the optional ASSUME portion of the ATTAIN block. An example is the ASSUME at line 03. The ASSUME allows for verification of bottom-up program development. If assertions required for the verification of the ATTAIN block are not accessible to the block, the ASSUME clause allows the system to isolate the block from the proof of those assertions. Thus, the ATTAIN can be shown correct independently of the proof of these required assertions. In the display, the ASSUME clause will be flagged as unverifiable until all the assertions required for its verification are accessible.

It should be clear that the ATTAIN block, by allowing verification of partially developed programs, can help catch errors at the earliest possible moment. This provides the kind of immediate visual feedback we specified in design principle 2.5.

3.4 Displaying Logical Dependency

One of the great strengths of an interactive development system is its ability to give immediate feedback to the system user. When dealing with proofs, it is frequently valuable to be able to determine the assertions on which some given assertion depends, and the assertions that depend on the selected assertion. The AVID system allows the programmer to display this information interactively.

The AVID system has a somewhat restricted notion of dependency. Not surprisingly, this concept of dependency corresponds exactly with that of the underlying PL/CV2 logic. The exact definition of dependency will be given in Chapter 5, where we present an extension to the equality decision procedure of [Johnson 81] to identify dependency. At this point, an example will probably best illustrate the logical dependency display facility.

In Figure 3.5 we give a simple program fragment. Suppose the cursor is positioned at line 04, and a display of logical dependency is requested. The assertion at line 04 is directly dependent on the assertions at lines 02 and 03. To get line 04 we apply the equality assertion implied by the assignment at line 03 to the arithmetic assertion proved at line 02. Note that while line 01 is used in the proof at line 02, the assertion we are considering is only indirectly dependent upon it. It should be obvious that the assertions at both lines 05 and 06 depend directly on the assertion at line 04.

Figure 3.6 gives the program fragment as it would be displayed by the AVID system upon request for a dependency display. The arrow indicates the line with respect to which dependency is being shown. The }} flags are used to indicate proclaimed assertions on which the selected assertion is directly dependent. The {{ flags are used to indicate assertions for which the selected assertion was queried in establishing their truth.

Figure 3.7 gives a sample dependency display where line 06 has been selected. Note that the logical dependency facility shows it as dependent on lines 01 and 03. The system could have just as well shown it dependent on lines 01 and 05, or on lines 01, 02, and 03. When there are alternate proof paths for some assertion, the system displays only the one it used internally to establish the selected assertion. The system user must realize that the displayed dependencies do not necessarily

```
(01)      x := 1;
(02)      x+1 > 0 BY ARITH, x = 1;
(03)      y := 0;
(04)      x+1 > y;
(05)      1+x > y;
(06)      x+x > y;
```

Figure 3.5 Program Fragment Illustrating Logical Dependency

```
          x := 1;
    }}    x+1 > 0 BY ARITH, x = 1;
    }}    y := 0;
    ->    x+1 > y;
    {{    1+x > y;
    {{    x+x > y;
```

Figure 3.6 Sample Display of Logical Dependency

```
    }}    x := 1;
          x+1 > 0 BY ARITH, x = 1;
    }}    y := 0;
          x+1 > y;
          1+x > y;
    ->    x+x > y;
```

Figure 3.7 Another Sample Dependency Display

indicate the only way in which the selected assertion could be proved. Nonetheless, we expect the facility to be useful in determining how proof changes will propagate through the proof.

CHAPTER 4

Incremental Change

4.1 Introduction

Systems such as AVID, CPS, and COPE have added a new dimension to the problem of change. Unlike systems supporting an edit/compile/debug cycle, these "synthesizer" systems allow change to occur while they have a completely parsed version of the program available. Potentially, this information can be used in creating a new parsed instance of the program that incorporates the change.

The interactive nature of these systems imposes new requirements on how they respond to change. One of the basic AVID design principles given in chapter 2 was:

> Principle 2.5. The environment should provide immediate visual response to user actions.

This principle implies that a structural or semantic changes made by the user should be immediately reflected in the display. For example, if the user deletes the declaration of variable x, occurrences of x in statements and expressions become undefined. In response, the system should immediately and visibly flag the statements in error and simultaneously print a message to inform the user of the problem.

Synthesizer systems can take one of two basic approaches in dealing with change. One approach is simply to treat the modified program as completely new input to the program parsing system. This approach has the advantage of simplicity. The process of parsing and compiling serial input text is well understood. Since some mechanism for parsing initial program input must exist in the language, no new concepts, and little new code, are required. The disadvantage of this approach is its potential inefficiency. It makes no use of the availability of a complete semantic analysis of the pre-change program. Thus, a small change in a large program may require a large amount of work.

The more sophisticated approach to handling change requires that the internal representation of the program be directly modified to reflect the change. We refer to this as incremental change. The advantage of doing incremental change is a potential gain in efficiency. The disadvantage is that new program mechanisms will probably be required to perform the change.

In the Cornell Program Synthesizer, a mixture of approaches is used. Very simple changes involving only the insertion of a new template or phrase are performed incrementally. Any change involving a declaration or a potential change in the scope of a phrase is done by brute force. The COPE system works about the same way.

4.2 The AVID Approach to Incremental Change

One of our earliest concerns in the design of AVID was incremental change. We tried to develop a general mechanism to deal uniformly with change throughout the AVID system. Our approach is as follows: A graph of dependencies among the program elements is maintained. When a change occurs in one element, the transitive closure of all the elements dependent upon it must be checked, and possibly reanalyzed. The new analysis must take place in the correct environment, which the system must be able to calculate. Observe that if the dependencies are cyclic, there is a problem.

For example, if a change is made to a declaration, the approach requires that the system have a graph of phrases dependent on the declaration. Each of these must be checked for error, depending on the exact type of change. The environment, in this case the symbol table, must be available in its updated form to reanalyze phrases affected by the change.

This approach to incremental change has been implemented in the AVID system.

Originally our intention was to extend these methods to allow incremental verification. One step in this direction was the development of an algorithm to determine logical dependency, which we describe in chapter 5. The most serious problem turned out to be creating the environment for reverifying dependent program elements in a way that was significantly more efficient than simply reevaluating from the beginning of the program. It would have been possible to create an audit trail, allowing the reevaluation to proceed either forward or backward from the current cursor position, but comparing the mechanisms and space requirements for doing this against the simplicity of complete reevaluation, we concluded that, for small examples, there was no gain.

Development on the AVID system began in the Fall of 1979. In the late summer of 1980, three researchers at Cornell [Demers et al 81] introduced the notion of modelling incremental change in syntax-directed editors with attribute grammars. When this work came to our attention, we realized that it had the potential to provide a clean, general solution to the problem of incremental change. Since, in the matter of incremental verification, our ad hoc approach had failed to yield significant improvements over the simple approach of just reverifying from the beginning of the program, we decided to use the simple approach in the first AVID implementation. Our intention is to incorporate the results of research in the attribute grammar model into AVID as they become available.

In the rest of this chapter, we examine the attribute grammar model in the light of the AVID system. We then outline some of the research into this area being actively pursued by the author and others here at Cornell.

4.3 Attribute Grammars

Attribute grammars [Knuth 68] allow the specification of semantic information about the language along with its syntax. [Demers et al 81] show how attribute grammars can be used in the incremental evaluation of change. They present an evaluator for attribute grammars that is incremental, reversible, and reasonably efficient. Very recently, [Reps 81] presented an "optimal"-time algorithm for the incremental evaluation of attribute grammars.

The methods described in these papers provide a general, clean solution to the problems of incremental change in a syntax-directed editing environment. However, in arguing for efficient execution, the authors make several requirements and assumptions, which do not hold for some interesting kinds of change. In particular, the approach requires that:

1) Each program element has associated with it the current value of the attributes meaningful at this element.
2) It is possible to check equality of attributes and to copy their values in unit time.

In particular, the "optimal" nature of the algorithm presented in [Reps 81] is dependent these assumptions.

The most interesting incremental change problem in AVID involves verification. To perform verification, the system gradually creates a large data structure that serves as the environment for a verification operation. In the simple attribute grammar model, this data structure is an attribute. Unfortunately, the size of this attribute can easily exceed that of the program. This means that keeping an instance of it at each program element can require space quadratic in the size of the program. Further, checking equality for two instances of this attribute cannot legitimately be assumed to take unit time.

We have termed the general version of this problem the large attribute problem. Other instances of large attributes are: the symbol table, global data flow information, and the print form of the program. Each of these has the property that neither of the assumptions given above is very realistic. Although some straightforward solutions have been proposed to variations on the large attribute problem (see, for example, [Skedzeleski 78]), they do not attempt to deal with incremental change, which creates an entirely new problem. Below, we discuss some of our current approaches to solving this problem.

It seems clear that the structure of the large attribute being considered and the nature of the functions upon it both have significant bearing on the problem. For example, if a symbol table is represented in the program as a linear linked list of declarations, the structure can be shared throughout the program, achieving a reasonable space efficiency. With this structure, incremental change is simple, since only the element in the list associated with the changed declaration needs to be modified.

With a linked list, however, a symbol table lookup can be quite expensive.

If a symbol table is represented as a B-tree ([Knuth 73]), lookup becomes cheaper, but the update problem is more acute. The problem is how to share the structure while still being able to perform change in a reasonable fashion. [Demers and Reps 81] suggest a solution to this problem. At each program element where a change occurs, do a standard update on the B-tree. Whenever a field in the B-tree would change, create a new node containing a copy of the old node in which the change has been made. Shared subtrees will be unaffected, and the space and time required will be at most equal to the logarithm of the size of the symbol table.

We see that for specific data structures and operations, good solutions to the large attribute problem seem possible. For other structures, such as the directed acyclic graph used in the congruence closure algorithm for verification, good solutions have not yet been discovered.

There is an alternative approach to dealing with large attributes. Rather than try to maintain any representation of the value of the attribute at each program element, attach a single instance of the attribute to a _cursor_. Whenever a calculation involving the attribute must be performed, the cursor must be moved through the program tree to the point of the calculation. As the cursor is moved, the attribute is continuously updated, so that it is always valid for the _current_ program element.

This approach avoids the problems of space blowup, but at a potential cost in time. If change is well localized, then this may be a better way to deal with it.

In considering this method, however, we quickly run into some fascinating problems. One common operation on large attributes is that of "opening" and "closing" blocks. The general definition of these operations is simple. Given the attribute in some state, if an "open" is performed it has no immediate effect on the value of the attribute. A series of changes can then be made to the attribute, which will behave exactly as they normally would. At this point a "close" operation may be performed. Its effect is to restore the value of the attribute to what it was immediately before the corresponding "open" operation. We say corresponding because these operations are paired and may be nested.

A general theoretical formulation of the problem of "open" and "close" operations on large attributes attached to the cursor might be stated as follows:

Problem: Given an array of locations and the operations open, close, modify, and retrieve, where open and close behave as described above, modify allows changing the value of a location and retrieve reads the current value of a location, is it possible to execute these operations in real time? If not, what is the complexity of this problem?

This seems to be a very hard problem. It is our conjecture that the problem cannot be solved in real time, but this is really no more than a guess. This problem is actually a whole collection of problems, since we can restrict the data structure being modified to a directed acyclic graph, to a tree, or to any other data structure that seems of interest. This seems a very interesting area for theoretical research.

To summarize, it seems to us that the attribute grammar approach to dealing with syntax-directed editors and the problem of incremental change has great potential. Research in this area is really just beginning, and we interestedly await the availability of some solid results.

They would have immediate and direct application to the AVID system, as well as a number of other similar systems.

CHAPTER 5

Determining Logical Dependency

5.1 Introduction

In designing AVID, a major goal was to provide the user of the system with as much useful information about the proof as possible. One feature that seemed potentially valuable was allowing the display of *logical dependency* within the proof. This feature would allow the interactive display of dependency information about any proof statement, in the form of a minimal set of proof statements from which it logically followed. Alternatively, the system could highlight the proof statements that directly depended on another statement. Since the AVID system interacts directly with the user, a further requirement was that this information be determined promptly.

In the PL/CV2 verifier used in AVID, the system automatically performs some simple deductions. For example, from the assertions "A => B" and "A", the assertion "B" can be deduced automatically, without explicit justification. As described in [Johnson 81], the basis for almost all these automatic deductions is a decision procedure for the theory of equality. The general method used by this decision procedure has been named *congruence closure*.

Congruence closure algorithms, in off-line form, have been described in [Nelson and Oppen 77] and [Downey, Sethi and Tarjan 80]. The on-line algorithm used in the PL/CV2 system is based on an algorithm given in [Kozen 77] as a solution to the uniform word problem for finitely presented algebras. This version of the congruence closure algorithm was first described in [Krafft 78] and was proved correct in

Chapter 6 of [Johnson 81].

In this chapter we present an extended version of the congruence closure algorithm implementing the PL/CV2 equality decision procedure. The extension allows us to solve the logical dependency problem proposed above for deductions made by the decision procedure. The extension consists of a new procedure and some changes to the existing procedures and data structures. The new procedure, given two expressions that the congruence closure algorithm can show to he equal, produces a certificate of their equality. The certificate is a minimal list of existing equalities from which the expression can he deduced to be equsl by applications of the axioms of equality.

As we will show later, adding the logical dependency extensions increases the running time of the standard congruence closure algorithm by only a small constant factor. The running time of the routine that produces certificates turns out to he almost-linear in the size of the certificate.

The chapter is organized as followa. In sections 5.2 and 5.3, we present a simplified version of the logical dependency algorithm; it determines equalities on only simple variables instead of on full expressions. The equality decision procedure in this case is almost exactly McIlroy and Morris's UNION-FIND algorithm [Aho, Hopcroft and Ullman 74]. In sections 5.4-5.6, we review the decision procedure for equality presented in [Johnson 81]. In section 5.7, we show how to integrate the algorithms of sections 5.3 and 5.6 to provide decision procedures for equality with certificates. In sections 5.8-5.11, we outline proofs of correctness for the new algorithms. In sections 5.12-5.14, we discuss their running time. Finally, in section 5.15 we present some conclusions and general observations.

5.2 A Simple Certification Algorithm

In this section and the next we introduce the reader to the basic concepts. To this end, we present a simplified version of our algorithm that solves the following problem: Execute a list of instructions consisting of equates of simple variables, queries on the equality of two simple variables, and requests for certification of equality between a pair of simple variables.

The sequence of pairs of simple variables that have been declared to be equal are referred to as EQS. Each equate instruction causes a pair to be added to the end of EQS. In response to a query on the equality of two simple variables, the algorithm returns true if their equality is deducible by means of repeated applications of reflexivity, symmetry, and/or transitivity on the pairs of variables in EQS, and false otherwise. A request for certification of equality for two simple variables returns a list of indices into EQS identifying a set of pairs that establish their equality. This list contains no duplicates.

Sequence EQS will not be used in the actual algorithms. It is simply used as an aid to presenting the problem being solved.

The algorithm used to determine equality for simple variables is based on the Union-Find algorithm [Aho, Hopcroft and Ullman 74]. Every variable initially represents an equivalence class consisting only of itself. Each time two variables are declared equal, a Union operation is performed on their equivalence classes.

In the table below, we show the contents of a node in the Union-Find tree. We will refer to the fields by the names given here in our discussions of the algorithm.

field	contents
label	The variable name associated with this node.
reprlink	Either a pointer to a node, or nil (marking the root) such that for all nodes x and y, x and y are in the same equivalence class if and only if their reprlinks lead to the same root node.
reprcount	The number of nodes in this node's equivalence class (if this node is a representative). This value is used as the weight for the Union-Find algorithm.
witnesslink	Either a pointer to a node, or nil. If it is a pointer to a node, then the input contained a proclaim equating this node and the node pointed to.
witnesslabel	The index of the proclaimed equality that caused the witnesslink to be introduced. If witnesslink is nil, this value is undefined.
marked	false (used only within the Certify procedure).
next	The next entry in this hash bucket.
previous	The previous entry in this hash bucket. These fields are used to chain nodes together in a bucket used by the hash function of the Find algorithm.

The reader should be able to see that the Find algorithm of Figure 5.1, the Compare algorithm of Figure 5.2, and the Equate algorithm of Figure 5.3 (ignoring lines 12-21 of that algorithm) make use of the information described above to implement a weighted Union-Find without path compression.

```
Find(x):
¢ Return an entry for x in the data structure, creating it ¢
¢ if necessary.                                            ¢
begin
     h := hash(x)
;    Search bucket[h] for x
;    if   the search fails to find a node n
     then n:= a pointer to a newly allocated node
     ;     Put n in bucket[h], setting n.next and n.previous
     ;     n.label := x
     ;     n.reprcount := 1
     ;     n.reprlink, n.witnesslink := nil, nil
     fi
;    Find := n
end
```

Figure 5.1 Algorithm Find

```
Compare(xx, yy):
¢ Determine if varisbles xx and yy are equal ¢
begin
     x := Find(xx) ; y := Find(yy)
;    while x.reprlink ≠ nil do x := x.reprlink od
;    while y.reprlink ≠ nil do y := y.reprlink od
;    Compare := (x = y)
end
```

Figure 5.2 Algorithm Compare

To be able to certify equalities, the algorithm must keep a record of which variables were declared equal by input assertions. This is done by recording these asserted equalities as witness edges between nodes of the Union-Find graph. The field "witnesslink" contains a pointer to represent this edge. Thus, there are either zero or one witness edges for a node. Each witness edge is labeled with the index in EQS of the input assertion of equality that caused its creation. This lahel is stored in the "witnesslabel" field of the node. The reader should observe that witness edges will coincide with Union-Find edges only when the nodes being declared equal are the representatives of their respective equivalence classes.

To certify that two variables are equal, we need only find a path through the witness edges from one variable to the other; the labels on the edges of the path form the desired certificate.

The problem is to find this path as quickly as possible. To do so, we established the following invariant:

Invariant: The directed witness path from any member of an equivalence class ends in the (Union-Find) representative of that equivalence class.

With this invariant, finding a witness path between two equivalent nodes is straightforward. From each node, trace the witness path to the representative. The symmetric difference of these paths will give a path along witness edges between the nodes. This is the algorithm used by the Certify routine of Figure 5.4.

We argue that the nodes and witness edges form a directed graph in which each node has out-degree zero or one. By the invariant, this graph is acyclic (since every path ends); thus the graph is a forest. Further, the roots of this forest are exactly the Union-Find representatives (since every path ends at a Union-Find representative); thus, each witness equivalence class is a single tree in the forest. Observe that the height of this tree may be the number of nodes in the equivalence class. The height of the Union-Find tree, on the other hand, will be bounded by the logarithm of the number of nodes in the equivalence class.

In designing the complete algorithm we are faced with one more

```
Equate(xx, yy):
¢ Assert the equality of variables xx and yy ¢
begin
      Equate_index := Equate_index + 1
      x := Find(xx) ; y := Find(yy)
;     while x.reprlink ≠ nil do x := x.reprlink od
;     while y.reprlink ≠ nil do y := y.reprlink od
;     if  x ≠ y
      then ¢ Let x be the name of the new representative ¢
           ;    if y.reprcount > x.reprcount
                     then x,y := y,x fi
      ;    y.reprlink := x ; x.reprcount := x.reprcount + y.reprcount
           ¢ Reverse the witness path (yy, ..., y)              ¢
           ;    w := yy
           ;    previouslabel := w.witnesslabel ; v := w.witnesslink
           ;    while v ≠ nil do
                ;    previouslabel, v.witnesslabel := v.witnesslabel,
                          previouslabel
                ;    v, v.witnesslink, w := v.witnesslink, w, v
                od
           ;    yy.witnesslabel := Equate_index ; yy.witnesslink := xx
      fi
end
```

Figure 5.3 Algorithm Equate

major problem: How can we perform Equate (Union) operations while maintaining the invariant necessary for Certification? The solution is as follows: To Equate two variables, choose the variable in the smaller equivalence class, reverse the edges on the witness path from that variable to its representative, and add the new witness edge from that variable to the other variable. Then do a normal (weighted) Union on the two equivalence classes.

Let us examine this solution, making use of the example shown in Figures 5.5 and 5.6. In this example, b and d are the initial representatives of the two equivalence classes. An Equate operation is being done on **a** and **e**.

The first step is to choose the variable in the smaller equivalence class. That variable is **e**. After choosing a variable, the next step is to reverse the edges on the witness path from this variable (**e**) to its representative (**d**). By our invariant, at the start of the operation the witness path out of every node in the equivalence class leads eventually to the representative (**d**). After reversing the edges, there is a path from the representative (**d**) to the variable (**e**). Therefore, there is now a path from every node in the equivalence class to that varible.

The next step is to add a witness edge from this variable (**e**) to the variable (**a**) we are Equating with it. Since by the invariant there was already a witness path from **a** to its representative, **b**, there is now a witness path from every node in the smaller equivalence class (**d**, **e**, **f**) to **b**.

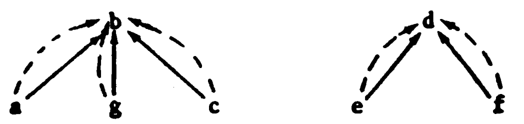
The final step is to take the Union of the two equivalence classes. This will make **b** the representative of the combined equivalence class, and the invariant will have been reestablished.

Returning to the problem of producing certificates, consider the data structure of Figure 5.6. To certify that **f**=**e**, the algorithm must calculate the symmetric difference of the witness path from **f** to the representative (**b**) and the path from **e** to **b**. If the algorithm fully traced these paths, it would follow two edges that could be completely ignored (**e**-**a**, and **a**-**b**) twice. In the Certify algorithm shown in Figure 5.4 the symmetric difference of the paths is calculated while avoiding this extra effort. This is done by stepping along the witness paths from the two nodes to the representative in parallel, using field "marked" to

```
Certify(xx, yy):
¢ Assuming that xx and yy are equal, output a list of previously ¢
¢ asserted equalities to certify it.                              ¢
begin
      x := Find(xx) ; y := Find(yy)
;     xl := x ; yl := y
      ¢ Find the lowest common ancestor (lca) of x and y in the     ¢
      ¢ witness tree                                                ¢
      ;   while ¬ xl.marked and ¬ yl.marked and x ≠ y
          do
               if   xl.witnesslink ≠ nil
               then xl.marked := true ; xl := xl.witnesslink
               fi
          ;    if   yl.witnesslink ≠ nil
               then yl.marked := true ; yl := yl.witnesslink
               fi
          od
      ;   if   xl.marked or xl = yl
          then lca := xl ; overshoot := yl
          else lca := yl ; overshoot := xl
          fi
      ¢ Clear any marks above the lowest common ancestor         ¢
      ;   t := lca
      ;   while t ≠ overshoot
          do   t.marked := false ; t := t.witnesslink od
      for t in {x, y}
      do  while t ≠ lca
          do   Output t.witnesslabel ; t.marked := false
          ;    t := t.witnesslink
          od
      od
end
```

Figure 5.4 Algorithm Certify

mark as we go. As soon as the marked paths intersect (at the nodes' lowest common ancestor in the witness tree), a simple path between the nodes will have been found. This technique serves to bound the amount of



(Dashed lines indicate witness links, solid lines indicate representative links.)

Figure 5.5 Data structure after a=b, b=c, d=e, d=f, g=b

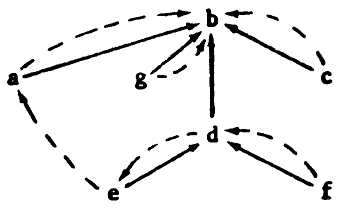


Figure 5.6 Data structure from Figure 5.5 after a=e

work done by Certify to almost-linear in the number of edges in shortest witness path between nodes.

5.3 Solving a Slightly Harder Problem

In this section we extend the algorithm of the previous section to allow a single instruction to request certificates of equality for an arbitrary number of pairs of simple variables. Again, this is a list of indices into EQS. Avoiding duplicates makes this a more difficult problem.

In determining logical dependency in the full congruence closure algorithm we need to deal with the problems posed by the presence of operands. We will reserve a complete discussion of these problems for section 5.7. For the moment, note that certifying the equality of two

<u>field</u>	<u>contents</u>
witnessrepr	Within the MultiCertify routine, this is either a pointer to a node or <u>nil</u> such that for all nodes x and y, x and y are in the same witness equivalence class if and only if their witness paths end in the same root node.
witnesscount	If this node is a witness representative, this is the count of the number of nodes in its witness equivalence class. (Used only within the MultiCertify routine.)
realroot	Within the MultiCertify routine, if this node is the representative of a witness equivalence class, this is a pointer to the "real" root of that equivalence class. Initialized to point to the node itself. (See text for a fuller explanation.)

Figure 5.7 The additional node fields for a slightly harder problem

expressions involves: (1) observing that the outermost operators of the two expressions are equal, and (2) certifying that their operands are pairwise equal. Clearly, finding a minimal certificate for the one queried equality is equivalent to finding a minimal set of proclaimed equalities from which <u>all</u> the operand equalities can be inferred. In other words, we will want to consider the operand pairs collectively and produce one minimal certificate for all of them.

We now present a change to the Certify algorithm to do this. In Figure 5.7, we describe the new fields that must be added to the graph nodes to solve this problem.

The new Certify procedure changes the graph. In this and all following sections, we will assume that at the end of a series of Certify instructions, all these changes are undone in some fashion. We will simply note here that this can be done in time at most double that of the simple destructive algorithm.

To solve the new problem, we must avoid duplicating emitted equalities. At the same time, we must keep the running time of the algorithm as close as possible to O("number of emitted equalities"). Our solution is as follows: Form equivalence classes of all variables that can be shown to be equivalent by emitted equalities. These equivalence classes will be referred to as witness equivalence classes. The equality of two varisbles in the same witness equivalence class can be deduced from previously emitted certifying equalities. If two variables are not in the same witness equivalence class, then, as in the simple Certify algorithm, trace the shortest witness path between them. This time, however, emit only witness edges connecting previously disjoint witness equivalence classes, forming the union of these classes as the labels on the witness edges are emitted.

A full specification of this algorithm (MultiCertify) is shown in Figure 5.8. To implement the equivalence classes, we again make use of the Union-Find algorithm. In this case, the version used is the weighted, path-compressing, Union-Find, as presented in [Aho, Hopcroft and Ullman 74]. The functions that implement the Union-Find operation, PathFind and PathUnion, are the standard algorithms, and are therefore not explicitly listed. PathFind is a function that returns the root node of the witness equivalence class for a node. PathUnion forms the union of two witness equivalence classes, returning the root node in the new class. These routines use the "witnessrepr" field for representative edges, and the "witnesscount" field for the weights on the nodes. Since the weighted, path-compressing, Union-Find is being used, the running time of the algorithm will be $O(N\ \alpha(N))$, where α is Tarjan's α function [Tarjan 75]. For the MultiCertify algorithm, N will be the number of emitted equalities.

The MultiCertify operation makes use of one more special field.

```
MultiCertify(xx, yy):
¢ Assuming that xx and yy are equal, output a list of previously ¢
¢ asserted equalities to certify their equality.                 ¢
begin
    x := Find(xx) ; y := Find(yy)
;   ToCertify := {(x, y)}
;   while ToCertify is not empty
    do  Remove an element (x, y) from ToCertify
    ;   x := PathFind(x) ; y := PathFind(y)
    ;   x := x.realroot ; y := y.realroot
    ;   xl := x ; yl := y
        ¢ Find the lowest common ancestor (lca) in witness tree ¢
        ;   while ¬ xl.marked and ¬ yl.marked and x ≠ y
            do
                if  xl.witnesslink ≠ nil
                then xl.marked := true
                ;   xl := PathFind(xl.witnesslink) ; xl := xl.realroot
                fi
            ;   if  yl.witnesslink ≠ nil
                then yl.marked := true
                ;   yl := PathFind(yl.witnesslink) ; yl := yl.realroot
                fi
            od
        ;   if  xl.marked or xl = yl
            then lca := xl ; overshoot := yl
            else lca := yl ; overshoot := xl
            fi
        ¢ Clear any marks above the lowest common ancestor     ¢
        ;   t := lca
        ;   while t ≠ overshoot
            do  t.marked := false
            ;   t := PathFind(t.witnesslink) ; t := t.realroot
            od
    ;   for t in {x, y}
        do  while t ≠ lca
            do  Output t.witnesslabel ; t.marked := false
            ;   tl := PathFind(t.witnesslink) ; tl := tl.realroot
            ;   u := PathUnion(t, tl) ; u.realroot, t := tl, tl
            od
        od
    od
end
```

Figure 5.8 Algorithm MultiCertify

Each witness equivalence class must have a witness link exiting from it going toward the representative of the equality equivalence class. The real root of a witness equivalence class is the node that contains this witness link. This may not be the same node as that chosen by PathUnion as the Union-Find representative for the witness equivalence class. For the purpose of tracing the witness path, however, the real root is the node that must be taken as the one representing the equivalence class, since it is the node that should be used when following the witness links.

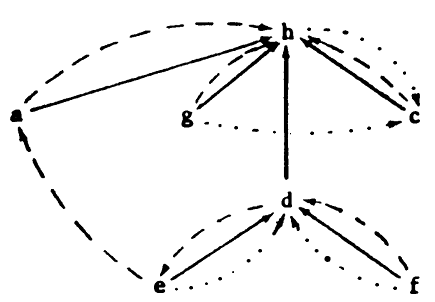
In order to find the real root of the equivalence class, the algorithm maintains for each PathFind representative the field "realroot". This points to the real root of the equivalence class, and it is the "witnesslink" from this real root that must be followed next. This field should be initialized for each node to point to the node itself.

In Figure 5.9, we show the effect of two simple MultiCertify operations. Note that the data structure contains three witness equivalence classes: {e, d, f}, {a} and {g, h, c}. Note also that a representative node in the simple equivalence relation is not necessarily a representative in the witness equivalence relation. In Figure 5.10, we show the data structure of Figure 5.9 after one more MultiCertify operation. Observe that the equivalence classes have all collapsed into one. As all the witness edges in the data structure have now had their labels emitted, any further MultiCertify operation will have no effect.

5.4 An Overview of the Equality Algorithms

The presentation in sections 5.4-5.6 has been adapted from that presented in chapter 6 of [Johnson 81] with the author's permission. For a complete presentation of this algorithm and a detailed proof of its correctness, the reader is referred to [Johnson 81].

Suppose we are given an alphabet A of function symbols. The syntactic object "expression" can be inductively defined as follows. If f



(Dashed lines indicate witness links, solid lines indicate representative links. Witness representative links are shown as dotted lines.)

Figure 5.9 Figure 5.6 after MultiCertify(g,c) and MultiCertify(e,f)

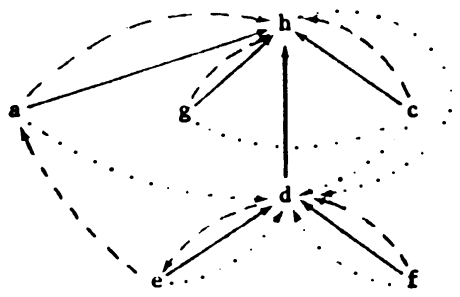


Figure 5.10 Data structure from Figure 5.9 after MultiCertify(f,c)

is an element of A, and x_1, x_2, ... x_k are zero or more expressions, then $f(x_1, x_2, \ldots x_k)$ is also an expression. The only expressions are objects that can be obtained by repeated applications of the preceding rules.

As in the simple variable case we let EQS represent the sequence of ordered pairs of expressions that have been declared to be "equal." Adding the pair {x, y} to EQS is an indication that in any context where x appears, y would do equally well. Two expressions are defined to be "equivalent" whenever one can be transformed to the other by a sequence of substitutions of expressions for their equals. (Observe that reflexivity will be represented by the null substitution sequence.) The equivalence of x and y in this sense is expressed as $x =_s y$. The input to Algorithm Eq is a series of instructions of the form "Assert(x,y)" and "Query(x,y)". The operation of algorithm Eq on these instructions is described in terms of the set EQS. (EQS is not used in the computations; it is merely used for specification). Initially, EQS is empty. When an Assert(x,y) instruction is encountered, the pair {x,y} is added to EQS. When a Query(x,y) instruction is encountered, a boolean value is output indicating whether $x =_s y$, where $=_s$ is the relation defined by the current value of EQS.

The problem of computing $=_s$ from EQS is equivalent to the uniform word problem for finitely presented algebras. Several algorithma for solving this problem have appeared in the literature. [Nelson and Oppen 77] and [Downey, Sethi and Tarjan 80] give algorithms that run in O(N log N) time. However, these algorithms are offline, which means that all the Assert instructions must come before all the Query instructions. [Samet 80] gives an online solution to the problem, but does not give a complete runtime analysis. He simply states that the worst case runtime (his algorithm uses hashing) for a single Assert instruction is quadratic. Algorithm Eq processes a list of instructions in expected time of O(N log N) (it also uses hashing), where N is the total size of all the expressions in the input.

5.5 An Outline of Algorithm Eq

A linked data structure is used to represent expressions. Routine EqFind(e) is used to obtain the linked representation of the expression

e. If e is of the form $f(x_1, x_2, \ldots, x_k)$, EqFind first calls itself recursively to obtain a node n_i representing each x_i. The node representing e contains the label f and pointers called operand links to the nodes $n_1, n_2, \ldots n_j$. A hashed search is used to determine if the data structure already contains such a node; if not, a new one is created.

The data structure is actually a directed acyclic graph, in which each node represents an expression. For example, if EqFind is called with arguments a(c(),d(g(),h())), b(d(g(),h()),f()) and e(h(),i()), the result will be the graph in Figure 5.11.

The asserted equality of two expressions e_1 and e_2 is recorded in the graph by connecting the node representing e_1 to the node representing e_2 using a pointer "reprlink". The nodes of the graph are partitioned into Union-Find equivalence classes, where all the nodes of a given equivalence class are connected by reprlinks to form a tree. One node in each equivalence class, the root of the tree, is singled out as the "representative" of the class. This node uniquely identifies the equivalence class.

Operand links are allowed to point only to representative nodes. Two equivalence classes are merged by connecting their representatives n_1 and n_2 with a reprlink. One of the nodes, say n_1, is chosen as the representative of the new equivalence class. All the operand links pointing to n_2 must be located and changed to point to n_1.

A node represents a class of equivalent expressions. For example, the node with label a in the graph in Figure 5.12 (reprlinks are shown as dashed lines in figures) represents the following set of formulas:

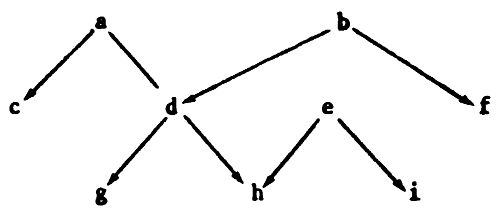


Figure 5.11 A graph created by EqFind

{a(b(),d()), a(b(),e()), a(c(),d()), a(c(),e())}.

Once reprlinks are introduced, the graph is no longer constrained to be acyclic. Nodes in graphs with cycles can represent an infinite set of expressions. For example, the graph in Figure 5.13 is the result of asserting x and f(x) to be equal. The node with label x represents the infinite set {x(), f(x()), f(f(x())), f(f(f(x()))), ...}.

A Query(x,y) instruction is processed by testing whether the two expressions x and y are represented by the same node. For this strategy to work, it is crucial that enough reprlinks are added so that all nodes representing the same expression are in the same equivalence class.

Consider the graph in Figure 5.14a. As the result of the instruction Assert(d(), e()), the nodes representing d() and e() are "merged" by adding the reprlink and moving the operand links as shown in Figure 5.14b. But now the two nodes with b as a label, which are both representative nodes, have identical operand links. Only one of these nodes can be used as the representative of, for example, b(d(),e()). Since the two nodes currently have equal weight, that is they are the representatives of the same number of nodes (one each), the node on the right is arbitrarily chosen as the representative. The two nodes are then merged, as

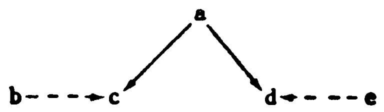


Figure 5.12 A graph with reprlinks

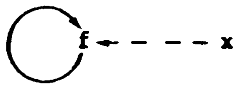


Figure 5.13 A graph with a cycle

shown in Figure 5.14c. But now nodes that have the label a are equivalent and must be merged, as shown in Figure 5.14d.

The problem is to find all the nodes that need to be merged. Algorithm Eq maintains another set of links, called container links, that are used to locate the nodes that need to be updated when a reprlink is added.

5.6 The Details of Algorithm Eq

Algorithm Eq consists of the routines shown in Figures 5.15-5.18 and a global data structure, or "graph", as it will be called. The graph has three parts, the first of which is a collection of "nodes". The fields of the node are listed below.

field	contents
label	An input function name.
operands	A list of pointers to the nodes representing the operands for this function node.

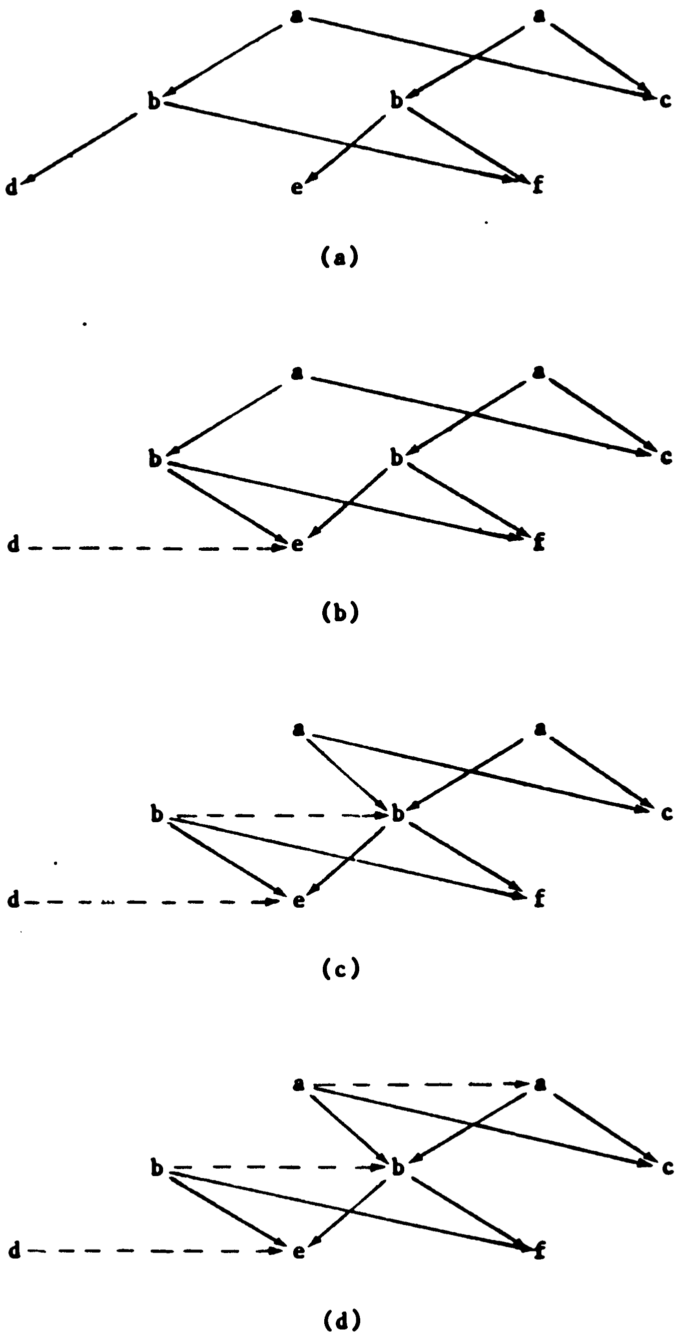


Figure 5.14 Merging equivalent nodes

```
Eq:
¢ Process a list of instructions ¢
begin
     while there are instructions left
     do    Read an instruction i
     ;     if    i is of the form Assert(x, y)
           then  EqAssert(x, y)
           elif  i is of the form Query(x, y)
           then  Output EqQuery(x, y)
           fi
     od
end
```

Figure 5.15 Algorithm Eq

```
EqQuery(xx, yy):
¢ Determine if expressions xx and yy are equal ¢
begin
        nx := EqFind(xx)
;       ny := EqFind(yy)
;       EqQuery := (nx = ny)
end
```

Figure 5.16 Algorithm EqQuery

containers A list of pointers to operand fields of the nodes for which this is an operand. The operands and containers fields are heads of singly-linked lists of nodes. For the purposes of this discussion, we will assume that nodes are flexible, and the operand and container fields contain variable-sized lists.

reprlink either a pointer to a node, or nil (marking the root) such that for all nodes x and y, x and y are in the same equivalence class if and only if their reprlinks lead to the same root node.

reprcount	the number of nodes in this node's equivalence class (if this node is a representative).
next	the next entry in this hash bucket.
previous	the previous entry in this hash bucket.
eligible	a boolean value, = "this node is in a hash bucket".

The term "node" will be used to refer to an area of storage, and not its contents. For example, if n_1 and n_2 are nodes, $n_1 \neq n_2$ means that n_1 and n_2 occupy different locations in memory, but does not exclude the possibility that corresponding fields of n_1 and n_2 have identical contents. Initially, there are no nodes. As the algorithm is executed, new nodes can be allocated, but nodes are never deallocated.

The second part of the graph is "bucket," an array of pointers to nodes. Each element of bucket is the head of a list of nodes that is doubly linked through the next and previous fields of each node. If a node is in a bucket, the node will be said to be "eligible"; otherwise it is said to be "ineligible". A bit in each node is used to keep track of whether the node is eligible; the bit is set when a node is put in a bucket, and cleared when it is removed.

Function "hash" takes as input a label and a list of the addresses of zero or more nodes and produces the index of a bucket. Initially, there are no nodes, so that all the buckets are empty.

The third part of the graph is "ToDo," which is a set of pairs of pointers to nodes. The notation "$\{n_1, n_2\}$ is in ToDo" will be defined to mean that either (n_1, n_2) or (n_2, n_1) is in ToDo. Initially, ToDo is empty.

```
EqFind(e):
¢ Add expression e to the graph ¢
begin
     e will be of the form f(x1, x2, ... xk)
        for a function symbol f and some k >= 0
  ;  for each xi do a[i] := EqFind(xi) od
  ;  h := hash(f, (a[1], a[2], ... a[k]))
  ;  Search bucket[h] for a node with operator f
        and operands a[1], a[2], ... a[k]
  ;  if   the search succeeds in finding a node n
     then while n.reprlink ≠ nil do n := n.reprlink od
     else n := a pointer to a newly allocated node
     ;    Put n in bucket[h]
     ;    n.label := f
     ;    n.operands := (a[1], a[2], ... a[k])
     ;    Add a pointer to the i'th operand field of n
             to the container list of a[i], for each i in {1 ... k}
     ;    n.containers := the empty list
     ;    n.reprlink := nil
     ;    n.reprcount := 1
     fi
  ;  EqFind := n
end
```

Figure 5.17 Algorithm EqFind

5.7 A Decision Procedure for Equality with Certification

In this section we show how to combine the simple certification algorithm of section 5.3 with the equality decision procedure of section 5.6 to produce the complete algorithm.

The major problem in combining the two previous algorithms lies in dealing with operands. In adding operands, we have added a new equality axiom for congruence. In the congruence closure algorithm, we apply this axiom when two expressions have the same outermost operator, and their

```
(01) EqAssert(xx, yy):
(02) ¢ assert the equality of expressions xx and yy ¢
(03) begin
(04)      nx := EqFind(xx) ; ny := EqFind(yy)
(05) ;    Add (nx, ny) to ToDo
(06)      ¢ add {xx, yy} to EQS ¢
(07) ;    EqLoop:
(08)      while ToDo is not empty
(09)      do   Remove an element (x, y) from ToDo
(10)      ;    while x.reprlink ≠ nil do x := x.reprlink od
(11)      ;    while y.reprlink ≠ nil do y := y.reprlink od
(12)      ;    if   x ≠ y
(13)           then
(14)                ¢ Let x be the name of the new representative ¢
(15)                     if y.reprcount > x.reprcount
(14)                     then x, y := y, x fi
(15)           ;    y.reprlink := x
(16)           ;    x.reprcount := x.reprcount + y.reprcount
(17)           ;    ConLoop:
(18)                for every w on y.containers
(19)                do  w.operand := x
(20)                ;    z := a pointer to the node containing w.operand
(21)                ;    if z is eligible
(22)                     then Remove z from the bucket it is in
(23)                     ;    h := hash(z.label, z.operands)
(24)                     ;    Search bucket[h] for a node with
(25)                              the same label and operands as z
(26)                     ;    if   the search finds a duplicate d
(27)                          then Add (d, z) to ToDo
(28)                          else Put z in bucket[h]
(29)                          fi
(30)                     fi
(31)                od
(32)           ;    Append y.containers to x.containers
(33)           fi
(34)      od
(35) end
```

Figure 5.18 Algorithm EqAssert

operands become equal. This occurs at line 27 of the EqAssert Algorithm (Figure 5.18).

How should we certify an equality that comes about by this congruence closure process? Since the certificates do not explicitly include references to reflexivity, we need only produce certificates for the equal pairs of operands. There is a cost involved in applying the congruence axioms, however, even if they do not show up explicitly in the certificate. We will consider this cost when calculating the running time of the algorithm.

We use normal witness edges to connect expressions that the congruence closure algorithm has internally declared to be equal. To identify these witness edges to the certification algorithm, we introduce a unique witness label, "equaloperands". When this label is encountered during a certify, the algorithm knows that it must certify the pairwise equality of the original operands of the two expressions.

Below, we list the fields for a node in the complete algorithm.

field	contents
label	An input function name.
operands	A list of pointers to the representative nodes of the operands for this function node.
originalops	A list of pointers to the nodes that were the actual original operands for this function.
containers	A list of pointers to operand fields of the nodes for which this is an operand representative.
reprlink	Either a pointer to a node, or nil (marking the root) such that for all nodes x and y, x and y are in the same equivalence class if and only if their reprlinks lead to the same root node.
reprcount	The number of nodes in this node's equivalence class (if this node is a representative).
witnesslink	Either a pointer to a node, or nil. If it is a pointer to a node, then the input contained a proclaim equating this node

```
ExtEq:
¢ Process a list of instructions ¢
begin
     while there are instructions left
     do    Read an instruction i
     ;     if    i is of the form Assert(x, y)
           then  ExtAssert(x, y)
           elif  i is of the form Query(x, y)
           then  Output ExtQuery(x, y)
           elif  i is of the form Certify(x, y)
           then  Output ExtCertify(x, y)
           fi
     od
end
```

Figure 5.19 Algorithm ExtEq

```
ExtQuery(xx, yy):
¢ Determine if expressions xx and yy are equal ¢
begin
        x := ExtFind(xx)
        while x.reprlink ≠ nil do x := x.reprlink od
;       y := ExtFind(yy)
        while y.reprlink ≠ nil do y := y.reprlink od
;       ExtQuery := (x = y)
end
```

Figure 5.20 Algorithm ExtQuery

and the other node.

witnesslabel The index of the proclaimed equality that caused the witnesslink to be introduced or the distinguished label "equaloperands". If witnesslink is <u>nil</u>, this value is undefined.

witnessrepr	Within the ExtCertify routine, this is either a pointer to a node or <u>nil</u>, such that for all nodes x and y, x and y are in the same witness equivalence class if and only if their witnessreprs lead to the same root node.
witnesscount	If this node is a witness representative, this is the number of nodes in its witness equivalence class. (Used only within the ExtCertify routine.)
realroot	Within the ExtCertify routine, if this node is the representative of a witness equivalence class, this is a pointer to the "real" root of that equivalence class. Initialized to point to the current node. (See text for a fuller explanation).
next	Points to the next entry in this hash bucket.
previous	Points to the previous entry in this hash bucket.
orignext	Points to the next entry in the original hash bucket.
origprevious	Points to the previous entry in the original hash bucket.
eligible	a boolean value, = "this node is in a hash bucket".
marked	<u>false</u> (used only within the ExtCertify procedure).

Except for the fields "originalops", "orignext" and "origprevious", these fields are simply the obvious combination of those given in sections 5.2 and 5.6. The new fields are needed to certify the equality of expressions linked by "equaloperands" witness links. In the congruence closure algorithm described in section 5.6, the operand list for an operator was continuously modified. This was done so that the operands of a node were always the representatives of the original operands. To process "equaloperands" witness links, the algorithm must be able to find the original operands of the expressions involved, in order to recur-

```
ExtFind(e):
¢ Return index of expression e in the graph (perhaps inserting it) ¢
begin
    e will be of the form f(x1, x2, ... xk)
       for a function symbol f and some k >= 0
;   for each xi do b[i] := ExtFind(xi) od
;   oh := hash(f, (b[1], b[2], ... b[k]))
;   Search origbucket[oh] for a node with operator f
       and operands b[1], b[2], ... b[k]
;   if  the search does not succeed in finding a node n
    then n := a pointer to a newly allocated node
    ;   Put n in origbucket[oh]
    ;   n.label := f
    ;   n.witnessrepr := nil ; n.witnesscount := 1
    ;   n.originalops := (b[1], b[2], ... b[k])
    ;   for each i in {1 ... k}
        do  a[i] := b[i]
        ;   while a[i].reprlink ≠ nil do a[i] := a[i].reprlink od
        od
    ;   h := hash(f, (a[1], a[2], ... a[k]))
    ;   Search bucket[h] for a node with operator f
           and operands a[1], a[2], ... a[k]
    ;   if  the search succeeds in finding a node m
        then rm := m
        ;   while m.reprlink ≠ nil do m := m.reprlink od
        ;   n.reprlink := m ;  m.reprcount := m.reprcount + 1
        ;   n.witnesslink := rm ; n.witnesslabel := "equaloperands"
        else Put n in bucket[h]
        ;   n.operands := (a[1], a[2], ... a[k])
        ;   Add a pointer to the i'th operand field of n to
               the container list of a[i], for each i in {1 ... k}
        ;   n.containers := the empty list
        ;   n.reprlink := nil ; n.reprcount := 1
        ;   n.witnesslink := nil ; n.witnesslabel := ""
        fi
    fi
;   ExtFind := n
end
```

Figure 5.21 Algorithm ExtFind

sively certify their equality. The "originalops" field provides a list of these operands.

A related problem arises when an expression is being inserted into the data structure. Again, the algorithm must be able to find the original operands of the expression. The "orignext" and "origprevious" fields are simply links in the hash table "origbucket". The "origbucket" hash

```
ExtAssert(xx, yy):
¢ Assert the equality of expressions xx and yy ¢
begin
     Equate_index := Equate_index + 1
  ;  nx := ExtFind(xx) ; ny := ExtFind(yy)
  ;  Add (nx, ny, Equate_index) to ToDo
     ¢ Add {xx, yy} to EQS ¢
  ;  EqLoop:
     while ToDo is not empty
     do   Remove an element (rx, ry, witness) from ToDo
     ;    x := rx ; y := ry
     ;    while x.reprlink ≠ nil do x := x.reprlink od
     ;    while y.reprlink ≠ nil do y := y.reprlink od
     ;    if  x ≠ y
          then
              ¢ Let x be the name of the new representative ¢
                  if y.reprcount > x.reprcount
                  then swap x and y fi
          ;   y.reprlink := x
          ;   x.reprcount := x.reprcount + y.reprcount
              ¢ Reverse the witness path (ry,...,old representative) ¢
              ;   w, previouslabel, v := ry,
                      w.witnesslabel, w.witnesslink
              ;   v := w.witnesslink
              ;   while v ≠ nil do
                  ;   previouslabel, v.witnesslabel :=
                            v.witnesslabel, previouslabel
                  ;   v.witnesslink, w, v := w, v, v.witnesslink
                  od
              ;   ry.witnesslabel := witness ; ry.witnesslink := rx
          ;   ConLoop:
              for every w on y.containers
              do  w.operand := x
              ;   z := a pointer to the node containing w.operand
              ;   if z is eligible
                  then Remove z from the bucket it is in
                  ;   h := hash(z.label, z.operands)
                  ;   Search bucket[h] for a node with
                          the same label and operands as z
                  ;   if   the search finds a duplicate d
                      then Add (d, z, "equaloperands") to ToDo
                      else Put z in bucket[h]
                      fi
                  fi
              od
          ;   Append y.containers to x.containers
          fi
     od
end
```

Figure 5.22 Algorithm ExtAssert

table is used in exactly the same way "bucket" is used in the simple congruence closure algorithm, but instead of accessing the expression based on the current representatives of its operands, it accesses the expression based on its original operands.

In Figures 5.19-5.23 we give the complete algorithm for the decision procedure for equality with logical dependency. In comparing these algorithms with their counterparts in section 5.6, note that the ExtFind function returns a pointer to the expression itself, and not to its representative as EqFind does. This is done so that the ExtCertify function can use ExtFind to find the expressions to be certified. Obviously, when doing a certify, the original expressions, and not the representatives, must be returned. Aside from this difference, the "Ext" algorithms are functionally equivalent to the "Eq" algorithms when deciding equalities.

The ExtCertify algorithm is shown in Figure 5.23. The need to certify arbitrary pairs of operands while still emitting a minimal set of certificates requires that we incorporate the MultiCertify algorithm of Figure 5.8 rather than the simpler Certify algorithm of Figure 5.4.

Again, as in section 5.3, the ExtCertify operation is destructive, in that it changes values in the data structure, and does not bother to reset them on exit. We assume that some sort of audit trail is being kept during the execution of ExtCertify, and on exit all values are restored to their values before ExtCertify was called.

As in section 5.3, the PathFind and PathUnion functions implement a weighted, path-compressing Union-Find. The "witnessrepr" and "witnesscount" fields are used by these functions. Note that this Union-Find operation is completely independent of other operations on the data structure. Only the PathFind and PathUnion functions will affect the "witnessrepr" and "witnesscount" fields, and, at the end of a call to ExtCertify, these fields are assumed to be reset to *nil* and zero respectively.

5.8 Correctness of the Algorithm

In the next two sections we argue for the correctness of the logical dependency algorithm described in this report. We present an informal argument, and do not attempt to approach the level of detail of [Johnson 81]. In particular, we argue for correctness only for the general algorithm. We will not be concerned with the details of the programs given in the sections above.

We need to make two basic arguments about the certificates produced by our algorithm. First, we must show that the queried equality can be deduced from the list of equalities in the certificate, using only the substitution and reflexive axioms for equality. Second, we must show that the certificate is minimal, i.e. that if we remove any equality from the certificate, then it is no longer possible to deduce the certified equality. In the arguments below, we will take as given that the congruence closure algorithm correctly implements a decision procedure for equality.

5.9 Why are Certificate Equalities Sufficient for the Deduction?

We claim that the equality certificate for two expressions gives us a "program" of ExtAssert commands that will, if all the steps in the program are executed, put the two expressions into the same congruence closure equivalence class. Ignoring for a moment the problem of operands,

```
ExtCertify(xx, yy):
¢ Assuming that xx and yy are equal, output a list of previously ¢
¢ asserted equalities to certify it.                             ¢
begin
    x := ExtFind(xx) ; y := ExtFind(yy)
;   Add (x, y) to ToCertify
;   while ToCertify is not empty
    do  Remove an element (x, y) from ToCertify
    ;   x := PathFind(x) ; y := PathFind(y)
    ;   x := x.realroot ; y := y.realroot
    ;   xl := x ; yl := y
        ¢ Find the lowest common ancestor (lca) in witness tree ¢
    ;   while ¬ xl.marked and ¬ yl.marked and x ≠ y
        do
            if  xl.witnesslink ≠ nil
            then xl.marked := true
            ;   xl := PathFind(xl.witnesslink) ; xl := xl.realroot
            fi
            if  yl.witnesslink ≠ nil
            then yl.marked := true
            ;   yl := PathFind(yl.witnesslink) ; yl := yl.realroot
            fi
        od
    ;   if  xl.marked or xl = yl
        then lca := xl ; overshoot := yl
        else lca := yl ; overshoot := xl
        fi
        ¢ Clear any marks above the lowest common ancestor    ¢
    ;   t := lca
    ;   while t ≠ overshoot
        do  t.marked := false
        ;   t := PathFind(t.witnesslink) ; t := t.realroot
        od
    ;   for t in {x, y}
        do  while t ≠ lca
            do  t.marked := false
            ;   if t.witnesslabel = "equaloperands"
                then Let the original operands of t.realroot be
                         a[1], ... a[k] and let the original operands
                         operands of t.witnesslink be b[1], ... b[k]
                ;   for each i in {1 ... k}
                    do  Add (a[i], b[i])) to ToCertify od
                else Output t.witnesslabel
                fi
            ;   tl := PathFind(t.witnesslink) ; tl := tl.realroot
            ;   u := PathUnion(t, tl) ; u.realroot, t := tl, tl
            od
        od
    od
end
```

Figure 5.23 Algorithm ExtCertify

if we begin at one of the queried expressions (at one end of the certificate) and follow the witness path (equalities in the certificate) doing ExtAsserts, each such assert puts the next expression on the path into the single growing equivalence class. The final step will put the second queried expression into the equivalence class. Since the congruence closure algorithm correctly implements a decision procedure for equality, the two expressions must therefore be equal simply on the basis of the equalities in the certificate.

The presence of operands changes the picture only slightly. When ExtAssert establishes an equality on the basis of equal operands, an "equaloperands" witness link is inserted. When ExtCertify encounters this witness link, the original operand pairs of the two expressions are placed on the ToCertify list (Figure 5.23, line 42). On some later iteration of the major loop in ExtCertify, these operand pairs will themselves be certified, and eventually all required certificates will have been emitted.

What about the Union-Find on witnesses? It can only cause trouble if a PathFind operation prevents the algorithm from emitting a witness label that would have otherwise been emitted. We claim that the labels on all witness edges internal to one of these equivalence classes have already been emitted. The only time a PathUnion is performed in ExtCertify is at line 46. Immediately prior to that line is an <u>if</u> statement. Taking either branch of this <u>if</u> statement causes the witness label to be effectively emitted from the edge on which the PathUnion is about to be performed. At this point it should be clear that PathFind operations can never cause ExtCertify to skip over unemitted witness labels.

5.10 Why Do the Certificate Equalities Form a Minimal Set?

The set of certificate equalities is minimal because removing any one of them will partition the equivalence class in such a way that the two expressions being certified will be in different equivalence classes. As in the section above, we will initially argue in the case of the simple Certify (without operands or witness Union-Find) and then examine the more complex case.

Suppose some equality in the certificate could be removed. This would correspond to removing a witness edge in the congruence closure graph. Since the witness edges form a tree, this would leave the two expressions at the ends of the certificate unconnected by witness edges. But the witness edges form a spanning tree on the congruence closure equivalence class, so that this would imply that the two expressions are in different equivalence classes. This contradicts the initial assumption that they were equal. We conclude that no equality in the certificate can be removed, and the certificate therefore forms a minimal set.

What happens when we include operands? It should be obvious that the case here is exactly that of the MultiCertify problem of section 5.3. That is, we can consider the problem of certifying operand equality as simply the problem of certifying a set of simple equalities.

The only problem we can encounter with the set of equalities is that of emitting duplicates. By exactly the argument we used above in the simple case, no equality that is deducible from the other equalities in a certificate (but is not equal to any of them) can ever appear in a certificate. Therefore, any superfluous equality in the certificate of the set of queried equalities must duplicate some other equality in the certificate.

The witness Union-Find guarantees that no duplicates will appear in the final output certificate. The argument for this is exactly that we gave for correctness at the end of section 5.9, with the added observation that the PathUnion operation is always performed immediately after the edge is emitted.

5.11 Can We Do Better Than Minimal?

The certificate produced by our algorithm is minimal, in the sense that removing any element makes it impossible to deduce the desired equality. However, it is not necessarily a minimum, or smallest possible, certificate. For example, if we ExtAssert a=b, a=c, and then b=c, the b=c equality will be deducible from the previous equalities, and hence will have no effect on the data structure. If we query b=c, our algorithm will produce the certificate {a=b, a=c}, even though the certificate {b=c} is obviously smaller. We might hope to find an algorithm that always produced a minimum certificate. Unfortunately, such an algorithm would allow us to solve an NP-complete problem.

In the problem of Steiner trees in graphs (problem [ND12] in [Garey and Johnson 79], restricted to the case of equal edge weights) we are given a graph G = (V,E) and a subset R of V. We are asked to find a subtree of G including all the vertices of R such that the number of edges in the subtree is no more than some constant B. The mapping to our problem is straightforward. The vertices (V) of the graph are expressions, and the edges (E) are asserted equalities. For a set of connected vertices R = {a, b, c, d, ...}, the question we ask is: "Does f(b, f(c, f(d, f(...))) = f(a, f(a, f(a, f(...)))". The minimum set of equalities solving this problem forms a tree of minimum size, thus providing a solution to the Steiner tree problem.

Occasionally the hard part of a problem can be factored out as an isolated, unusual case (see [Chan 77] for an example). We believe that in this problem no such factoring can be done, and the problem we are solving is inherently difficult.

5.12 Running Time of the Algorithms

On the question of running time, we wish to argue for two points. First, our changes to the congruence closure algorithm do not affect its basic running time. Second, the certification algorithm runs in time almost-linear in the number of equalities in the certificate plus the number of applications of congruence axioms required. We could explicitly list applications of the congruence axioms in the certificate, but they are so obvious that this seems undesirable.

In discussions of running time below we will take N to be the total number of characters in the input to ExtEq. The expected running time of congruence closure is taken to be O(N log N). (The algorithm uses hashing, so of course the worst case running time may be greater. For a complete discussion of the running time of the congruence closure algorithm, the reader is referred to [Johnson 81].)

5.13 Changes to the Running Time of Congruence Closure

Only two of our changes to the congruence closure algorithm involve loops or recursion. These are the only changes that can affect the running time by more than a constant factor.

The first such change is the addition of a constant number of representative lookups to ExtFind (Figure 5.21, lines 16-19) (note that the number of operands of a function is assumed to be bounded by some constant). Each of these lookups can cost at most O(log N), and ExtFind can be called at most O(N) times. Therefore the total added cost here will be O(N log N), which will affect the overall running time of congruence closure by only a constant factor.

The second change involving a loop is somewhat more subtle. In the ExtAssert algorithm (Figure 5.22), code has been added to the basic congruence closure algorithm to reverse the witness path from a node to its representative in the graph. This code is at lines 21-29. Since the length of the witness path from a node to its representative can reach the size of the equivalence class, this step in the algorithm can take time up to O(N). However, each time this reversal is done, it is done in the smaller of two equivalence classes that are being merged. The operation is only done when two equivalence classes are merged, and the cost is O("size of the smaller class"). Each time this operation is done, the size of the smaller class is at least doubled. Therefore, this operation can be done at most O(log N) times. Thus, the total cost of all such operations is O(N log N). Once again, we have affected the overall running time of the congruence closure algorithm by only a constant factor.

5.14 Running Time of Certification

We have been claiming that we can do the certify operation in time almost-linear in the number of equalities in the certificate. This would be exactly correct if applications of the congruence axioms were listed in the certificate. To be precise, we will require time proportional to the sum of the number of applications of congruence axioms and the sizes of the equalities in the certificate times Tarjan's α function [Tarjan

75] (a very slow-growing function related to the inverse of Ackermann's function) on this value. We will refer to the sum of the number of applications of the congruence axioms and the sizes of the certificate equalities as K.

We must consider each loop in the ExtCertify routine (Figure 5.23). The the number of iterations of the outermost loop (at line 7) is based on the total number of pairs put into ToCertify. The input expression pair to be certified is put into ToCertify at line 6. The only other time an expression pair is put into ToCertify is at line 42. With each pair of operands put into ToCertify at line 42 we can associate an application of the congruence axioms or a pair of operands in one of the certifying equalities. This outer loop will therefore be bounded by O(K). Moreover, the work done for each ToCertify pair is, except for PathFind operations, disjoint from the work done for any other pair.

The PathFind operations at lines 9, 17, 21, 32, and 45 will, over the O(K) iterations of the loops containing them take at most time $O(K\ \alpha(K))$. The loops at lines 13-23, 28-33, and 34-48 are each bounded by twice the length of the witness path between the two expressions, and thus by O(K). Moreover, since the different ToCertify pairs require disjoint effort, the total work done in these loops in all iterations through the outermost loop will be O(K). Thus, we see that the total cost of the algorithm will be bounded by $O(K\ \alpha(K))$, as desired.

5.15 Conclusion

We believe that the algorithm for determining logical dependency presented here is both practical and efficient. It is being incorporated into the AVID system, and we feel it will be a very useful proof development aid. The algorithm can potentially be included in any of several verification systems that make use of the congruence closure algorithm for deciding equality (e.g. the Stanford Pascal Verifier [Luckham et al 79]).

CHAPTER 6
System Design and Implementation

6.1 Overview

The implementation strategy for AVID called for the system to be easy to build, easy to modify, and, wherever possible, to make use of preexisting software tools and modules. We feel that this strategy was successfully carried out. As implemented, language constructs can be added or modified with little effort. A version of AVID to support any simple block structured language, e.g. PASCAL, could be constructed from the existing system in just a few man-months of effort.

The AVID system has been implemented in the C language [Kernighan and Ritchie 78] on a DEC VAX 11/780 computer running the Berkeley UNIX operating system. The choice of C was dictated mostly by the available operating system. However, there were several excellent software tools and modules already available in C, of which we made extensive use.

Interestingly, although the Cornell Program Synthesizer [Teitelbaum and Reps 80] is also written in C and is currently available under Berkeley VAX/UNIX, the systems share no common code. The Synthesizer was designed as an introductory programming system, with a number of execution and debugging aids. It was also brought up initially on a small microcomputer system. AVID, however, was designed as a program development system for sophisticated users. It does not have any facilities for program execution, only for verification. It was also designed to be modular and easy to maintain and extend. No particular attempt was made to make the system small or to keep it very efficient.

As a result of these different goals, we felt that it was far better to design AVID from the ground up, rather than attempt to modify the existing Synthesizer system.

In this chapter we describe the basic structure of the AVID implementation, in order to give the reader a good understanding of how the system is constructed and how difficult it might be to adapt it to solve other similar problems. The system breaks down naturally into five major components. These are: the template structure, the phrase grammar, the symbol table, the editing facilities, and the verification procedures.

The template structure is the basis for the internal representation of AVID programs. It consists of a collection of layouts and initial values for template data structures, together with some routines to access and modify these structures. These structures are dynamically allocated to represent the AVID program being created.

The phrase grammar is just a description of the parser for AVID phrases. The symbol table is also relatively straightforward and interacts with the parser in the usual fashion.

The editing facilities provide the user's access to the system. The available commands allow the user to modify the internal representation of the program in certain well defined ways. The routines that implement the editing commands invoke the template structure routines to perform the actual changes to the program representation.

The verification procedures are quite separate from the rest of the system; they process serial streams of verification code instructions and calls on the arithmetic verification package. In the current implementation, when the user invokes the verify command, the V-code for the entire proof is processed through the verifier. This is the only time the verification routines are used.

We have devoted one section of this chapter to each of the five components listed above. The final section of the chapter discusses problems with the current system and proposes some solutions and overall system improvements.

6.2 Template Structure

The template descriptions are central to the AVID system. They give most of the semantics and proof rules for the language constructs. The remaining semantics are given by the phrase grammar described in the next section.

In designing AVID, it became clear that the templates were both moderately complex and extremely important. Although they can be specified directly as C language routines and data structures, this makes them very hard to modify and even harder to understand. We therefore designed a special purpose high-level language in which to describe the templates. We will refer to this language as TDL (template description language). TDL compiles into a set of internal tables and C language routines that implement the base language constructs.

In designing TDL we had several conflicting concerns. We wanted the language to be at least somewhat comprehensible to individuals who were not intimately aware of the details of the AVID implementation. To this end, we made the structural form of the language significant. When indentation occurs in the TDL description of a template, that indentation also occurs in the template as it is eventually displayed on the screen. Similarly, the brackets surrounding a TDL element are in most cases the same brackets that appear surrounding it on the display.

We also required that TDL be reasonably rich. TDL allows the expression of the print form of all template elements, the verification code that must be generated to ensure correctness, the commands legal at any point in the program/proof being developed, and the start symbol for parsing any text phrase to be entered. In addition, several other checking and control functions (e.g. symbol table block entry) can be expressed in TDL.

While TDL is fairly rich, it is not complete. Any semantic information not directly expressed in TDL must be written in C. These are much less accessible to someone attempting to understand the system. We have tried to express the most important features of the templates directly in TDL, but we have not been totally successful. In particular, the semantics for dependent non-terminals has been written almost entirely in C, although we do not feel that this has resulted in a great loss of comprehensibility.

Our final goal in designing TDL was to make the language itself easy to modify and extend. In this we feel we have succeeded. The compiler for TDL is written as a Yacc parser [Johnson 78]. Yacc generates a LALR(1) parser from an extended BNF description of TDL in which semantic actions are written in C. Lexical analysis for TDL is done using the LEX scanner [Lesk 75].

On several occasions during the process of implementing templates for the various PL/CV2 constructs TDL came up short. In each case, it took only a few hours to make the appropriate changes to the TDL compiler. This augurs well for the adaptability of the TDL system to new languages.

In Figure 6.1 we give an example of the TDL form of an implication

```
IMPIN:

%impass:<implication-assertion:ASSCODE::%{
    return(((majoroperator(%$impass) == IMPLY)?
        CNULL : "Major operator must be IMPLY"))
%}:>" BY IMPIN."
%V{ RETURN vopen() %}
    "PROOF;"
        "ASSUME "%hyp:<*hypothesis:%{
            RETURN make_syntuple(t,imp_hyp(%$impass)) %}>";"
        %V{ RETURN vproclaim(%$hyp) %}
        {argument:ASSCODE:just:::ARGUMENT}
        %conc:<*conclusion:%{ RETURN make_syntuple(t,
            imp_conc(%$impsss)) %}>[O_JUST:djust]";"
        %V{ RETURN vquery(%$conc,
            "Verification error: cannot deduce conclusion") %}
    "QED;"
%V{ RETURN vclose() %}
%V{ RETURN vproclaim(%$impass) %}
```

Figure 6.1 TDL form of implication introduction template

introduction template. The example is taken directly from the TDL compiler input for the current version of AVID, except that line numbers have been added to aid in the discussion of the construct.

This example illustrates most of the major features of TDL. To make a few general observations about the language: C language code fragments are often directly included in the description. These fragments are enclosed in bracket pairs of the form **%{ ... %}** or **%V{ ... %}**. Nonterminals in the language can be labelled with a name. The form of a label is **%name:**. A reference to the nonterminal is then made by **%$name**.

In order to give the reader a good understanding of TDL, we provide a line by line analysis of the template of Figure 6.1 below.

Line 1: This line labels the entire template. It is used in the command section of the templste description to associate an input command with the template. It is also used within the AVID system code when the template must be referred to directly. For example, the C code that sets up the initial state of AVID uses the name OBJECT to refer to the initial object template.

Lines 3-6: These lines describes the implication to be introduced. It is the first simple nonterminal in the template. The string "%impass:" labels this nonterminal so that it can be referenced elsewhere in the template. The nonterminal itself is contained in angle brackets. Within these angle brackets are fields (separated by colons) giving (in order): the nonterminal name, the phrase grammar start symbol for any phrase that may be used to refine this nonterminal, the (null) name of a set of commands that may be used to refine this nonterminal, the function to be executed when this nonterminal is refined (it checks that the major operator is an IMPLY operator), and the (null) function to be executed when a refinement for this nonterminal is deleted.

Line 6: The string "BY IMPIN," is a syntactic element of the template. It will be displayed on the screen to aid the user in interpreting the template.

Line 7: This line specifies an instruction to the verification code interpreter. It tells it to open a proof block, within which the implication hypothesis can be assumed.

Line 8: This is another syntactic element.

Lines 9-10: These lines describe the hypothesis of the implication. This is the specification for a derived nonterminal. It is to be obtained by executing function "imp_hyp" on the implication assertion.

This function will be written separately in C.

Line 11: This line specifies a V-code instruction. The implication hypothesis is to be "proclaimed", that is it is declared to be equivalent to **true** within the current proof block.

Line 12: In the argument portion of the template, the user provides the logical argument showing that the conclusion can be derived from the hypothesis. The argument is an iterated nonterminal. It can be refined with an arbitrary number of templates or phrases. The iterated nonterminal specification is enclosed in curly brackets. Like the simple nonterminal specification, it consists of a series of colon-separated fields. These are: the name of the nonterminal, the parse type of any phrase used to refine it, the name of the set of template commands (containing, for example, **:andim**, **:arith**, etc.) that can be used to refine it (in this case any justification template may be used), the insert and delete functions (both null), and finally, the generic name of this iterated nonterminal, in this case just "ARGUMENT".

Lines 13-14: The conclusion of the implication is specified by a derived nonterminal at this point in the template. Its form is identical to that of the hypothesis.

Line 14: At the end of line 14, the string "[O_JUST:djust]" specifies an optional nonterminal. This allows for the conclusion derived nonterminal to be optionally justified by some other template. The optional nonterminal itself contains the name of the template to be inserted at this point when the cursor is at the option point (O_JUST in the example), and the commands that can be legally used to refine the optional. These commands are just those that are legal to justify derived nonterminals.

Lines 15-16: This line causes the generation of a V-code instruction that queries the derivability of the conclusion to the implication. The "Verification error: ..." string will be displayed to the user if this query operation fails.

Line 17: This line just specifies another syntactic display element.

Line 18: This is a V-code instruction that closes the proof block that was opened at line (07). This returns the assertion table to its state at the time of the open. In this case, the implication hypothesis would no longer be equivalent to **true** (unless it had been proved elsewhere, of course).

Line 19: The final V-code instruction in the template proclaims the truth of the implication being introduced.

In Figure 6.2 we show the template of Figure 6.1 as it would be displayed on the screen immediately after being invoked. Observe that the indentation and new line formatting for the displayed version of the template is taken directly from the indentation and formatting of the TDL version. Within C code fragments in TDL (e.g. lines 4-5 of Figure 6.1), formatting is ignored.

The internal representation of a template in AVID is a linked list of nodes, known as *tuples*. Each tuple is associated with a single nonterminal, syntactic element, or V-code instruction. When a nonterminal is refined, the refinement is attached to the linked list as the son of the nonterminal. Thus, a complete program or proof will form a tree, with each tuple list (template) serving as a node in the tree. To generate the print form of the program, or to generate the V-code instructions needed to verify its correctness, the program does a tree traver-

```
<implication-assertion> BY IMPIN,
   PROOF;
      ASSUME <*implication hypothesis>;
      {argument}
      <*implication conclusion>;
   QED;
```

Figure 6.2 Display form of implication introduction

sal, outputting the print strings or V-code instructions as it goes. An example of the tree traversal process is given in the sample development session in the appendix.

6.3 The Phrase Grammar

The phrases of AVID are described using the Yacc parser generator and LEX lexical analyzer. Since the grammar is expressed as a high-level BNF description, rather than being built into an algorithm, the definition of the phrases is easy to understand and easy to modify.

The semantic actions of the phrase grammar perform three major functions. They serve to look up and insert symbols in the symbol table, perform type checking on expressions, and create the abstract syntax tree (*ast*) representations of the phrase being parsed.

A node in the *ast* of a phrase has four significant attributes. The first defines it as an operator node with some number of arguments or a leaf node with some kind of value. Its second attribute is its semantic meaning, for example, the operator means *plus* or the leaf represents a label reference. Its third attribute is its value type, which is chosen from the set *fixed*, *boolean* and *proposition*. This is used in type checking. Finally, it has either a set of arguments, if it is an operator, or a specific value, if it is a leaf. For operators, the arguments are themselves asts. For leaves, the values may be symbol table references, boolean constants, fixed constants, etc.

In Figure 6.3 we give the specification for the program-statement nonterminal of the phrase grammar. This specification has been taken directly from the Yacc input file for the current AVID system. To make it easier to describe and follow the example, we have numbered the lines, and the BNF grammar portion of the specification has been put in bold font. The semantic actions are displayed in normal font.

In the BNF portion of the example, **stmt** is the nonterminal being defined, **comment**, **assign**, and **assertion** are nonterminals defined elsewhere in the grammar, **kwLEAVE** is the lexical token for the LEAVE keyword, and **tID** is the lexical token for an identifier. The **':'** simply represents the colon lexical token.

As an example of the process of creating the ast, consider lines 6-7. The function "operator" creates an operator node, in this case of a statement, with no value type and one argument, which is a leaf. The function "leaf" creates a leaf in the parse tree. In this case it is an instance of a LEAVE statement, where the label on the loop to exit is the **tID** (the $2 refers to the second element in the BNF part of the line).

At lines 8-11, the type of the identifier given as an argument to the LEAVE is checked. If it is not a label, then a type error is generated. Lines 14-21 demonstrate a symbol table declaration. The function "forcenew" guarantees that no conflicting declaration exists. Lines 20-21 then establish the kind and the value type for the identifier. For more on the symbol table, see the next section.

We are using the Yacc/LEX package in a completely standard manner. For further information and more examples on using this package, see

```
stmt :
  comment        |
  assign         |
  assertion      { $$ = operator(STMT,ASSERTION,1,$1); }
                 |
  kwLEAVE  tID   { $$ = operator(STMT,NOTYPE,1,
                           leaf(LEAVE,NOTYPE,$2));
                   /* Must check type of label here */
                      if($2 -> enttype != LABELTYPE) {
                          typerror("LEAVE with non-label");
                      }
                 }
                 |
  tID ':' assign  { addlist($3, leaf(STMTLABEL,NOTYPE,$1));
                  /* Must check that tID has not been declared */
                      sdptr = forcenew($1,PROC_LOCAL);
                      if(sdptr == SNULL){
                          yyerror("statement label redeclaration");
                      }
                  sdptr -> enttype = LABELTYPE;
                  sdptr -> adtype = NOTYPE;
                  $$ = $3;
                 }
                 ;
```

Figure 6.3 An example of a phrase grammar entry

[Johnson 78] and [Lesk 75].

We trust that this short example supports our contention that the representation of the phrase grammar is relatively easy to understand and to modify. While we have glossed over some of the details in this presentation, we do not feel we have hidden anything that would greatly increase the complexity of the system.

6.4 The Symbol Table

The design of a symbol table for AVID posed some new and interesting problems. To our knowledge, this is the first system to support interactive incremental change with a full block structured symbol table. Neither the Cornell Program Synthesizer, nor the COPE system have such a block structured symbol table, since the PL/CS language does not support either begin blocks or internal procedures. The PL/CV2 logic, however, requires full block structure. Both the ALL introduction template and the SOME elimination template open new symbol table blocks.

AVID poses another symbol table problem not encountered in CPS or COPE. A single phrase in AVID can contain nested, block structured declarations. For example, the phrase:

ALL (x,y) FIXED. (x>y | SOME z FIXED. x+z>y)

contains two levels of nested declarations, a declaration of x and y in the outer block, and a declaration of z in the inner block. These declarations are not subject to incremental change, since the entire phrase will be reparsed if any change is made to it.

The AVID symbol table is thus composed of two combined symbol table representation methods. During the parsing of a phrase of the type shown above, a standard stack structured symbol table of the type described in [Gries 71] is used. By the end of the phrase, all the stacked blocks will have been discarded. The other symbol table supports arbitrary incremental change to declarations and references of variables, functions and labels. This type of change was discussed in Chapter 4.

The main AVID symbol table consists of a set of hash buckets containing linked lists of symbols. The symbols are identified by their name and a unique block identifier for the block that contains them. At this point in the design, we made one crucial observation. All symbol table change and lookup takes place at the current cursor location. That is, in order to change or enter a symbol, the user of the system must first move the screen cursor to the location of the change or entry. Thus, our solution to symbol table lookup is to keep a stack of unique block identifiers that is always correct for the current cursor position. To look up a symbol "x", the system checks the hash table for "x" and each block identifier going back through the stack until the symbol is found. As the cursor is moved through the program, the current block stack is constantly updated (at little cost).

It should be obvious that, with this approach, the symbol table can be modified arbitrarily, and each lookup will still return the correct instance of any symbol. As discussed in Chapter 4, there may be more general and elegant solutions to this problem, but they will require significant new research. At the time the AVID symbol table was designed, exploration into the problem of incremental change in interactive environments was just beginning. of modelling syntax-directed editors with attribute grammars.

6.5 Displaying and Editing Proof Text

The display and modification of proof text in AVID makes use of two software modules developed for the Berkeley UNIX system. They are *curses*, a package supporting text display on arbitrary terminals [Arnold 80], and *vi*, a screen oriented visual text editor [Joy 80]. In addition, a number of routines for functions specific to the AVID system were written.

The *vi* editor is a fairly standard screen-oriented editor. However, both it and *curses* support a large number of different terminal types (at last count, over 150) together with most of their intelligent features, such as insert line, insert character, full cursor addressing, etc. This makes the AVID system very portable among UNIX sites, since it is not tied to a specific hardware configuration.

The format of commands in the *vi* screen-oriented editor is a single command character followed by a target motion. For example, the command character **d** (delete) followed by the motion character **w** (the next word) has the effect of deleting the next word. A command is stuttered to address the current line, thus **dd** would delete the current line. Commands can be used with counts to repeat their effect, so that **3dd** would delete the current line and the two lines following it. The *vi* motions include word, character, line, end of line, to balancing parenthesis, to a given character, and many others. Thus *vi* allows the modification of many different textual objects through a uniform set of commands.

The *vi* editor was particularly easy to incorporate as the user interface to the AVID system. It had originally been designed as a screen editor front-end to the standard Bell Labs UNIX editor *ed*. As a result, the interface between the screen functions and the underlying line oriented editor functions was clean. Some other editor might be harder to adapt in this fashion.

From the user's point of view, AVID's basic change to *vi* was a reinterpretation of the line-oriented commands. In AVID the *vi* line commands affect either simple subtrees, in the case of single "line" commands, or a set of sibling subtrees (adjacent iterated nonterminals) when a command targeted to several "lines" is used. In terms of the implementation, this meant that the code to perform line oriented functions had to be completely rewritten.

In a standard text editor, any string of characters can be modified directly. This is not the case in AVID. Text used to refine phrases, and text parsed by the phrase grammar parser, can be directly edited. The remaining character strings are syntactic display elements. They are implied by the underlying abstract syntax tree structure, and can change only when that structure is modified. Thus, although the text in the display appears uniform, only parts of it can be modified.

It might be possible to allow a modification to a syntactic element to cause the modification of the program structure. For example, the system could allow the user to edit the phrase "DO WHILE" directly into the phrase "DO UNTIL". This would immediately pose some impressively difficult problems. This change implies a change to the structure and interpretation of the logical assertions used to prove correctness and termination for the construct. Developing a coherent strategy to modify these assertions in some meaningful fashion in response to an arbitrary change in the structure would be a difficult task.

The COPE system [Archer 80] does allow this kind of modification. The program text can be changed arbitrarily by the user. To parse the resulting text, the system uses a two level grammar, which takes advantage of the fiduciary nature of keywords in the language (i.e. these are symbols that can be "trusted" to correctly reflect the language structure). The language accepted by COPE is, however, much simpler than that accepted by AVID. Further, the parsing techniques put restrictions on the language structure that are not completely understood. Finally, such changes are not always well defined. A change may have a completely different effect from that intended. The design of COPE assumes that the user will have an "undo"/"redo" facility available, and that experimentation may be necessary to achieve the desired effect. The size and complexity of AVID templates would seem to make the COPE approach an inappropriate one.

The Cornell Program Synthesizer project is experimenting with a system that allows the transformation of one template to another through special commands. All the transformations must be predefined, and where there is some question about the interpretation of the transformation, the system currently does not allow the transformation to occur. Given AVID's very complex templates, it is not clear that this approach would be successful.

The question of cursor movement is closely related to that of editing. It seemed desirable to us to allow the cursor to move arbitrarily over the text. However, the cursor is only meaningful within phrases or at discrete points associated with the start of program structures. The solution was to treat the cursor as though it were at the closest preceding legitimate location. This allows the user to do, for example, a pattern-matching search over the program text as it appears on the display, and to have the cursor move to some reasonable position. This facility is unavailable in the Cornell Program Synthesizer, which allows the cursor to move only in tree-structured quantum jumps.

We should make a brief comment on the display of error messages in the system. In AVID, a single user command can result in a number of meaningful error messages associated with various program locations. For example, a verify operation can attach a verification status to a number of logical assertions. We decided to actually attach the error messages to the program statements. As the user moves the cursor through the program, the system displays whatever message is associated with the phrase or structure at the current cursor position.

The current AVID system includes one choice, due partly to expediency, that may be questionable. Phrase editing is done out of line in a separate edit buffer at the bottom of the screen. As soon as the user issues any command to move into or to modify a phrase, the phrase is put into the edit buffer, and the operations are performed there. When the user is done with the phrase and moves away from it, the phrase is parsed and reinserted into the main display. This allows the main screen to truncate lines at the right edge of the screen, while the edit buffer wraps around to display the entire phrase at once. If all lines wrapped on the screen, the number of lines available for display would be decreased.

The COPE system uses the same out-of-line edit buffer approach. We need to gain user experience with both AVID and COPE and contrast the results with user feeling about CPS, which uses inline phrase editing, before we will be able to make an informed decision on which system is better. It would be fairly easy to modify AVID to perform inline phrase editing.

AVID provides facilities for clipping and inserting pieces of proof, and for saving these on the file system. The vi yank (**y**), delete (**d**), and put (**p**) commands allow saving and inserting program text using named buffers. These commands work much as they do in vi. In addition, the **:save** and **:read** commands save and restore programs from the file system. All of these facilities make use of the same representation for AVID programs. When one of these representations is inserted into a proof, the effect is exactly as though the user had entered the series of commands and phrases that were used to create the saved file at the new insert point. The representation is simple and fairly intelligible. In Figure 6.4 we show a small program fragment and its saved representation. The meaning of a line in the representation is determined by the first character in the line. The meaning of the characters shown in the figure is: ; - a comment line, _ - the cursor position, : - this is a template command, e.g. **:allim** to create an all introduction template, <space> - this line contains a phrase to be entered at this point, + - move down into the template, and > - move right in the template. The saved representation also has provisions for keeping the verification status and error status of a saved routine.

6.6 Verification

Verification in AVID is performed using the PL/CV2 V-code interpreter. An excellent description of this interpreter can be found in [Johnson 81]. The PL/I code for the version of the interpreter currently running on the IBM 370 was hand translated into C. The PL/CV2 arithmetic package [Chan 77] was also brought over from the IBM 370.

Since the PL/I implementation of the V-code interpreter on the IBM 370 was not impressively fast, we were worried that the system would be unacceptably slow for interactive use on the VAX. We were pleasantly surprised. We have not done extensive testing, but the system appears to the user to run about as fast on the VAX as it did on the 370. In actual use, its performance seems quite acceptable.

We are currently in the process of adding the logical dependency algorithm of Chapter 5 to the V-code interpreter. We do not anticipate any performance loss. As we argued in section 5.12, the new algorithm is

```
ALL x FIXED . ((x >= 3 => x = 3) | (x <= 3 => x = 3)) BY ALLIN.
   PROOF;
      ARB x FIXED
      tri : ((x >= 3) | (x <= 3)) BY ARITH, true;
      ((x >= 3 => x = 3) | (x <= 3 => x = 3));
   QED;
```

```
;
_
:allin
+
>
 ALL x FIXED . ((x >= 3 => x = 3) | (x <= 3 => x = 3))
>
>
:arith
+
>
 tri : ((x >= 3) | (x <= 3))
>
 true
```

Figure 6.4 A program fragment and its saved representation

quite efficient.

6.7 Doing it Over

The list of things that we would do differently if the system were being completely redesigned and reimplemented is not as large as it might be. A version of AVID that was designed and partially implemented on a PDP 11/60 running UNIX V7 was almost completely thrown away. Only the phrase parser and symbol table routines survived the transition to Berkeley VAX/UNIX essentially unchanged.

In the PDP 11/60 implementation, the AVID user interface was very much like that of the Cornell Program Synthesizer. It did not, however, seem appropriate for the more sophisticated user community we expected for AVID, and so when a larger machine became available, we were eager to find a better user interface. The template description facility also became much more powerful in the transition from the 11/60 to the VAX. Many functions that had been pushed off into special C language routines were made directly visible and modifiable in the VAX implementation. Finally, the internal representation of all objects was simplified and made consistent through the use of a single basic construct, the tuple. Although the program representation became slightly less compact, the savings in the size and complexity of the code to manipulate the internal representation more than made up for it.

The use of a more consistent and modular implementation in the VAX version had other beneficial effects. The system as a whole seems to be both easy to modify and robust to the effects of change. In our experience with the system to date, it has remained very stable even as we have been adding new facilities and modifying old ones. As the size of problems being handled by the system is scaled up, the simple and modular design should also make it easy to provide any improvements in efficiency that are required.

The availability of the Berkeley system allowed us to make use of the vi editor front-end and the curses display package. These two software packages gave us the powerful editing features and the terminal independence for which we had been looking. They also made it possible to implement the system quite quickly.

One major possible change in the system design and implementation has already been discussed. In Chapter 4, we looked at some of the problems and potential benefits in using the attribute grammar approach to modelling incremental change. Using attribute grammars would allow the system to handle in a uniform, high-level fashion a number of facilities that have been implemented at a much lower level in the current system.

In particular, the maintenance of the symbol table, the print form of the program, the generation of verification code, and the updating of derived nonterminals could all be more cleanly implemented with an attribute grammar approach.

As with all general approaches, however, attribute grammars have their costs. Without losing all the advantages of the high-level approach, you cannot obtain the efficiency and specialization with an attribute grammar specification that you can with a C implementation like the current one. The AVID symbol table is a good example of the trade-offs involved. Since we had complete freedom in designing the symbol table specifically for the AVID environment, the design could be simple, efficient, and unique. Using attribute grammars, however, the best currently available symbol table design is much less efficient, although it can be specified and changed at a higher level. In an editing environment, however, efficiency losses can usually be completely masked by the interactive user response time, weighting the choice in favor of the more general approach.

In a sense, the current template description language (TDL) is a specialization of the general attribute grammar approach to a specific problem. Functional dependencies in TDL are more restricted than they would be using the general approach (although escapes to the C language allow full generality), but in TDL specific, complex objects (e.g. iterated nonterminals) are very simple to specify.

There are other problems with going completely to the attribute grammar approach. We have already discussed the problem of large attributes in Chapter 4. There is also some question as to how easy it would be to implement a powerful screen-oriented editor user interface in such an environment. The editing environment in current experimental attribute-grammar-based systems is closely tied to the tree structure of the developing program.

It seems likely that the best approach to applying attribute grammars would be a hybrid one. Where there are clear advantages in simplicity and clean design, attribute grammars could be used. In other cases, a straightforward, lower-level approach could be used. We could make some of this flexibility directly available to the system designer using AVID as a base by integrating a general attribute grammar mechanism into the current version of TDL. Although further research needs to be done, we would almost certainly try to make use of attribute grammars if the system were to be rebuilt.

As a final note, although we have stressed the value of integrating a sophisticated screen-oriented editor such as vi into the system, it seems likely that vi itself is not the final answer. It was readily available in a form that was very easy for us to use, and it was certainly more powerful than any editor currently employed in a Synthesizer-like system. An improved "undo" and "redo" feature is one of the first things that should be added to the editor. This could be done quite easily using an approach similar to that of COPE, but it would be impressively inefficient. Designing an efficient implementation would be fairly difficult.

CHAPTER 7
Conclusions and Future Directions

7.1 The Current Status of AVID

The implementation of AVID was begun in the spring of 1980 on a DEC PDP-11/60 running Bell Labs V7 UNIX. In January 1981 the project was transferred to a newly acquired DEC VAX 11/780 running Berkeley UNIX. As of late summer 1981, a version of the system that allows the development and verification of predicate calculus proofs is available. In the immediate future, we plan to extend the system to allow the definition and invocation of recursive functions with conditional statements. This would allow the development and verification of some fairly interesting number theory proofs.

The implementation of the full PL/CV2 version of AVID, complete with the PL/CS programming language constructs, should be straightforward. Only the template descriptions of the constructs involved and their C language support routines still need to be written. We therefore expect that the full AVID system, as described in this thesis, should be available by January 1982.

The current version of the system, which supports the development and verification of the predicate calculus portion of PL/CV2, consists of approximately 20,000 lines of source code. When running on the VAX, the system requires 230K bytes of memory.

We have not done extensive timing tests on the system, but we do have some preliminary results and impressions. Simple movement and editing commands take place in real time for the AVID user on a lightly loaded VAX/UNIX system. A sample 49 line AVID proof takes approximately

1.1 seconds to read into the system from disk storage. The most expensive operation is verification. Verification of the same 49 line proof takes about 3 seconds actual user time. The verification time is comparable to that of the IBM/370 implementation of the PL/CV2 verifier.

In general, we are pleased with the progress of the implementation. The interactive system seems pleasant to use, although to date only small examples have been run. There are clear strategies for improving the efficiency of the system if it proves unable to deal reasonably with larger problems.

7.2 Where is AVID going?

Several extensions to AVID are planned for the near future. One of the most interesting is a proposed proof database system. This would allow AVID to store, in some accessible fashion, proofs and programs previously verified within the system.

In developing a proof, a mathematician will constantly appeal to well-known lemmas and obvious deductions. In AVID, except where these deductions are built into the automatic facilities of PL/CV2, they must be completely specified by the programmer. We would like to have some repository of common lemmas and deduction sequences that could allow the programmer to avoid laboriously reproving well-known results. Because the programmer may need a result that is basically similar to one already know, but that differs from it in some detail, it should be possible to search the proof database for lemmas that are "close" to the one needed.

It should be possible to develop such a database using current information retrieval technology. However, the problem of defining "closeness" in a useful and efficiently realizable manner seems to be both difficult and intriguing. Since this would impact on the representation as well as the retrieval, it seems central to providing a proof database facility. However, even a system that used a simplistic matching algorithm would greatly aid proof development.

Using the AVID system as a base, a research group at Cornell plans to begin investigating this problem in the Fall of 1981. When this facility is available, it will eliminate some of the tedious and unnecessary effort that is currently required for AVID proofs. It would no longer be necessary to "reinvent the wheel" with every small result.

Another potential use of the AVID system is as an implementation base for the new PL/CV3 language [Constable and Zlatin 81]. PL/CV3 provides a constructive programming logic with an extremely rich set of datatypes. The lack of sufficient datatypes and good methods for arguing about them is one of the main problems with PL/CV2.

To implement PL/CV3 under AVID, the language must first be analyzed in terms of templates and phrases. Next, the phrase grammar and template description for PL/CV3 must be written. This will involve writing a type checker for objects in the language, which in the case of PL/CV3 is a very complex problem. Finally, special functions for such problems as alias checking must be written.

This project has a much longer timescale. It will probably interact closely with the Program Refinement Logics system being developed at Cornell [Bates 79], and the two may even coalesce. We hope that work on the PL/CV3 implementation can begin some time in 1982.

The area of user interaction also provides some rich areas for future research with AVID. Currently, the system provides simple messages in response to errors in verification. The user cannot enter a dialogue with the system to obtain more information about the error.

[O'Donnell 81] has suggested using the positional information provided by labels to allow a "dialogue" on the exact cause of the error. If the system has an explicit reference to what the user believes is the hypothesis of some deduction, it can determine exactly where the problem lies. Without this reference mechanism, the system can say no more than that the hypothesis is not automatically deducible at this point.

This is only one of a number of possible approaches to improving the error handling of AVID. An English language dialogue to establish the nature of errors and the effect of changes has been successfully implemented [Moriconi 77]. A similar system could be implemented in AVID.

AVID provides a research base for a broad range of possible explorations into the problems of interactive program and proof development. Its design makes such extensions particularly easy to implement. We hope that other researchers will be able to take advantage of the AVID system in pursuing investigations in this area.

7.3 Summing Up

We feel that the AVID system has made a significant contribution to the area of interactive development. It is the first system to apply the new syntax-directed editor approach to the simultaneous development of programs and their correctness proofs. Further, we have developed and implemented an algorithm in AVID that makes possible the efficient interactive display of logical dependency in proofs. Finally, the AVID system provides an excellent base for future experimentation with interactive proof development systems. Not only can the system itself be easily extended, but completely new languages can be implemented using the base system (as we plan to do with PL/CV3 [Constable and Zlatin 81]). The insights provided by experience with the AVID system should also serve to guide the development of related proof-based development systems (e.g. PRL [Bates 79]).

Although the current version of AVID is probably not itself a practical tool for the day-to-day development of correct programs, with a more powerful logic and enhanced user facilities a practical system would quickly become feasible. We hope that the experience gained with the AVID system will bring us closer to the goal of rapid development of practical, correct programs.

APPENDIX
A Development Example

In this appendix we present a short sample development session. The development shown was actually performed on the VAX implementation of AVID. The figures in this appendix were created by editing files produced by the AVID **:print** command.

A few comments on the general format of the figures. They basically show the screen as it would appear to a user of the AVID system. Messages and commands appear at the top of the screen. The edit buffer for phrases is at the bottom of the screen. When the edit buffer is shown in a figure, it is separated from the main screen display by a line containing three dots (...). The bold box (□) is used throughout these examples to indicate the current position of the cursor on the screen.

In the sample session, we are developing a proof of the PL/CV2 assertion

    ALL (p, q, r) PROP . ((p => q) & (q => r) => (p => r))

(for a description of PL/CV2 see [Constable et al 81]). We will describe the development process step by step. We will also give the reader some brief descriptions of the internal structures and actions of the AVID system.

When the AVID system begins execution, the single nonterminal "<object>" is displayed. To begin a proof, the user must enter the command **:proof** to create the proof template (which just consists of the iterated nonterminal "{argument}"). In Figure A.1 we show the screen as it would appear after the user had entered the **:proof** command. In this

:allin▯

{argument}

Figure A.1 Just prior to selecting an all introduction template

figure, the user has also entered the text of the **:allim** command, used to select an all introduction template, but has not yet typed the return key, which will execute the command.

Figure A.2 shows the user display immediately after execution of the **:allim** command. The cursor is positioned at the start of the all introduction template. To enter the all assertion itself, the user types the return key to move the cursor to the "<all-assertion>" nonterminal and types **i** (insert) to begin the insert. This command moves the cursor to the edit buffer area of the screen, and allows the user to type in the all assertion in the normal fashion.

In Figure A.3 we show the screen as it would appear after the user has typed the all assertion, but before he indicates it should be inserted into the template. In order to demonstrate AVID's response to errors, there is an error in the typed input (the variable "z" has not been declared and should actually be a "q").

To enter the all assertion into the template, the user types the return key (this is one of several possible actions to indicate that the entry of this phrase has been completed). Figure A.4 shows the screen immediately after this action. An error message has been displayed at the head of the program, and the line in error has been flagged at the left margin with "##".

To correct the error, the user enters the <u>vi</u> command string **fzrr**. The <u>vi</u> command **f** searches in the phrase for the first occurrence of the next character typed (**z**) (also moving the phrase back into the edit

```
□<all-assertion> BY ALLIN,
   PROOF;
      <*arbitrary statement>
      {argument}
      <*all conclusion>;
   QED;
```

Figure A.2 An empty all introduction template

```
<all-assertion> BY ALLIN,
   PROOF;
      <*arbitrary statement>
      {argument}
      <*all conclusion>;
   QED;

...

all (p,q,r) prop.((p=>q)&(q=>r)=>(p=>z))□
```

Figure A.3 Completing entry of an incorrect all assertion

```
Undeclared variable in expression

##all (p,q,r) prop.((p=>q)&(q=>r)=>(p=>z)) BY ALLIN,
     PROOF;
        <*arbitrary statement>
        {argument}
        <*all conclusion>;
     QED;
```

Figure A.4 Error message resulting from incorrect all assertion

buffer). The r command replaces the character under the cursor with the next typed character (r). Figure A.5 shows the screen immediately after this change. At this point, the user again types the return key to indicate that the line is complete and that the cursor should be moved to the next legal position. The display resulting from this action is shown in Figure A.6.

Let us now consider the internal representation of the proof so far developed. Figure A.7 gives a block diagram of the internal tree structure of the proof. There are several things to observe in this diagram.

```
Undeclared variable in expression

#all (p,q,r) prop.((p=>q)&(q=>r)=>(p=>z)) BY ALLIN,
     PROOF;
        <*arbitrary statement>
        {argument}
        <*all conclusion>;
     QED;


all (p,q,r) prop.((p=>q)&(q=>r)=>(p=>█))
```

Figure A.5 Correcting the error

```
ALL (p, q, r) PROP . ((p => q) & (q => r) => (p => r)) BY ALLIN,
   PROOF;
     █ARB (p, q, r) PROP
      {argument}
      ((p => q) & (q => r) => (p => r));
   QED;
```

Figure A.6 All introduction with all assertion correctly entered

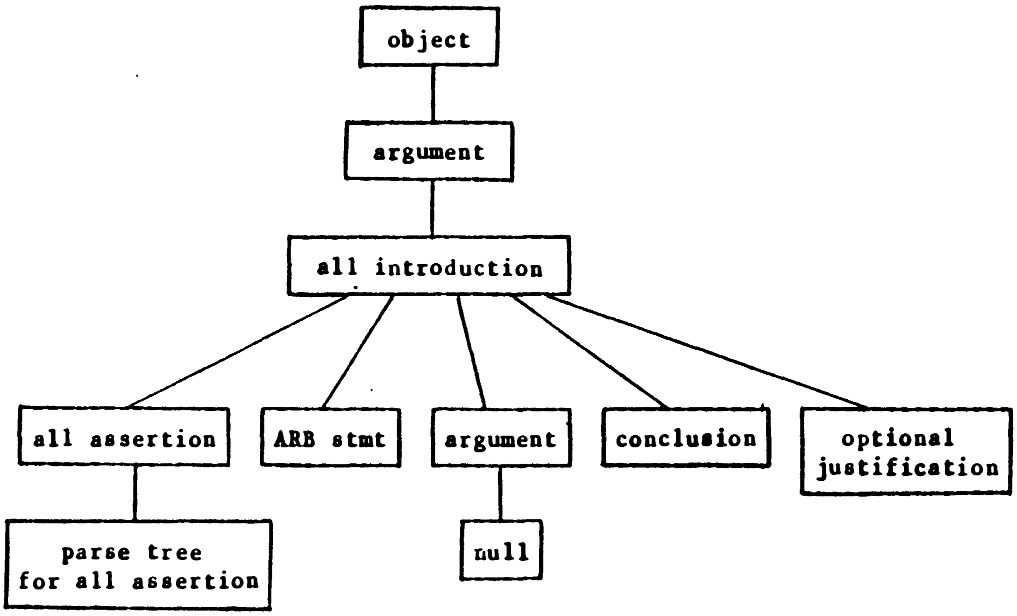


Figure A.7 Block diagram of internal representation

Note that each command (e.g. **zallim**) that produced a template has attached it as the descendant of the node at which the cursor was located. Each such inserted template may itself contain a number of elements. These occur as siblings under the parent template node.

The boxes for "ARB stmt" and "conclusion" represent derived nonterminals. Both of these are functionally dependent on the value of the "all assertion" box. These two nodes illustrate inter-node dependencies that are outside of the strict tree structure of the basic program representation.

Note that the "optional justification" does not appear anywhere in the template shown in Figure A.6. This nonterminal element is not displayed until the cursor has been moved to its location in the program, as in Figure A.8. This suppression of optional nonterminals helps reduce the amount of unnecessary information displayed on the screen.

Returning to the development, to reach the state shown in Figure A.8, the user typed the return key three times. This had the effect of moving the cursor through the program, suppressing the "{argument}" nonterminal (since no refinement was entered), and causing the display of the "<justification>" optional nonterminal.

Figure A.9 shows the display immediately after executing an **zimpim** command to select an implication introduction template to justify the conclusion of the all introduction. Observe that the hypothesis and conclusion of the implication have been automatically broken out and displayed.

At this point the user might wish to try to verify the proof. In Figure A.10 we show the effect of a verify command. The system is not able to verify the proof, and the error message and "??" flag serve to

```
ALL (p, q, r) PROP . ((p => q) & (q => r) => (p => r)) BY ALLIN,
   PROOF;
      ARB (p, q, r) PROP
      ((p => q) & (q => r) => (p => r))□<justification>;
   QED;
```

Figure A.8 Preparing to enter justification for all conclusion

```
ALL (p, q, r) PROP . ((p => q) & (q => r) => (p => r)) BY ALLIN,
   PROOF;
      ARB (p, q, r) PROP
      ((p => q) & (q => r) => (p => r))□ BY IMPIN,
         PROOF;
            ASSUME (p => q) & (q => r);
            {argument}
            (p => r);
         QED;;
   QED;
```

Figure A.9 After justifying all conclusion

```
:verify
Verification error: Cannot deduce implication conclusion

  ALL (p, q, r) PROP . ((p => q) & (q => r) => (p => r)) BY ALLIN,
     PROOF;
        ARB (p, q, r) PROP;
        ((p => q) & (q => r) => (p => r))  BY IMPIN,
           PROOF;
              ASSUME (p => q) & (q => r);
??           □(p => r);
           QED;
     QED;
```

Figure A.10 Attempting to verify proof

identify the problem. In this case, the automatic deduction facilities of PL/CV2 are not sufficiently powerful to allow it to directly deduce "p => r" from "(p => q) & (q => r)". The user must provide a more detailed proof.

Let us consider for a moment the internal steps involved in executing a verify command. First, the system traverses the entire program tree, generating V-code instructions to pass to the PL/CV2 verifier (see [Johnson 81] for a discussion of V-code). This traversal takes place top

```
:verify
Verification complete: no errors

  ALL (p, q, r) PROP . ((p => q) & (q => r) => (p => r)) BY ALLIN,
     PROOF;
        ARB (p, q, r) PROP;
        ((p => q) & (q => r) => (p => r))  BY IMPIN,
           PROOF;
              ASSUME (p => q) & (q => r);
              (p => r)□ BY IMPIN,
                 PROOF;
                    ASSUME p;
                    r;
                 QED;
           QED;
     QED;
```

Figure A.11 Successful verification of completed proof

down and left to right across the leaves of the program tree. During this traversal, special nodes in the tree representation of the program cause V-code to be generated (see section 6.2 for more information on how V-code is specified in the templates). Once all the V-code has been generated, it is passed to the V-code interpreter, which actually applies the deduction rules to determine verifiability. Finally, error messages are inserted back into the program to mark unverifiable assertions.

In our example development, the user now justifies the conclusion of the first implication with a second **:impim** command. After this, the user again enters a **:verify** command. This time, the automatic facilities of PL/CV2 are sufficient. From "p" and "(p => q) & (q => r)", the system can deduce "r". The final state of the screen, with the proof and verification status displayed, is shown in Figure A.11.

We hope that this short example has given the reader some understanding of the process of developing and verifying AVID proofs. We have demonstrated only a very few of the many AVID commands and facilities. For a complete discussion of these, see [Krafft 81].

The version of AVID that is currently running here at Cornell is a preliminary test version with a number of unimplemented facilities, although it is sufficiently powerful to develop and verify the example of this appendix as well as a number of more complex examples. Development work on the AVID system is continuing, and we hope to have a complete and robust version of the system available by January 1982.

REFERENCES

[Aho, Hopcroft and Ullman 74]
Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms, pages 124-139, Addison-Wesley, 1974.

[Archer 81] James E. Archer. The Design and Implementation of a Cooperative Program Development Environment. Ph.D. Thesis, Department of Computer Science, Cornell University, August 1981.

[Arnold 80] Kenneth C.R.C. Arnold. Screen Updating and Cursor Movement Optimization: A Library Package. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, September 1980.

[Bates 79] Joseph L. Bates. A Logic for Correct Program Development. Ph.D. Thesis, Department of Computer Science, Cornell University, August 1979.

[Bishop and Conway 77]
User's Guide to Release 2, PL/CS. Computer Science Technical Report 77-315, Cornell University, 1977.

[Boyer and Moore 79]
R.S. Boyer and J.S. Moore. A Computational Logic. Academic Press, New York, 1979.

[Buines-Rozas 79]
Juan Batista Buines-Rozas. GOAL: A Goal Oriented Command Language For Interactive Proof Construction. Computer Science Department Report No. STAN-CS-79-743, Stanford University, June 1979.

[Cartwright 76]
Cartwright, R.S. User-defined Data Types as an Aid to Verifying LISP Programs. Proceedings of the Third Intl. Colloq. on Automata, Languages and Programming, Edinburgh Press, Edinburgh, Scotland, 1976.

[Chan 77] Tat-Hung Chan. An Algorithm for Checking PL/CV Arithmetic Inferences. Computer Science Department Technical Report 77-326, Cornell University, October 1977.

[Constable and Johnson 79]
Robert L. Constable and Scott D. Johnson. A PL/CV Precis. Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 7-20, January 1979.

[Constable and O'Donnell 78]
Robert L. Constable and Michael J. O'Donnell. A Programming Logic. Winthrop, 1978.

[Constable and Zlatin 81]
Robert L. Constable and Daniel Zlatin. Report on the Type Theory (V3) of the Programming Logic PL/CV3. Computer Science Department Technical Report 81-454, Cornell University, June 1981.

[Constable et al 81]
Robert L. Constable, Scott D. Johnson and Carl D. Eichenlaub. Introduction to the PL/CV2 Programming Logic. Computer Science Department, Cornell University, April 1981.

[Conway and Gries 79]
Richard W. Conway and David Gries. An Introduction to Programming: A Structured Approach using PL/I and PL/C. Winthrop Publishers, Cambridge, Mass., Third Edition, 1979.

[Dahl et al 72]
O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare. Structured Programming. Academic Press, 1972.

[Demers et al 81]
Alan J. Demers, Thomas Reps and Tim Teitelbaum. Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors. Proceedings of the Eight Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, January 1981.

[Demers and Reps 81]
Alan J. Demers and Thomas Reps. Private communication, April 1981.

[Dijkstra 76]
Edsger W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.

[Donzeau et al 80]
Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn and Bernard Lang. Programming Environments Based on Structured Editors: The MENTOR Experience. INRIA Technical Report, Rocquencourt, France, May 1980.

[Downey, Sethi and Tarjan 80]
Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the Common Subexpression Problem. _Journal of the Association for Computing Machinery_ XXVII(4):758-771, October 1980.

[Garey and Johnson 79]
Michael R. Garey and David S. Johnson. _Computers and Intractability: A Guide to the Theory of NP-Completeness_. pages 208-209, W.H. Freeman, 1979.

[Gordon et al 79]
Michael J. Gordon, Arthur J. Milner and Christopher P. Wadsworth. _Edinburgh LCF_. _Lecture notes in Computer Science 78_, Springer-Verlag, 1979.

[Gries 71] David Gries. _Compiler Construction for Digital Computers_. John Wiley & Sons, 1971.

[Hansen 71] Wilfred J. Hansen. Creation of Hierarchic Text with a Computer Display. Argonne National Laboratory ANL-7818, Argonne, Illinois, 1971.

[Johnson 81]
Scott D. Johnson. A Computer System for Checking Proofs. Ph.D. Thesis, Department of Computer Science, Cornell University, January 1981.

[Johnson 78]
Stephen C. Johnson. Yacc: Yet Another Compiler Compiler. Computing Science Technical Report, Bell Laboratories, July 1978.

[Joy 80] William Joy (revised by Mark Horton). An Introduction to Display Editing with Vi. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, September 1980.

[Kernighan and Ritchie 78]
Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. Prentice-Hall, 1978.

[Knuth 68] Donald E. Knuth. Semantics of Context Free Languages. Math. Systems Theory Journal, Vol. 2, No. 2, pp 127-145, 1968.

[Knuth 73] Donald E. Knuth. The Art of Computer Programming - Volume 3: Sorting and Searching, pp. 473-479. Addison-Wesley, 1973.

[Kozen 77] Dexter Kozen. Complexity of Finitely Presented Algebras. Ph.D. Thesis, Department of Computer Science, Cornell University, May 1977.

[Krafft 78] Dean B. Krafft. The Assertion Table System for the PL/CV2 Program Verifier. Computer Science Technical Report 78-337, Cornell University, April 1978.

[Krafft 81] Dean B. Krafft. AVID User's Manual. In preparation, August 1981.

[Lesk 75] M. E. Lesk. Lex - A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, New Jersey, October 1975.

[Luckham et al 79]
D. C. Luckham and the Stanford Verification Group. Stanford Pascal Verifier User Manual. Computer Science Department Report No. STAN-CS-79-731, Stanford University, March 1979.

[Mikelsons 81]
Martin Mikelsons. Prettyprinting in an Interactive Programming Environment. IBM Research Rpt. RC 8756 (#37459), IBM T.J. Watson Research Center, March 1981.

[Moriconi 77]
A System for Incrementally Designing and Verifying Programs. Ph.D. Thesis, Department of Computer Sciences, The University of Texas at Austin, December 1977.

[Nelson and Oppen 77]
Greg Nelson and Derek C. Oppen. Fast Decision Algorithms Based on Union and Find. Proceedings of the Eighteenth Annual Symposium on the Foundations of Computer Science, pages 114-119, IEEE Computer Society, 1977.

[O'Donnell 81]
Michael J. O'Donnell. Private communication. July 1981.

[Reps 81] Thomas Reps. Optimal-time Incremental Semantic Analysis for Syntax-directed Editors. Computer Science Technical Report 80-453. Cornell University. March 1981.

[Samet 80] Hanan Samet. Efficient On-line Proofs of Equalities and Inequalities of Formulas. IEEE Transactions on Computers C-29(1):28-30. January 1980.

[Skedzeleski 78]
Stephen K. Skedzeleski. Definition and Use of Attribute Reevaluation in Attributed Grammars. Computer Science Department Technical Report No. 340, pp. 44-48 and 83-97. University of Wisconsin-Madison. October 1978.

[Tarjan 75] Robert E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the Association for Computing Machinery XXII(2):215-225. April 1975.

[Teitelbaum and Reps 80]
Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Computer Science Technical Report 80-421. Cornell University. May 1980.

[Teitelbaum et al 81]
Tim Teitelbaum, Thomas Reps and Susan Horowitz. The Why and Wherefore of the Cornell Program Synthesizer. Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981.

[Thompson et al 79]
D.H. Thompson, et. al. AFFIRM Reference Manual. Information Sciences Institute. University of Southern California. 1979.

[Wood 81] Steven R. Wood. Z - The 95% Program Editor. Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981.