AN ANALYSIS FRAMEWORK FOR THE McCAT COMPILER

by
Bhama Sridharan

School of Computer Science
McGill University, Montreal

September 1992

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

In designing compilers for high-performance computers, the development of program analyses and optimizations are of fundamental importance. In order to perform sophisticated code-improving optimizations, it is essential to: (i) design appropriate intermediate representations, and (ii) develop advanced program analyses on these intermediate representations. This thesis deals with the development of structured intermediate representations for the programming language C, and the development of a framework for the implementation of new and sophisticated flow analysis techniques which have been incorporated in the McCAT (McGill Compiler Architecture Testbed) compiler.

In the first part of this thesis we discuss the design and implementation of FIRST and SIMPLE, the structured intermediate program representations used in the front-end of the McCAT compiler. Furthermore, we illustrate how SIMPLE forms a natural platform to perform sophisticated analyses and high-level program transformations.

In the second part of this thesis we describe the development of an analyzer-generator tool which works on SIMPLE to produce both intra- and interprocedural analyzers quickly and efficiently in a structured, rule-based manner. We illustrate the benefits of our tool by automatically generating analyzer modules for interprocedural live variable analysis, reaching definitions and constant propagation.

Résumé

Dans la conception de compilateurs pour ordinateurs de haute performance, le développement d'analyses et d'optimisations de programmes est d'une importance fondamentale.

Afin d'effectuer des optimisations sophistiquées, il est essentiel : (i) de concevoir des représentations intermédiaires appropriées et (ii) de développer des procédures d'analyses avancées sur ces représentations. Cette thèse porte sur le développement de représentations intermédiaires structurées pour le langage de programmation C et sur le développement d'une infrastructure pour l'implantation de nouvelles techniques sophistiquées d'analyses de flux qui ont été incorporées dans le compilateur du projet de recherche McCAT (McGill Compiler Architecture Testbed).

Dans la première partie de la thèse, nous discutons de la conception et de l'implantation de FIRST et SIMPLE : la représentation intermédiaire structurée de programme utilisée dans la phase initiale du compilateur McCAT. De plus, nous illustrons comment SIMPLE forme une plate-forme pour effectuer des analyses sophistiquées et des transformations avancées de programmes.

Dans la seconde partie de la thèse, nous décrivons le développement d'un outil de génération d'analyseurs s'exécutant sur SIMPLE pour produire à la fois des analyseurs intra- et interprocedurals, et ce rapidement, efficacement, d'une manière réglée et structurée. Nous illustrons les avantages de notre outil en produisant automatiquement les modules d'analyse interprocédural pour les variables et les définitions accessibles ainsi que pour la propagation des constantes.

Acknowledgements

I wish to thank my supervisor, Prof. Laurie J. Hendren, for so many things that I don't know where to begin. First of all, I wish to thank her for the complete support provided by her without which my Masters program at McGill would not have commenced at all. Her excellent guidance, constant encouragement and the unfailing trust in me have contributed a great deal to this thesis. She is more than a supervisor to me; she is a very good friend. I will always cherish my association with her. Thanks for everything, Laurie!!

I wish to thank Prof. Guang R. Gao for all his extremely interesting courses and seminars. It is impossible not to be inspired by his enthusiastic lectures and stimulating discussions in his favorite subjects: computer architectures and compilers.

Next, I wish to acknowledge the help and support received from the members of the McCAT group. The discussions I had with Maryam, Sreedhar, Hans, and Chris greatly improved my understanding of the subject. I am thankful to Weiren, Maryam, Sreedhar, Justiani, Ana, and Chris for discovering many bugs in my software. Special thanks goes to Ravi for his help with C and for making the atmosphere around the NeXT machines as lively as possible. I wish to thank Chandrika, Sreedhar and Chris for their valuable comments on the initial draft of this thesis, and Michel for translating the abstract in French. Maryam, Chandrika and Lakshmi have been my close friends and confidants, whose friendship I hope will continue for many more years to come.

The loving care and affection received from my parents and other family members have given me the moral strength to continue and complete my Masters program. Our special friend Venku deserves more than a mention for his constant encouragement.

Last but not the least, I am extremely grateful to my husband, Govind, but for whom I would never have gotten into the trouble of understanding a million lines of

GCC code. He is my own special friend, without whose moral support and constant encouragement I would not have successfully completed this research program.

Dedicated to the memory of my paternal grandfather Sri. A. Parthasarathy
and to my maternal grandfather Sri. R. Raman.

Contents

List of Figures

Chapter 1

Introduction

Rapid advances in VLSI technology have provided new challenges to compiler and architecture designers in the development of both uniprocessor and multiprocessor systems. In order to effectively exploit the ample resources provided by these new architectures, aggressive compilation techniques and innovative architecture designs are essential. New approaches in compiler technology are required to suit the different architecture design philosophies emerging today, from RISC machines to multiprocessors to multithreaded architectures. It is essential that compilation techniques and architecture models are developed together, so that the effects of one on the other can be studied.

The design of a good optimizing or parallelizing compiler is crucial in the development of high performance single and multi-processor architecture systems. An optimizing compiler performs a series of code-improving transformations, such as constant propagation [WZ85], common subexpression elimination [ASU86], and instruction scheduling [Alt90, BEH91, GM86, Lam90, Muk91], before producing efficient machine code. In order to perform any sort of optimizing transformation, it is essential to collect accurate information about the variables used in the program. Data-flow analysis is a process of collecting information about definitions and uses of variables in a program. Typical examples of traditionally performed data-flow analyses are reaching definitions, live-variable analysis, and last-use information. A different kind of analysis that is receiving an increasing attention is alias and array dependency analysis[Bar78, Ban79, Coo85, CK89, LR92, Lan92, HDG+92, Ema92]. Alias analysis determines whether or not two variables refer to the same memory

location at any point during program execution. Optimizing compilers make use of data-flow and alias information to produce efficient code. Furthermore, parallelizing compilers need this information to extract parallel threads from a sequential program.

Intra-procedural data-flow analysis, i.e., analyzing one procedure at a time [ASU86], has been widely studied and implemented in existing compilers. Gathering information about many interacting procedures, known as interprocedural analysis, is essential to accurately analyze large programs. Performing interprocedural analysis is much more challenging, especially in the presence of recursion. Analysis of pointer and structure variables [HN89, Deu92, HDG+92, LR92, Lan92, Ema92] is critical too, particularly when one wants to handle non-scientific programs.

While the need to produce highly efficient code is generally on the increase in all application areas, modern RISC architectures demand much more from the compiler designer. In particular, register allocation and instruction scheduling are vital for high performance in RISC-based processor systems. Further, with all the other sophisticated and novel architecture designs emerging today, like superscalar and multithreaded architectures, we need new compiling techniques for these as well. Thus, developing the necessary framework and tools for analyzing programs to produce efficient code for a variety of architectures is crucial in compiler design and forms the main goal of this thesis.

1.1 The McGill Compiler Architecture Testbed

The design of any high performance processor system requires architectural designers and compiler writers to coalesce their efforts and work hand-in-hand. To study quantitatively the effect of various compilation techniques on sophisticated architectures, it is necessary to develop a complete compiler-architecture testbed. The first component of such a testbed is the compiler that supports both high-level and intermediate-level compiler transformations that translate high-level programs to low-level programs suitable for an architecture simulator. The second component consists of architecture simulation tools that process the output of the compiler to produce a variety of performance results. The McGill Compiler Architecture Testbed(McCAT) is being designed and developed with the above objectives in mind.

Figure 1.1 presents an overview of McCAT and its major components. As shown in the figure, the input program passes through three phases before it is converted into

machine code. For each of the three intermediate phases, different kinds of Abstract Syntax Trees (ASTs) are used to represent the program. An AST was chosen to be the intermediate representation since it retains all the information about the input program in a structured manner.

The programming language C was chosen to be the source language for our compiler because of its wide-spread use. Further, C is very powerful and supports a wide variety of programming language constructs and user defined data types. We decided to take an existing C compiler and modify its front-end to meet our goals. The reasons for this choice are: (i) it relieves us from the mundane task of writing the parser and the lexical analyzer; (ii) the front-end of a production compiler is more reliable and supports all constructs of the language; and (iii) it generally has a very good error recovery. For these reasons, the front-end of McCAT compiler is based on the GNU C-compiler. The source code for GNU C[1] compiler is freely available.

1.2 Thesis Contributions

This thesis concentrates on the development of the compiler component of the testbed. In the compiler component, our interests are in the front-end of the compiler, which parses the input program and translates it to an intermediate form, and the analysis components of the compiler which operate on the intermediate form. The first important contribution is the design and development of intermediate representations suitable for various high-level analyses and optimizations. A major portion of the analyses and optimization transformations takes place on the intermediate code. Thus the appropriate choice of an intermediate representation is vital in the design of an optimizing compiler. Traditionally, the intermediate code consists of three-address statements and control flow graphs[ASU86]. The program represented by these three address statements is partitioned into basic blocks, where each basic block consists of a sequence of consecutive statements with no branches in between. The optimizations are performed on control flow-graphs, in which the edges represent flow of control and the nodes represent basic-blocks. The main drawback of this representation is that the types of loops and the structure of the program in general is completely lost.

In order to perform accurate optimizing transformations, it is necessary to retain the structure of a program and high-level information about the data structures used.

[1]GNU C is a production compiler developed and distributed by Free Software Foundations Inc

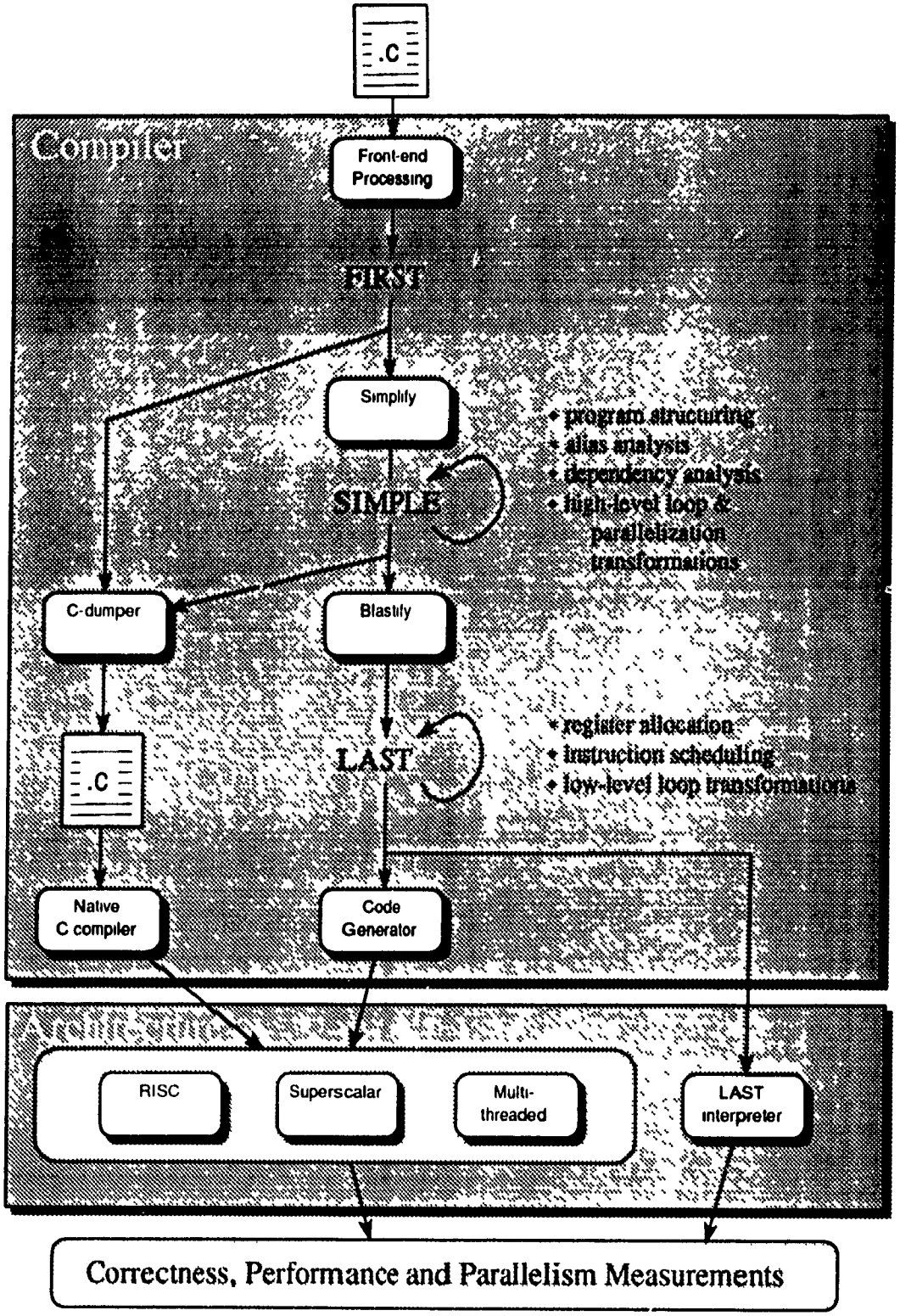


Figure 1.1: The McCAT System

For example, high-level optimizations like loop unrolling [DH79] and loop transformations [PW86, ACK87, Wol89] require the structure of the program and the identity of record and array references to be retained in the intermediate representation. Thus the first part of this thesis concerns the design and development of the front-end of a compiler which translates C programs to a structured intermediate representation that is suitable for performing various high-level optimizing transformations.

We modified the GNU C Compiler front-end significantly to create the FIRST Abstract Syntax Tree (henceforth called FIRST) for the entire program. Next, a series of tree transformations are performed on FIRST to create SIMPLE, which is a simplified AST and forms the intermediate tree representation for optimizing transformations in our testbed. The grammar of SIMPLE is powerful enough to incorporate all of the C language constructs [2], yet simple and regular enough so that the optimization and analyses rules can be specified in a structured and straightforward manner.

The second important contribution of this thesis is the development of a analyzer generating tool called McGill Tree-based Analyzer Generator (McTAG) which can be used to produce various interprocedural data-flow analyzer modules. Such program generating tools are popular because they (i) simplify the task of a compiler writer and (ii) produce certain well-defined and straightforward parts of a compiler quickly and reliably. Automating the generation of compilation modules has met with reasonable success. Program-generating tools like LEX [LS75], YACC [Joh75] and BURG [FHP92] are used extensively to automate producing the lexical analyzer, the parser, and the code generator respectively. The optimization phase, however, is complicated and cannot be automated so easily. One reason for this is that there is a wide variety of optimizations and program transformations reported in the literature; some of these optimizations may interfere with one another and the questions of which subset of these optimizations to apply and in what order are still unsolved. However, the different data-flow analyses can be classified into certain categories; these could then be automated given a certain set of specifications. For example, the analyses could be classified as either forward or backward ones and the merge operator could be either union or intersection.

The framework for McTAG is based on SIMPLE, and the data-flow analyzer automatically generated by the tool also works on SIMPLE. Structure-based algorithms are used to implement the data-flow analyses; this has the limitation that it cannot

[2] Restructuring of programs can be performed to eliminate arbitrary goto's [Ero92]

support arbitrary goto's, but has the advantage that the rules for the different kinds of language constructs can be specified in a neat, compositional and elegant manner.

In summary, the main contributions of this thesis are:

- The creation of a high-level AST representation of the program, FIRST;
- The design and implementation of SIMPLE, a simplified intermediate tree representation, which retains the program structure so that high-level data analyses and program transformations could be performed in a relatively straightforward manner;
- The design of a general framework for structural interprocedural data-flow analysis, which handles recursion;
- The design and development of a tool which automatically generates different intraprocedural and interprocedural data-flow analyzers for a set of input specifications.

1.3 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2 we describe the major modifications made to the GNU compiler front-end to create FIRST and the series of transformations implemented to create SIMPLE. In Chapter 3 we present our data-flow analyzer tool and the framework on which the data-flow analyzer tool is based. We illustrate the effectiveness of our tool using concrete examples of different data-flow analyzers automatically generated Chapters 4 and 5. Our case studies include constant-propagation, reaching definitions and live-variable analysis. Finally, we put forward some conclusions and give suggestions for future work after discussing related research in these areas.

Chapter 2

Design and Development of SIMPLE

The proper choice of intermediate representations is crucial in the design of any optimizing or parallelizing compiler. As shown in Figure 1.1, in our compiler, we have designed a family of three intermediate representations, FIRST, SIMPLE and LAST. These intermediate representations range from a high-level abstract representation, FIRST, that accurately captures the original program, to a low-level representation, LAST, that is suitable for register allocation, instruction scheduling, and code generation. The design of each intermediate representation is driven by the requirements of the analyses and transformations that we considered most important. In addition, we considered how each intermediate representation related to the next lower level so that the results of the analysis performed at a higher-level representation could be used at lower-level representations.

In this chapter, we describe the design features of FIRST and SIMPLE. Section 2.1 describes the creation of FIRST. FIRST retains program and data structures as is written by the programmer. Analyzing a program with such an intermediate representation would be more involved in the presence of complex structures, especially if the programmer has resorted to various tricks allowed by the high-level language. In order to make the analysis simple and regular, we transform FIRST to SIMPLE. In SIMPLE, complex program and data structures are represented in a simplified form. Section 2.2 discusses the design criteria of SIMPLE. As the name suggests, the grammar of SIMPLE is simple, yet powerful enough to represent all constructs of C.

In Section 2.2, we describe the various tree transformations used in the creation of SIMPLE. In chapters 4 and 5, we illustrate how the design of SIMPLE is eminently suited for high-level analyses and optimizations.

2.1 Design of FIRST

As seen from Figure 1.1, the first translation step converts a C program to FIRST, a high-level abstract syntax tree representation. The main purpose of the FIRST intermediate representation is to cleanly separate the front-end processing namely, parsing and type-checking from the back-end processing viz. analysis, transformation, and code-generation. Since this translation step originated from the GNU C compiler, there was a natural abstract syntax tree form that already existed at the expression level. We have extended this form so that it captures completely and accurately the structure of an entire module or program. An important characteristic of FIRST is that all information about declarations, types, and type casting is completely and accurately encoded. In this section, we first give a brief description of the original GNU C compiler, and then describe the major modifications made to it to create FIRST.

2.1.1 The Original GNU C Compiler

The front-end of the McCAT compiler is based on the GNU C compiler(version 1.37.1). In this section, we present some aspects of the front end of the original GNU C compiler (GCC), that are relevant to further discussions in this and in the following chapters of the thesis.

In the original GCC compiler [Sta90], the intermediate representation employed is the Register Transfer Language(RTL), in which each statement has almost a one-to-one mapping to a machine-level instruction. The parser parses each statement of the program, builds a syntax tree for it and then converts it into RTL. Once the RTL for a statement is generated, the storage used for the syntax tree is reclaimed. In this manner, the RTL intermediate code for an entire function is generated. Several intraprocedural optimizations are performed on the RTL to produce the target code for one function. In a similar way, the assembly code for a program is generated, one function at a time.

Storage for types, declarations, and the representation of binding contours and how they nest remains until the completion of the compilation of a function. After generating code for a function or a top-level declaration, all storage used by the function definition is freed completely unless the function is 'inlined'. As a result, interprocedural optimizations cannot be performed by the original GCC.

Tree Node Structure:
We briefly describe the general structure of a tree_node used in the syntax tree of the original GCC. The syntax tree is built from different kinds of tree_nodes; there are distinct tree_nodes to represent various data types and expressions in C. A tree_node consists of two basic parts:

- A common part, which is present in all tree nodes. The major fields found here are:
  - Tree_uid : Every tree node is associated with an unique integer as part of its identification.
  - Tree_type: This points to the type of the node.
  - Tree_code: This contains an enumerated type integer, which essentially gives the name of the node.
  - Tree_chain: This field is used to chain nodes together.
  - Tree_bit_fields: There are a number of bit fields, to indicate various node characteristics.

  Figure 2.1 illustrates the macros used to access these various fields. The Tree_info field is an extra field that has been added while creating FIRST to store data-flow analyses information. We shall see how this field is used in the next chapter.

- A tree specific part, where different node types have different fields in them. For example, the PLUS_EXPR node has two specific fields to hold its two operands. A TREE_LIST node, which is a general purpose node used for chaining nodes together, has the specific fields TREE_PURPOSE and TREE_VALUE.

2.1.2 Creation of FIRST

The two major disadvantages of working with the front-end of GCC are:

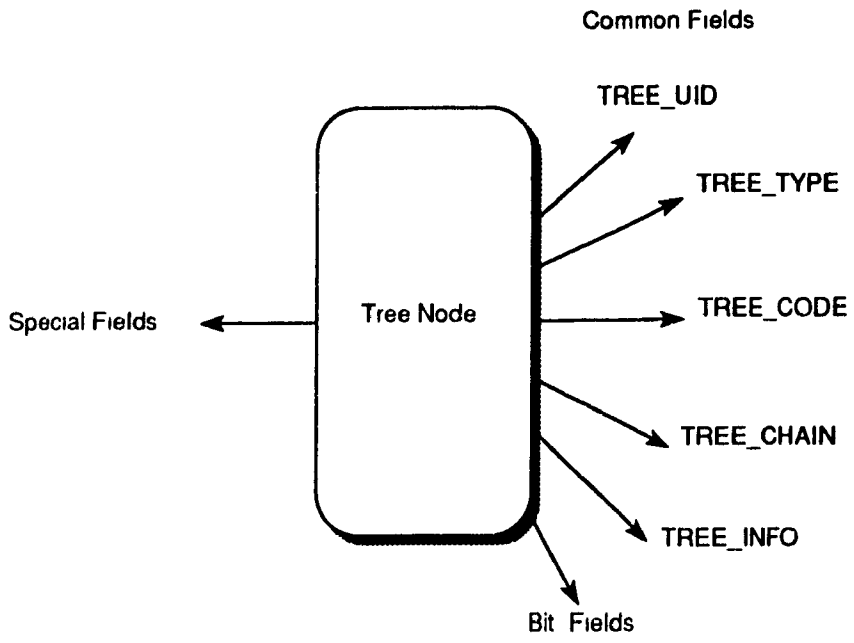


Figure 2.1: Basic Tree Node Structure

1. High-level compiler optimizations such as loop transformations and array optimizations are extremely difficult to perform on the RTL code because the identity of loop structures and array references are completely lost at this low level. For example, array dependence analysis is difficult to perform as array references are broken down to lower-level statements. Similarly, since loops are transformed into blocks with goto's and labels, high-level loop transformations are difficult to perform. Retaining the identity of loop structures and array references enable us to perform a number of high-level loop transformations which are far more difficult to perform otherwise.

2. Since the intermediate code for the entire program is not available, high-level interprocedural analysis and detailed alias analysis techniques cannot be performed.

We modified the front-end of the GCC compiler significantly to create FIRST. These modifications are highlighted below.

- In the original GNU C compiler, as mentioned above, the syntax tree is created up to the expression level. The tree nodes representing these expressions are

freed once the statement has been parsed. Since we are now interested in building the syntax tree for the entire program, we modified the parser so as to retain the tree nodes.

- We further modified the parser to continue building the AST for the complete program. New nodes to construct different kinds of loop structures, such as, while-loop, for-loop, and do-loop constructs, have been added. The tree_node structure used in the modified front-end is based on that of the original GCC. The parser now builds the FIRST tree for the entire program.
- An additional field, the TREE_INFO field, has been incorporated in the basic tree_node structure to store data-flow analysis information.

Notation:

Before we proceed any further, certain conventions that are used in the diagrams are explained here. Boxes represent tree nodes, and the text within them represents the name of the node. Arcs from one node (say 'a') to another (say 'b') means that a particular field in the starting node 'a' points to node 'b'. Arcs are labeled with the macros which may be used to access the particular field of the node. Sometimes the labels on the nodes or arcs are shortened for the sake of clarity; the full names are listed as a legend in the diagram. Certain subtrees which are irrelevant to the current discussion are denoted by ellipses and the details of these subtrees are not shown in the figure; these are labeled with the respective C expressions that the subtrees stand for.

2.1.3 Representation of Statements in FIRST

We illustrate the construction of FIRST using the following examples.

We first consider an example consisting of two simple statements.

```
a = b + c;
f(a);
```

Figure 2.2 depicts the AST for the above example. Every statement is headed by a TREE_LIST node. The TREE_VALUE field of this node points to the tree for that statement. The statements are connected together by the TREE_CHAIN field in the TREE_LIST node.

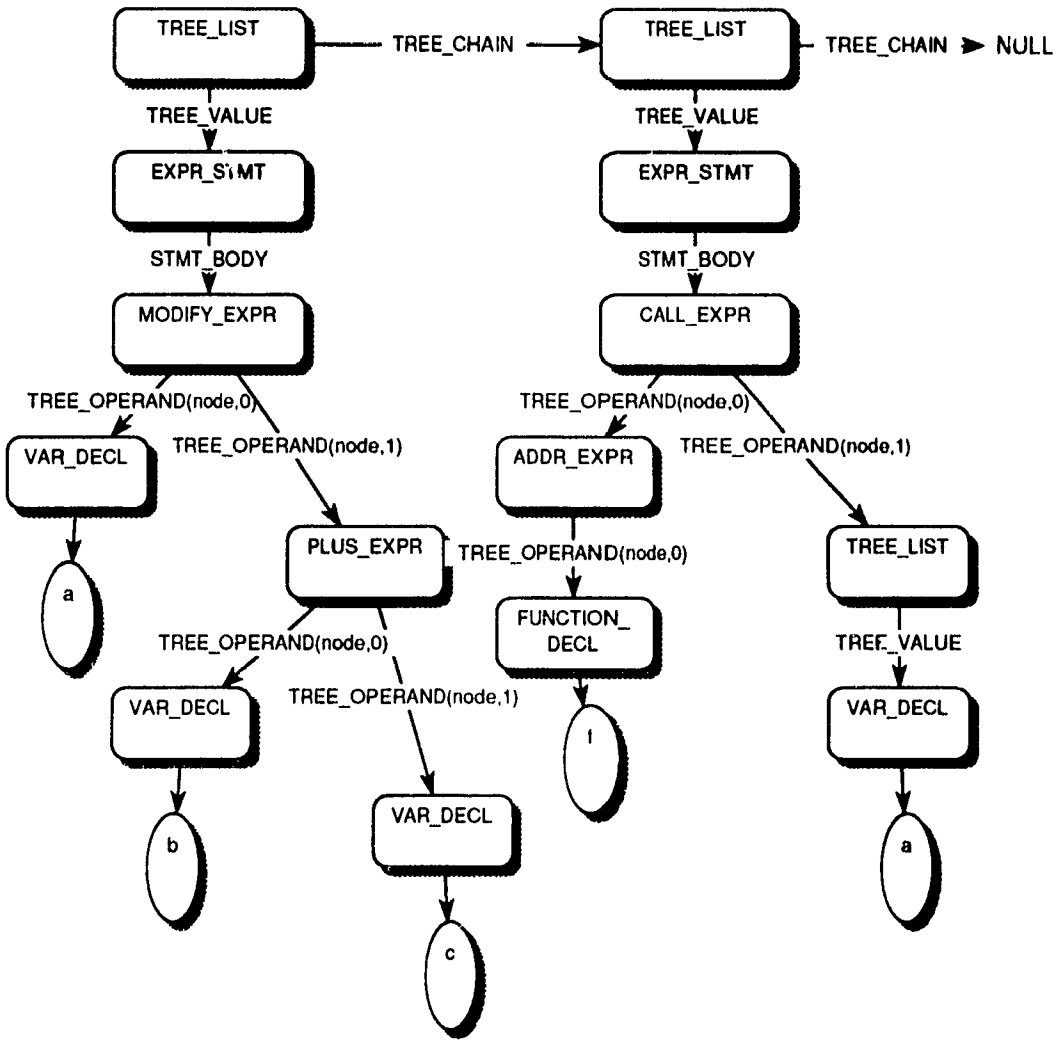
Next, we illustrate how a set of compound statements are represented in the AST and how the variable nestings are taken care of. Consider the following example:

```
{
    int i;   ..............(1)
    stmt1;
    {
        float j; ...........(2)
        stmt2;
    }
    stmt3;
    {
        int j; .............(3)
        {
            float i,k ; .....(4)
            stmt4;
        }
    stmt5;
    }
}
```

The Abstract Syntax Tree constructed for the above example is given in Figure 2.3.

A list of statements consists of a sequence of TREE_LIST nodes connected by the TREE_CHAIN field and terminated by a NULL pointer. Every statement is headed by a TREE_LIST. In the case of simple statements, the TREE_PURPOSE field of a TREE_LIST node is NULL, and the TREE_VALUE field points to the body of the statement. The TREE_PURPOSE field for compound statements points to the LET_STMT node, which contains the scoping information. The LET_STMT is used to hold the variables declared in that block.

A VAR_DECL node represents a unique variable, and has fields pointing to the name of the variable (DECL_NAME) and the type of the variable (TREE_TYPE). Note



EXAMPLE : two consecutive simple statements

```
a = b + c,
f ( a );
```

Figure 2.2: Examples of Basic Statements

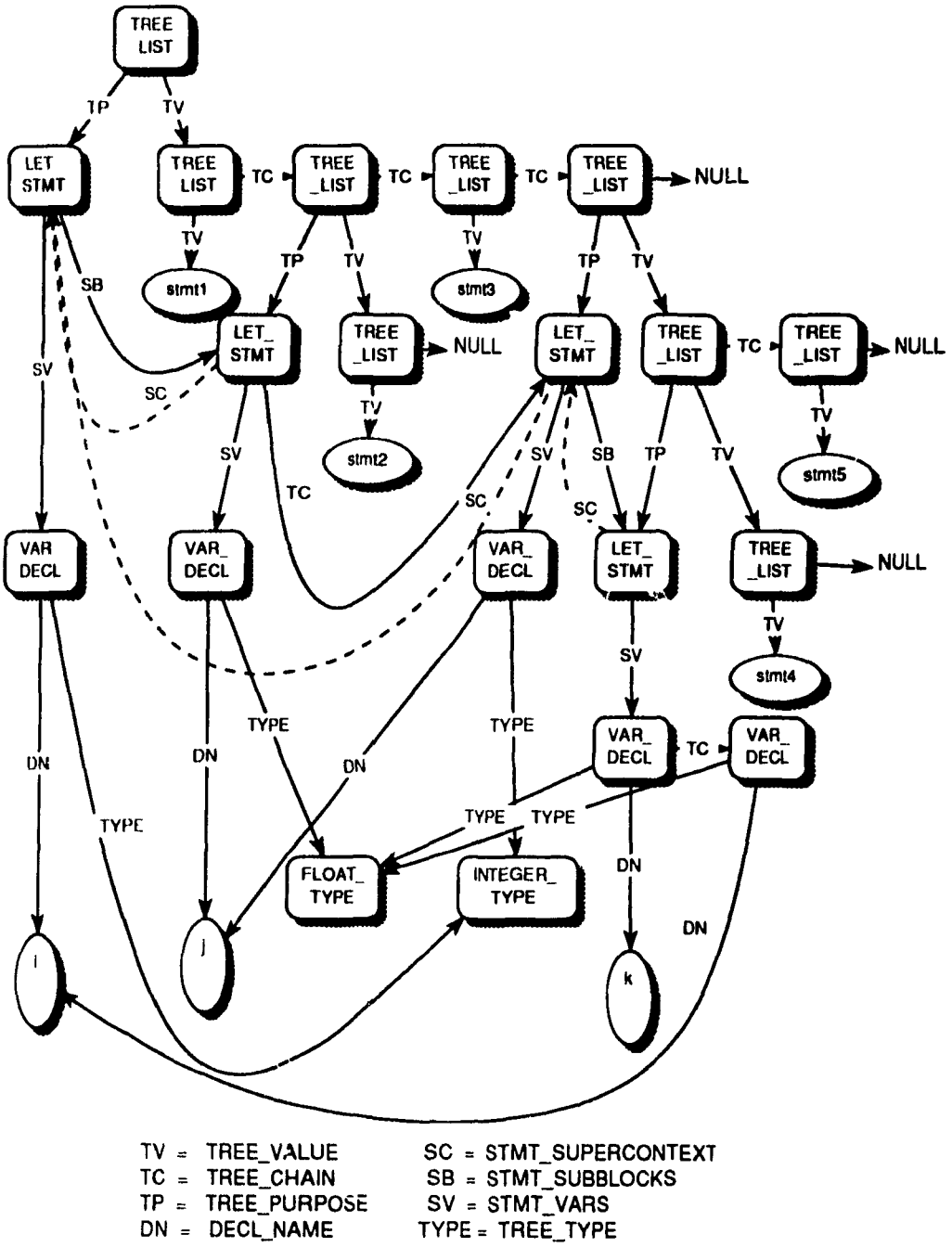


Figure 2.3: Examples of Compound Statements

that two variables which are in two different scopes but have the same name are represented by two different VAR_DECL nodes, though the DECL_NAME field in both of them point to the same IDENTIFIER_NODE. For example, the variable 'i' in the declaration (1) and the variable 'i' in declaration (3) have their own unique DECL_NODE, but the DECL_NAME in both of them point to the same IDENTIFIER_NODE having the string "i".Thus, every logically different variable has its own VAR_DECL node, and an IDENTIFIER_NODE is created for every unique variable name string in the program.

The LET_STMT blocks at nested levels are connected by the STMT_SUPERCONTEXT and the STMT_SUBBLOCKS fields. The STMT_SUPERCONTEXT pointers are backward pointers and are represented by dashed lines in the figure. The LET_STMT blocks at the same level are connected by TREE_CHAIN fields.

2.1.4 Global Functions and Variables

We now illustrate how top-level declarations and definitions of global variables and functions are represented in an AST via the following example.

```
int i;
int func1(int p) {
    int x,y;
    body_func1;
}
typedef struct mm {
    ...
} nn;
void func2(int q) {
    int a,b;
    body_func2;
}
```

The ASTs constructed for the above program are shown in Figure 2.4.

In general, each function definition in C constitutes one AST. The complete AST for a function is headed by an AST_DECL_NODE. The AST_DECL_PTR field points to the function declaration, and the AST_DECL_BODY field points to the compound statement which is the body of the function. In the case of global variables, the

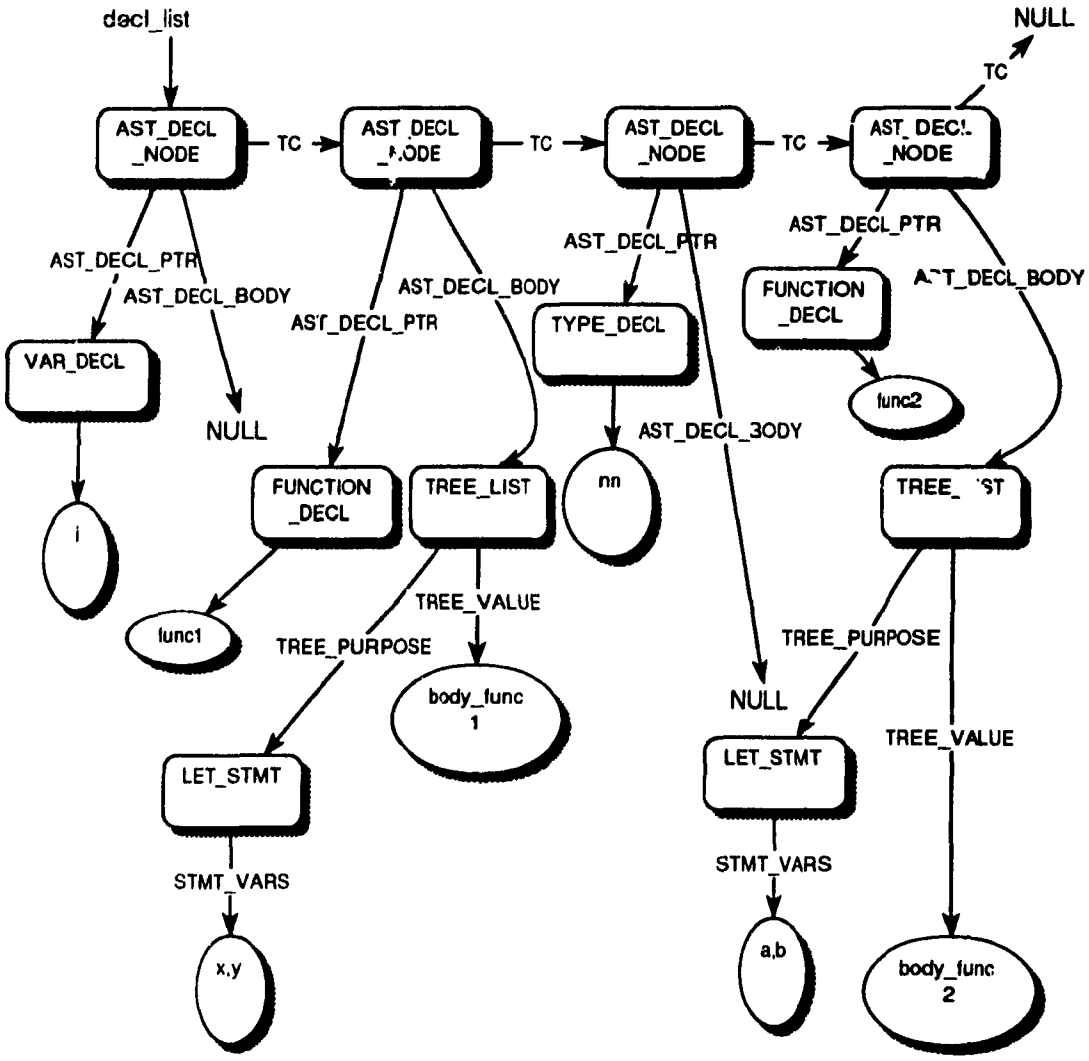


Figure 2.4: Representation of Global Variables and Functions

AST_DECL_PTR field points to the VAR_DECL or the TYPE_DECL node, and the AST_DECL_BODY field points to NULL. All these ASTs for global functions, variables and types are chained together into a linked list structure in the order in which they are parsed. Thus, global variables are found in this top-level list of AST_DECL_NODEs, whereas local variables are attached to the LET_STMT node. Note that *decl-list* is a global pointer which provides access to the list of ASTs. Thus, FIRST retains program and data structures as is written by the programmer. For more details about the working of the GNU C parser and the creation of FIRST, refer to [Sri91].

In order to simplify the analysis of a complicated program, we need to break down complex structures into simpler ones, to a level that is most suited to high-level analysis and transformations. Therefore, we transform FIRST to SIMPLE, the intermediate representation that has been designed to support alias and array dependency analyses, and high-level loop and parallelization transformations. In the following sections, we discuss the design and implementation of SIMPLE.

2.2 Design of SIMPLE

The second intermediate representation, SIMPLE, has been designed to be most suitable for high-level analyses like accurate alias analysis and dependence analysis. The following criteria have influenced the choices made in the design of SIMPLE.

Compositional Representation: The intermediate representation should be a compositional representation of the program, where the control flow is regular and explicit. For example, it should be possible to analyze a while loop by analyzing only its components: the conditional expression and the body. This kind of compositional representation has three advantages: (i) the flow of control is structured and is explicit in the program representation, (ii) structured analyses techniques and tools supporting such techniques can be used to analyze all the control-flow constructs, and (iii) it is simple to find and transform groups of loop nests. It should be noted that in addition to ordinary compositional constructs such as conditionals and loops, our compositional approach should directly support the commonly used `break` and `continue` statements for loops, and the `return` statement for procedures and functions. However, unrestricted

use of goto is not compositional and cannot be supported directly. Any program with unstructured control flow must be converted into an equivalent program with structured control flow [WO75, Bak77, Amm92].

Explicit Array and Structure References: The identity of array and structure references should be retained i.e., the array and structure references should not be broken down into a series of lower-level statements that perform address calculations. This is required so that we can make full use of high-level information such as array dimension, array size, pointer types, and recursive structure types.

Types and Typecasting: The exact type information and type casting should also be retained. Often alias analysis can take advantage of type information to provide more accurate results. For example, it can be inferred that a variable of one type cannot be aliased to a variable of another type using type information if there are no type casts. A more advanced example is the use of recursive types for dynamically-allocated pointer structures [HN89, Deu92].

Pervasive Data-flow Information: It should be possible to transmit important data-flow information collected at higher-level intermediate representations to the lower-level representations, and thus improve the effectiveness of the low-level transformations. For example, alias analysis information collected at a high-level can be used to perform better dependence analysis and therefore better instruction scheduling at a lower-level.

Simple to Analyze: The intermediate representation should be simple enough so that it could be analyzed in a straightforward manner. A proficient programmer will use complex structures and all the tricks allowed by the language (especially C!) to develop his/her software. To simplify accurate analysis of such a program we need to break down complex structures and statements into simpler ones. There should be a restricted number of basic statements so that the structured analyses rules could be specified easily and in a regular fashion. Further, we should be able to represent any complicated C statement or expression as a sequence of these statements. Similarly, the conditional parts of while-loops, for-loops, do-loops and if-statements should be simple expressions. Any complicated expression should be simplified when represented in the intermediate representation. Furthermore, in C, side-effects can occur in many places where one expects an expression. In our simpler form we would like to clearly separate statements that can have side-effects from expressions that cannot have side-effects.

Clear Semantics: The intermediate representation should have a clear and obvious semantics. One part of this process is clarifying some of the implicit meanings in C programs. The example program in Figure 2.5 illustrates one of the implicit meanings in C. Let us first consider the statement, `b = a`. Since there is a special correspondence between pointers and arrays in C, this statement really means "assign to `b`, the address of the first item of `a`", and not "assign to `b` the value of `a`" as one would expect for a scalar assignment. These implicit semantic rules in C must be made explicit in the intermediate representation. Similarly, the meaning of the two array references `b[2]` and `a[2]` are quite different. In the first case there is an implicit dereference of `b`, while in the second case `a` denotes the address of the array.

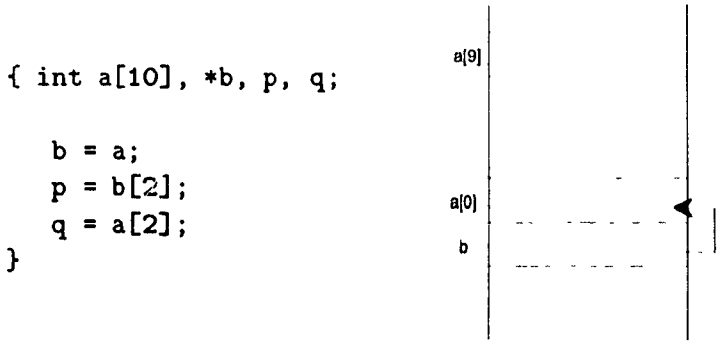


Figure 2.5: An Example of C Array and Pointers

Interprocedural Analysis: The intermediate representation should retain all the information about the complete program or module, so that interprocedural data-flow and alias analyses can be performed. This is particularly important when we have non-scientific code that is composed of many small and possibly recursive procedures.

Standard Representation: Similar to the spirit of DIANA[Ros85], we are aiming at a standard intermediate representation for programs, so that a large number of researchers and students can have a common ground to independently develop analysis and optimization techniques. This also allows a number of users to share a class of software tools.

Automating the Analysis and Optimization Phases: The intermediate representation should be able to support the automatic generation of the analysis

and optimization phases of the compiler.

SIMPLE to C: After performing certain high-level transformations and program restructuring, there should be a straight-forward translation from the intermediate representation back to C. This will help in two ways: (i) we can verify the syntactic and semantic correctness of our transformations by testing the output C program with a standard C compiler, and (ii) we can check the effectiveness of high-level transformations without requiring a complete back-end for every architecture.

2.2.1 Overview of SIMPLE

In this section, we give an overview of the salient features of SIMPLE. As illustrated in Figure 1.1, the *simplify* translation takes a high-level FIRST representation of a program and produces an output representation at the SIMPLE level. As its name suggests, the SIMPLE intermediate form is a simplified form of FIRST; control flow is structured, complex statements are broken down into a series of simpler statements, complicated variable names are split whenever possible and all loops, switches and conditionals are modified to adhere to the restricted SIMPLE format.

The following subsections describe the special features found in SIMPLE, while a complete grammar is given in Appendix A, and a complete description of SIMPLE is given in [Sri92]. While transforming FIRST to SIMPLE, simplifications are performed in these three major areas: (i) in the representation of variables, (ii) in the format of basic statements — only a restricted number of operands are allowed, and (iii) in the representation of control flow constructs — these are simplified.

Variables

In high-level programming languages such as C and Pascal which allow user defined type structures, arbitrarily complex types can be defined by the programmer. One could have nested array and structure references, with pointers in them to further complicate matters. To perform accurate analyses of the variables of these types, we need to break them up in a systematic manner.

As indicated by the grammar rules in Figure 2.6, we define a **varname** to be a simple variable name (e.g., '`myname`'), a pure structure reference(e.g., '`a.b.c`'),

a pure array reference (e.g., 'a[5][7]'), or a pointer to a structure reference (e.g., '(*a).b.c'). Any other complicated array/structure reference is broken down to the abovementioned form.

```
val : ID                          reflist : '[' val ']'
    | CONST                               | reflist '[' val ']'

varname : arrayref                idlist : idlist '.' ID
        | compref                        | ID
        | ID
                                  compref : '(' '*' ID ')' '.' idlist
arrayref : ID reflist                     | idlist
```

Figure 2.6: SIMPLE Grammar for a `varname`

For example, the array/structure reference shown on the left-hand side of Figure 2.7 is transformed into the sequence of statements shown on the right-hand side of Figure 2.7. The variables `temp1` through `temp4` are temporary variable names generated by the *simplify* translation. The translation ensures that these new variables are created with proper declarations and types which are fully represented in the transformed SIMPLE tree. By standardizing the variable references in this way we can reduce the number and complexity of advanced alias analysis rules that must be defined. Thus, structure and array names are kept at the right level for high-level analyses and transformations.

```
f = a.b[3].c.d[2][5].e   ⇒   temp1 = &a.b;
                             temp2 = &temp1[3];
                             temp3 = &(*temp2).c.d
                             temp4 = &temp3[2][5];
                             f = (*temp4).e;
```

Figure 2.7: Variable Transformation

Detailed trees for array reference and structure reference nodes are shown in Figures 2.8 and 2.9.

A COMPONENT_REF node denotes a structure reference. It has two main fields, TREE_OPERAND(`node`,0), and TREE_OPERAND(`node`,1) (where `node` points to the

COMPONENT_REF node), which points to the two parts of a structure reference. For example, in Figure 2.8, the topmost COMPONENT_REF node which denotes the structure reference `e.a.d` points to `e.a` and `d`. The variable `e` is represented by a VAR_DECL node, and its TREE_TYPE field points to a RECORD_TYPE. The RECORD_TYPE node has pointers to the name of the record, and also points to a list of FIELD_DECL nodes corresponding to all the fields in that record. Each of these FIELD_DECL nodes have pointers to their names and types; for example, the field `a` of the structure `bb` is itself a record -- `struct cc`. Notice too that the type of the field `d` in record `cc` is a pointer to the record `cc`; this is denoted by the node POINTER_TYPE, whose type is the record `cc`.

An ARRAY_REF node denotes an array reference. The two main fields of this node points to the base and the index of the array reference. A two-dimensional array is treated in a hierarchical fashion, as an array of arrays. In the example shown in Figure 2.9, the array reference `a[3][4]` is treated as `(a[3])[4]`. The reason for the additional ADDR_EXPR operator is to distinguish between pure array references, and pointers treated as arrays.

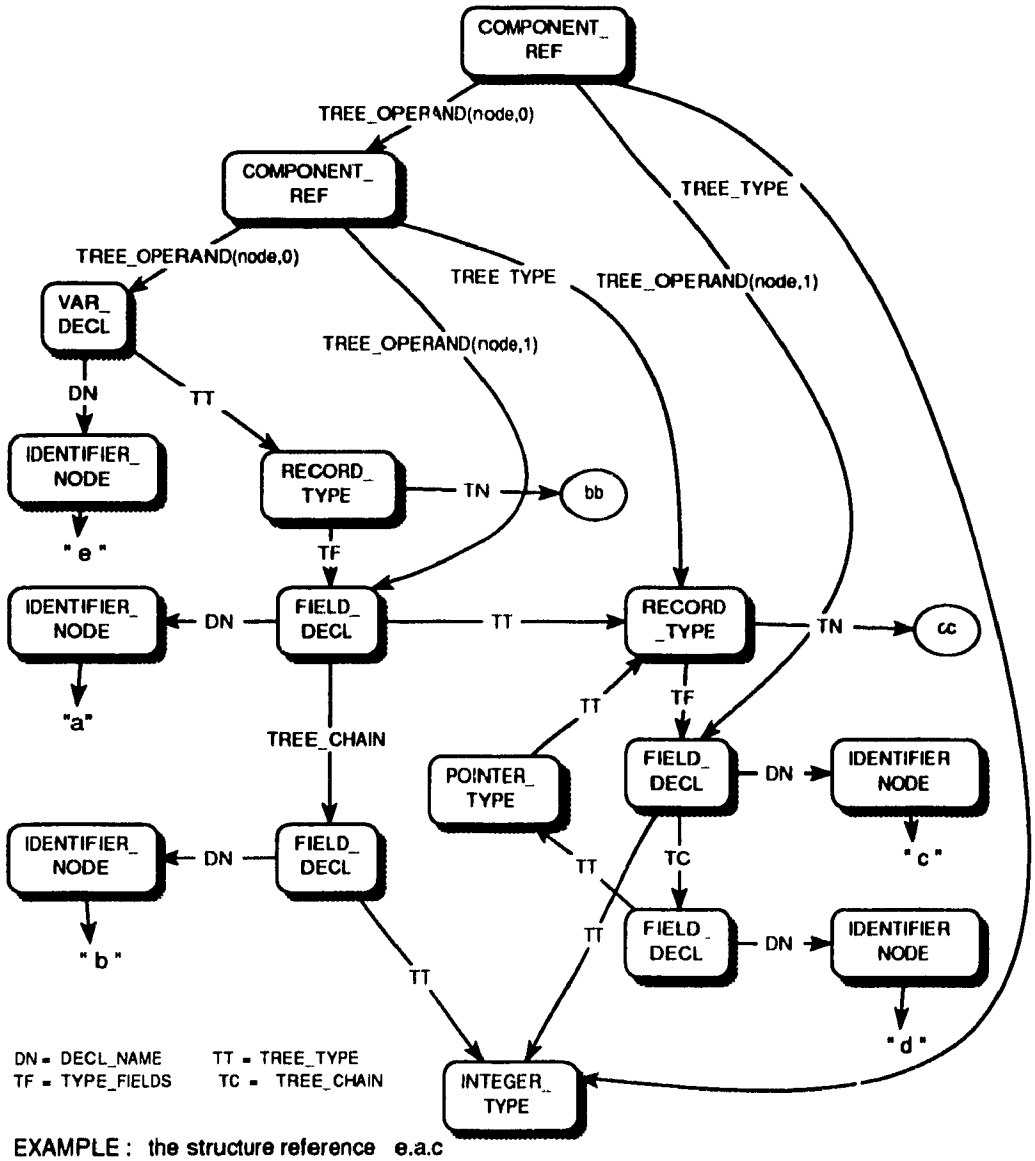
Further, the semantics of C is made obvious wherever possible. In the example shown in Figure 2.5, special ADDR_EXPR nodes are inserted in the SIMPLE tree to clarify where the variables should be dereferenced. Figure 2.10 shows the differences between the trees built for a pure array reference, and for an array reference when a pointer is treated as an array.

Figure 2.11 shows the tree when arrays are assigned to pointers. Since `a` is defined to be an array of integers, and `b` is a pointer to an integer, the statement `b = a` has an ADDR_EXPR node in front of the node for `a` to clarify that the address of `a` is assigned to `b`.

Basic SIMPLE Statements

In the design of SIMPLE, we have identified fourteen basic statements. Any other complicated statement in C can be broken down into a sequence of these statements. Figure 2.12 gives an example of an assignment statement which is broken down into a series of simpler statements. In Figure 2.13, we list the set of basic simple statements.[1] Note that variables '`x`' and '`y`' denote `varnames`, whereas the variables '`a`', '`b`', and '`c`' denote `vals`. Refer to Figure 2.6 for the definition of a `varname`.

[1] These basic statements are formally specified in the SIMPLE grammar included in Appendix A



EXAMPLE : the structure reference e.a.c
where the structures are defined as :

```
struct  bb  {              struct  cc {
     struct  cc  a,           int c;
     int   b;                 struct  cc  *  d,
 }  e ;                     } ;
```

Figure 2.8: Representation of a Structure Reference

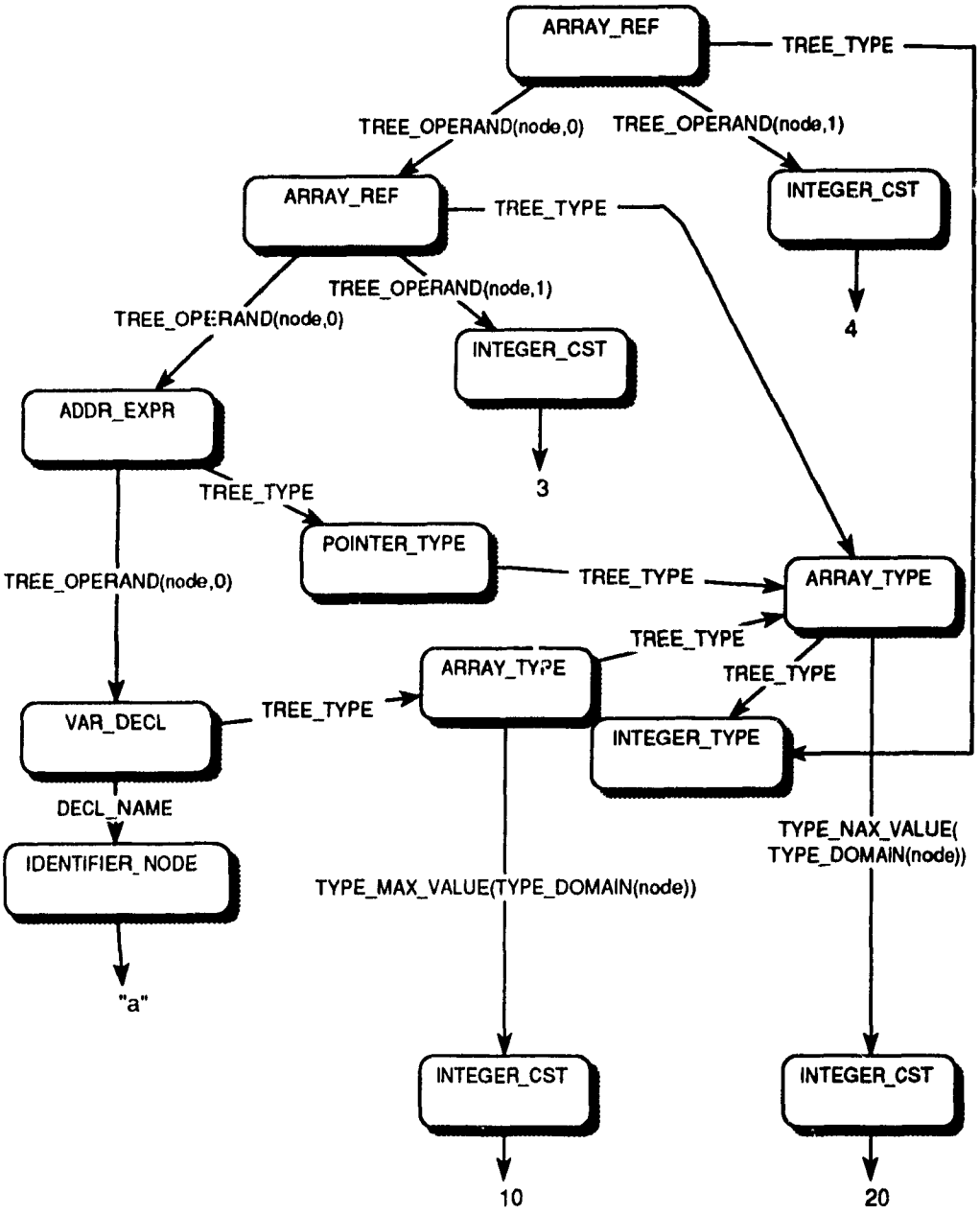
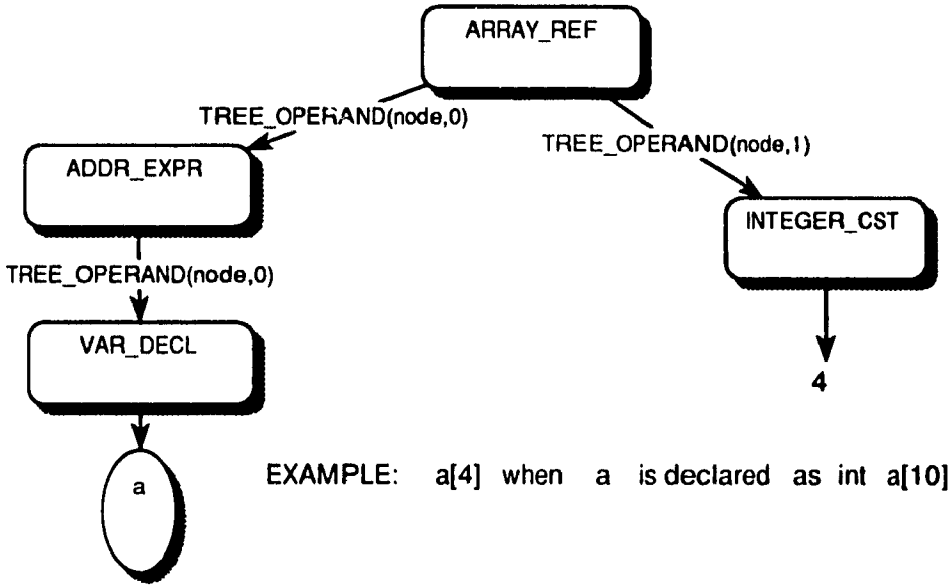
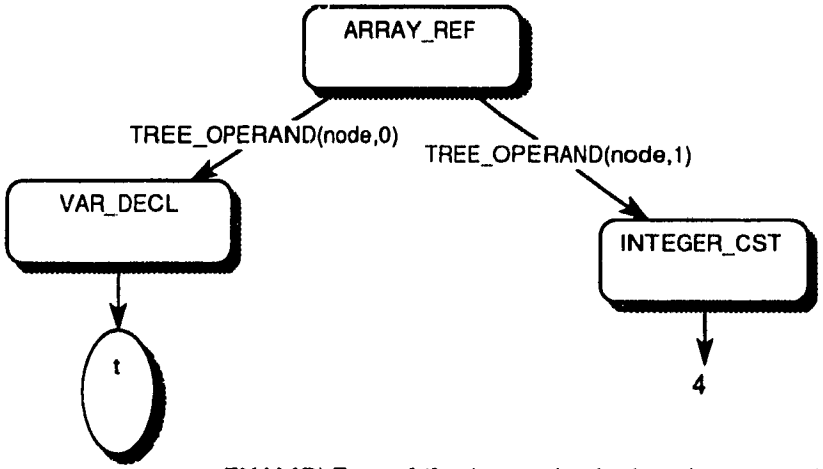


Figure 2.9: Representation of an Array Reference



EXAMPLE: a[4] when a is declared as int a[10];



EXAMPLE. t[4] when t is declared as int * t,

Figure 2.10: Differences in Pointer and Array Representations

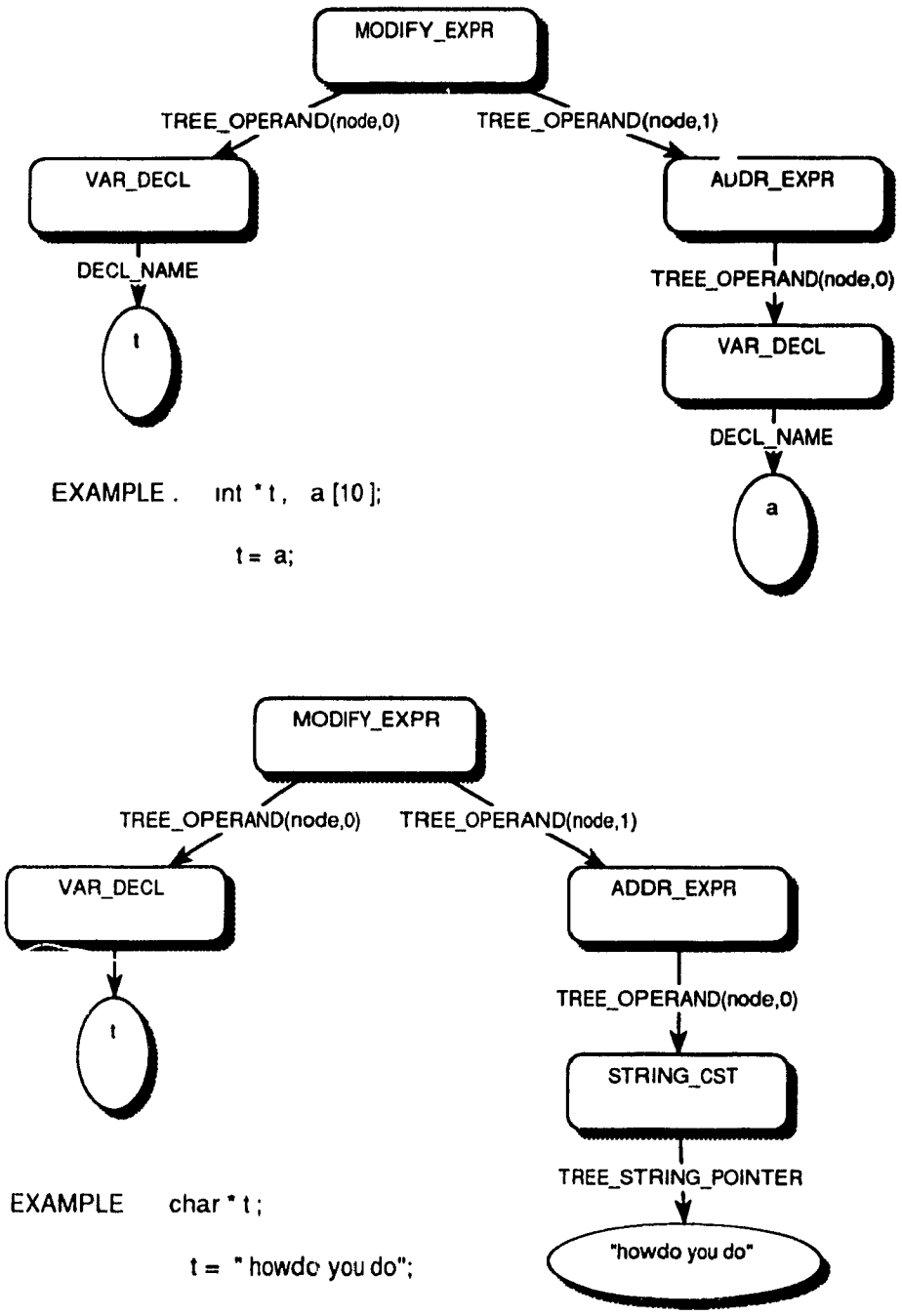


Figure 2.11: Assignment of Array Addresses

```
a = b * c + (*d) / e;  ⇒   temp1 = b * c;
                           temp2 = *d;
                           temp3 = temp2 / e;
                           a = temp1 + temp3;
```

Figure 2.12: Basic Statements Transformation

```
 1.   x  =  a op b     where op is any binary operation
 2.  *p  =  a op b
 3.   x  =  op a       where op is any unary operation
 4.  *p  =  op a
 5.   x  =  a
 6.  *p  =  a
 7.   x  =  f(args)    where args is a possibly empty list of arguments
 8.  *p  =  f(args)
 9.   x  =  (cast)b   where cast is any typecast
10.  *p  =  (cast)b
11.   x  =  &y
12.  *p  =  &y
13.   x  =  *q
14.  *p  =  *q
15.  f(args)
```

Figure 2.13: List of Basic Statements

```
expr : rhs | modify_expr

arglist : arglist ',' val
        | val

modify_expr : varname '=' rhs
            | '*' ID '=' rhs

rhs : binary_expr
    | unary_expr

unary_expr : simp_expr
      | '*' ID
      | '&' varname
      |call_expr
      | unop val
      | '(' cast ')' varname
/* cast here stands for all */
/* valid C typecasts        */
```

```
call_expr : ID '(' arglist ')'

binary_expr : val binop val

unop : '+' | '-' | '!' | '~'

binop :  relop
   | '-' | '+' | '/' | '*'
   | '%' | '&' | '|' | '<<'
   | '>>' | '^'

relop : '<' | '<=' | '>' | '>='
  | '==' | '!='

condexpr : val
         | val relop val

simp_expr : varname
          | CONST
```

Figure 2.14: SIMPLE Grammar for an **expr**

```
c = *(a.b);
d = &(a.b);
```

$\Rightarrow$

```
temp1 = a.b;
c = *temp1;
d = &(a.b);
```

Figure 2.15: The '*' and '&' operators

As seen from the grammar of SIMPLE presented in Figure 2.14, an unary_expression could be either a **varname** or a constant. It could also be an indirect reference. Only a simple variable name is allowed after the '*' operator. Complex **varnames** after the * operator are translated as shown in Figure 2.15. On the other hand, a **varname** is allowed to appear after an '&' operator. Thus the expression `t = & (a.b)` will be kept as it is, and will not be simplified further. The intuition is that when we apply the address operator, we are taking the address of a variable (which could be a complex one), and this should not be simplified into a temporary variable, otherwise we will get the address of the temporary variable, which is incorrect.

We handle casting of expressions and function calls in a similar manner. In a function call, all the complex argument expressions are simplified to **vals**. Casting of expressions is represented by special NOP_EXPR nodes, the TREE_TYPE field of which points to the type to which the expression is being cast. For more details, refer to [Sri92].

Certain special kinds of expressions allowed by C, namely compound and conditional expressions, are also transformed into equivalent SIMPLE statements. For example, conditional expressions are transformed into **if-else**-statements, and compound expressions are transformed to a sequence of simple statements of the above form as shown in Figures 2.16 and 2.17.

```
(a>b)? a : b = c;        ⇒        if (a>b)
                                     a = c;
                                  else
                                     b = c;
```

Figure 2.16: Conditional Expressions

```
c = z = x + y, z > p;    ⇒        z = x + y;
                                  c = z > p;
```

Figure 2.17: Compound Expressions

As illustrated by the example in Figure 2.18, special care needs to be taken to handle expressions involving the logical operators && and ||. This is because, in C, the second operand in the following example is evaluated only if the first operand evaluates to true.

```
c = expr1 && expr2;   ⇒   temp1 = expr1;
                          if (temp1)
                             temp1 = expr2;
                          c = temp1;
```

Figure 2.18: Logical Operators

Compositional Control Statements

The compositional control statement forms supported directly by SIMPLE are restricted (simplified) versions of statement sequences, **for**-loops, **while**-loops, **do**-loops, **switch/case** statements, and **if-else** statements. In addition, **return** is supported for exiting a procedure or function, and **break** and **continue** are supported for exiting a loop.

In each of the simple control constructs the complexity of the conditional expressions is reduced. For example, Figure 2.19 shows how the conditional expression in a **while**-loop is simplified. The conditional parts in **do**-loops, and **if/else**-statements are handled in a similar fashion. **For**-loops are given special treatment because compound expressions are possible in the **for**-loop header.

```
while (a + b > c)          ⇒   temp1 = a+ b;
   {                              while (temp1 > c)
   ...                               {
   }                                 ...
                                     temp1 = a + b;
                                     }
```

Figure 2.19: Simplification of a WHILE Loop Conditional Expression

In simplifying **switch/case**-statements, special care must be taken. In C, the case labels are implicit **gotos** and therefore a **switch/case** statement may have a unstructured control flow. In fact, the **switch/case** statement may not even have structured block nesting because a block may begin in one branch of the **switch** and end in another branch of the **switch**. As illustrated in the grammar detailed in Appendix A, the body of a simplified **switch/case** statement consists of a sequence of **case** statements, finally ending with a **default** statement. Each **case** statement is

made up of one or more case expressions, followed by a statement list, and ending with a stop statement, which could be either a **break**, a **continue** or a **return** statement. Various transformations are performed on FIRST to bring it to the above format. If a **default** statement is missing, one is created and appended to the end of the switch statement. If a case does not end with a stop statement (a **return**, **break** or a **continue**), then code is replicated so that the above form is achieved. If there are unstructured block nestings, these are taken care of by a process called an unnesting transformation where block nestings are removed by yanking variables to the function level[Sre92]. Here, renaming of variables is performed whenever necessary. Figure 2.20 shows an example of some of these transformations.

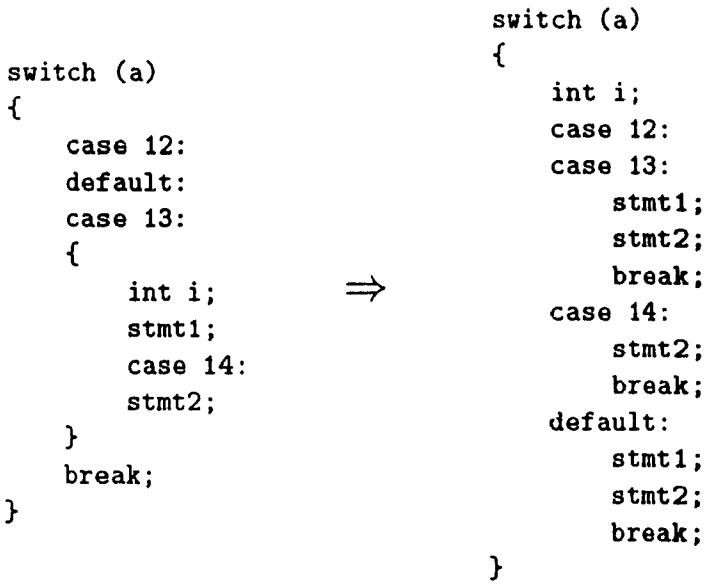


Figure 2.20: A Switch Statement Transformation

2.3 Conclusions

Different types of compiler analyses and optimizations can be performed on SIMPLE. The main advantage of working with SIMPLE rather than with FIRST is that any variable in SIMPLE is either a scalar, a simple array reference, a structure reference or a structure reference through a pointer. Similarly, the type of a statement is

one of a fixed number of types discussed in Section 2.2. The SIMPLE grammar is powerful enough to incorporate all of the C Language constructs, yet simple and regular enough so that the optimization and analyses rules can be specified in a structured and straightforward manner.

Further, we have a developed a `dump-C-routine` which produces C code by performing a tree-walk over SIMPLE. Thus, after performing simplifying transformations on the tree and creating SIMPLE, we can dump the C code corresponding to the SIMPLE tree. This C dump program could be recompiled and executed to verify the correctness of the transformations used to create SIMPLE.

Chapter 3

The Analyser Generator

3.1 Introduction

There are strong similarities among different data-flow analyses problems[ASU86]. The data-flow problems are distinguished by

- The direction of propagation of information: it is either forward or backward.
- The abstraction and the operators used to manipulate the abstraction.
- The type of analysis: it is either intraprocedural or interprocedural.

Since flow analysis problems are well-structured, it should be possible to treat all of them in an unified way. Developing a tool which helps in the implementation of different data-flow analyses by treating them all in an unified fashion will help a compiler writer to produce data-flow analyzers quickly and efficiently.

Traditionally, data-flow analyses were implemented on flow-graphs and many of the analyses were intraprocedural. Another way of implementing data-flow analyses when the input is structured and compositional, is to use a structure-based approach and to specify the analysis rules for each program component. This kind of implementation is closer to how one specifies analyses using abstract interpretation techniques. In this approach one specifies the data abstraction, the operations on

the abstraction, and rules or functions for each kind of program component including simple statements, conditionals, and loops. This sort of approach has been used successfully for many complex analyses problems including a variety of alias analysis problems [HN89, Deu92, Ema92]. Performing these sophisticated analyses as well as the traditional analyses in a rule-based and structured manner has two advantages: (i) it makes the implementation clear even when the abstraction is very complex, and (ii) it simplifies proving the correctness of the implementation. Furthermore, having a structured and compositional representation of the program (like SIMPLE) makes the job of implementing these sophisticated analyses more straightforward and elegant.

In this chapter, we describe the implementation of an analyzer-generator, McTAG (McGill Tree-based Analyzer Generator), which takes advantage of the structured and compositional nature of the SIMPLE intermediate form to provide a straightforward way of specifying new analyses. This tool allows us to quickly implement the traditional flow analyses, as well as providing the backbone for the development of relatively advanced analysis techniques like alias analysis. McTAG takes a set of specifications which describe the data-flow problem as input, and the output is the data flow analyzer module which operates on SIMPLE. The following section describes the overall structure of the generator, and gives examples of the generator input and output. Our approach to generate data-flow analyzers has an additional advantage in that parallelism can be easily exploited in executing these analyzers. In Section 3.3, we make a brief remark on parallelizing these analyzers.

3.2 The Tool

The overall structure of McTAG is shown in Figure 3.1. The generator takes as input a set of specifications for the abstraction being computed (sets, stack matrices, path matrices, etc.) for a particular data-flow problem, and a set of pattern-action rules, one rule for each expression and statement type supported by SIMPLE The output is a data-flow analyzer program which traverses the SIMPLE tree applying the appropriate rules and collecting the specified data-flow information at each program point. The analyzer specifications are completely independent of the SIMPLE tree characteristics; they *only* specify precisely how to collect the particular data-flow information. The generator generates all the extra information pertaining to the SIMPLE tree and generates the tree walk automatically.

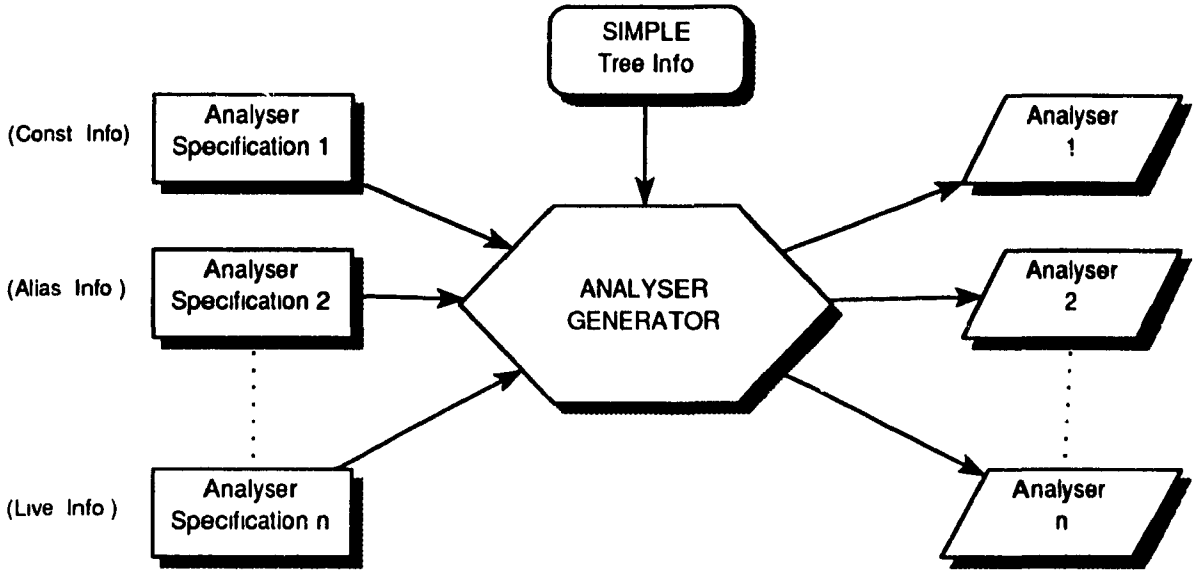


Figure 3.1: Overall Structure of McTAG: McGill Tree-based Analyzer Generator

3.2.1 Input for McTAG

The input specification file is made up of three main parts:

- a C code section, which has all the routines to implement the abstraction,
- the characterization section, which specifies the general characteristics of the data-flow problem, and
- the main section, which has a list of pattern-action pairs, one for every statement/expression type found in SIMPLE.

Thus, the input specification file has the general structure shown in Figure 3.2.

The complete grammar in BNF notation for the above specification is included in Appendix B. We shall now describe each of the above sections in detail by illustrating the specification file for the following data-flow analysis problem.

Let us consider the problem of determining the set of available expressions at every point in a program. An expression `x+y` is available at a point `p` in the program if every path from the initial node to `p` evaluates `x+y`, and after the last such evaluation

```
%{
C code to implement the abstraction  being approximated.
%}

<characteristics of the data-flow problem>

PROCEDURE analyzer_name(parameters):returntype
<pattern> {action}
<pattern> {action}
      ...
```

Figure 3.2: General Structure of the Input Specification for McTAG

prior to reaching p, there are no subsequent assignments to x or y [ASU86]. The abstraction in this case is a set of expressions. Since we want the expression to be available on every path, the merge operation is an intersection. A sample C-code section of the specification file for the above flow problem is shown in Figure 3.3. More detailed examples are presented in Chapters 4 and 5.

As shown in Figure 3.3, the C routines are inserted between the delimiters %{ and %} and these are used in the action part of the specifications. These are the routines that will be used to manipulate the abstraction. Since the abstraction here is *a set of expressions*, there are routines to add expressions to the set, delete expressions from the set, to check if two sets are the same, and, of course, the merge operation has to be defined between two sets. These will be copied verbatim to the output file. It should be noted that the programmer might use a much more complicated abstraction like path matrices for a different problem [HN89].

The characteristics of the specific data-flow problem are specified next. A sample characteristics section for the above available expressions data-flow problem is given below and the various possibilities that can arise are discussed subsequently.

```
%{
typedef struct {
   ...
} * SET_TYPE;
/** This merges set1 and set2 and returns the new merged set **/
static SET_TYPE merge_avail(SET_TYPE set1, SET_TYPE set2) { ... }

/** This returns a copy of the set set1                      **/
static SET_TYPE copy_avail(SET_TYPE set1) { ... }

/** This prints set1                                         **/
static void print_avail(SET_TYPE set1) { ... }

/** This is a routine to build the call graph                **/
static void build_cg(SET_TYPE set1) { ... }

/** This deletes all the expressions in exprlist from        **/
/** set1 and returns the new set.                            **/
static SET_TYPE delete_expr(SET_TYPE exprlist, SET_TYPE set1) { ... }

/** This adds expr to set1 and returns the new set           **/
static SET_TYPE add_expr(SET_TYPE expr, SET_TYPE set1) { ... }

/** This returns all the expressions that have the           **/
/** variable var in them.                                    **/
static SET_TYPE get_expr(tree var) { ... }

/** given the variables and the operators, it creates and    **/
/** returns expressions                                      **/
static SET_TYPE create_expr(tree op, tree var1, tree var2) { ... }

/** given two sets, this routine returns a 1 if they are     **/
/** both the same; 0 otherwise.                              **/
static int same_sets(SET_TYPE set1, SET_TYPE set2) { ... }
%}
```

Figure 3.3: C-code inclusion in the Specification File

```
ANALYSIS : FORWARD
STOREOPTION: STORE
PROCTYPE : INTER
DATANAME: set
DATATYPE: SET_TYPE
TREEINDEX: TREE_AVAIL_INDEX
CGINDEX: CG_AVAIL_INDEX
MERGER: merge_avail
COPIER: copy_avail
PRINTER: print_avail
CGBUILDER: build_cg
```

- ANALYSIS: This is specified as either **FORWARD** or **BACKWARD**, depending on whether it is a forward or a backward analysis. This information will be used when we group the analyses to run a number of analyses in parallel.

- STOREOPTION: If this option is specified as STORE, then extra code is generated by the tool to store the data-flow information in the tree. Otherwise, if the option is NO_STORE, the computed data-flow information is not stored in the tree.

- PROCTYPE: This option specifies whether the analysis is interprocedural or intraprocedural. If the analysis is intraprocedural, then all the procedures are traversed once in random order. If it is interprocedural, then a call-graph is constructed and traversed, collecting information across procedure calls.

- TREEINDEX: This should be specified only if the STORE_OPTION option is specified as STORE. This is used to specify an unique field in the AST the computed data-flow information is stored.

- CGINDEX: This is similar to the TREEINDEX. This specifies a unique index in the call graph where the computed information is stored.

- DATANAME: The name of the set or abstraction is specified here. This is used by the generator to automatically generate a default action when the pattern-action pair for a particular statement type is not given.

- DATATYPE: The type name of the set or abstraction is specified here.

- **MERGER**: The name of the merging routine (to merge two sets of information) is specified here. This is required if the **STORE_OPTION** is set to **STORE**. The merging routine has to have the function prototype as shown below:

  ```
  SET_TYPE merge_avail (SET_TYPE set1; SET_TYPE set2);
  ```

  That is, it takes in two sets or abstractions and returns a merged set. This routine has to be defined previously in the C code section.

- **COPIER**: The name of the copying routine (to copy two sets of information) is specified here. This is also required if the **STORE_OPTION** is set to **STORE**. The copier routine has to have the function prototype as shown below:

  ```
  SET_TYPE copy_avail (SET_TYPE set1; SET_TYPE set2);
  ```

  That is, it takes in a set or abstraction and returns a copy of that set. This routine also has to be defined previously in the C code section.

- **PRINTER**: This is the name of the routine that is used to print the information collected. The routine has to have the function prototype as shown below:

  ```
  void print_avail (SET_TYPE set, FILE * filename);
  ```

  That is, it takes in a set and prints the set information in the file given by 'filename'. Default file used is 'stderr'.

- **BUILDCG**: This is an optional routine that could be specified by the user if he/she wants to build the call-graph in his/her own way. Otherwise, the standard built-in routine to build a call graph is called. This option has not yet been implemented.

We now discuss the main part of the specification file. This starts with the name of the main routine which implements the data-flow problem, followed by the list of inputs and output to this routine, and the pattern-action pairs. The grammar for this part of the specification file is specified in Figure 3.4. We start with the keyword **PROCEDURE**, followed by the name of the procedure, and zero or more parameters. The body of the pattern-action pairs starts with the keyword **CASE**, followed by the identifier name which represents the tree-node, followed by **OF** and a list of pattern action pairs. Every pattern could have one or more templates, where each template represents a statement type/expression type found in SIMPLE, and they all can share the same action. For example, we could have

```
<[id1: WHILE cond DO stmt][id2: DO stmt WHILE cond]>
     {
     process_loop(stmt);
     }
```

That is, for both the **WHILE** and the **DO** loops, we want a common action to be executed. Every template which represents a statement/expression type has the following format `[ stmtid : stmt-template ]`.

A sample of the main section for the above available expressions data-flow problem is shown in Figure 3.5.

Thus, the name of the main routine is **avail_expr**, and it takes in the input set **indata** and a pointer to the tree_node, and, depending on the type of tree_node, computes the output set **outdata**. The pattern part of each rule matches a particular statement type in SIMPLE. For example, if you consider the first pattern, it matches a while-loop. The words in capitals like WHILE and DO are the keywords used to distinguish it as a while-loop. The words in small letters are any identifier names which represent the different parts of the while-loop. The generator maps these identifier names to the actual parts of the SIMPLE tree they represent. In the action part, these identifier names are used to perform an operation on that part of the while loop. Other than this, in the action, we can use any other variables which we define either in the C-code header or as a parameter, or defined within the action part itself. For example, in the rule for a 'while-loop', the action part uses some variables like 'next_approx' which is defined right there, 'indata' which comes in as a parameter, and 'stmt' which is a used in the pattern to represent the statement part of the 'while loop'.

3.2.2 Output of McTAG

The C code output of the generator tool just for the **if-** statement rule is given in Figure 3.6.

The generator works in two phases. Since SIMPLE has a fixed number of statement/expression types, the generator has a table consisting of one entry for every statement/expression type found in SIMPLE. In the first phase, as the generator parses the specification file, it fills up the table entries with the variable names used

```
routine: PROCEDURE identifier '(' paramlist ')' ':' type body
paramlist: param | paramlist ',' param |
param  : identifier ':' type
type   : identifier
body   : CASE identifier OF cases
cases  : case | cases case
case   : '<' caseheadlist '>' C_CODE
caseheadlist : caseheadlist casehead | casehead
casehead : '[' identifier ':' stmttype ']'
stmttype : WHILE identifier DO identifier
       | DO identifier WHILE identifier
       | FOR identifier identifier identifier DO identifier
       | RETURN
       | BREAK
       | CONTINUE
       | BREAK identifier
       | RETURN identifier
       | SWITCH identifier DO identifier
       | CASE identifier DO identifier
       | DEFAULT DO identifier
       | IF identifier THEN identifier ELSE identifier
       | IF identifier THEN identifier
       | CALL '(' identifier ',' identifier ')'
       | identifier '=' identifier identifier identifier
       | '*' identifier '=' identifier identifier identifier
       | identifier '=' identifier
       | identifier '=' '*' identifier
       | identifier '=' '&' identifier
       | identifier '=' identifier  identifier
       | identifier '=' CALL '(' identifier ',' identifier ')'
       | identifier '=' CAST '(' identifier ',' identifier ')'
       | '*' identifier '=' identifier
       | '*' identifier '=' '*' identifier
       | '*' identifier '=' '&' identifier
       | '*' identifier '=' identifier  identifier
       | '*' identifier '=' CALL '(' identifier ',' identifier ')'
       | '*' identifier '=' CAST '(' identifier ',' identifier ')'
       | identifier ';' identifier
       | identifier identifier identifier
       | identifier
       | DEFAULTACTION
```

Figure 3.4: Grammar for the Specification File

```
PROCEDURE avail_expr (node:tree, indata:SET_TYPE) :SET_TYPE
CASE node OF
<[stmtid: WHILE cond DO stmt]> {
    SET_TYPE next_approx,out2;
    SET_TYPE out1,last_approx;

    next_approx = avail_expr(cond,indata);
    do {
       last_approx = next_approx;
       out1 = avail_expr(stmt,last_approx);
       out2 = avail_expr(cond,out1);
       next_approx = merge_avail(last_approx,out2);
    } while (next_approx !=last_approx);
    return next_approx;
}
<[stmtid: IF cond THEN thenpart ELSE elsepart]> {
    SET_TYPE out1, out2;
    out1 = avail_expr(thenpart, indata);
    out2 = avail_expr(elsepart, indata);
    out = merge_avail(out1, out2);
    return out;
}
<[stmtid: var1 = val1 binop val2]> {
    indata = delete_expr(get_exprs(var1), indata);
    indata = add_expr(create_expr(binop,val1,val2), indata);
    return indata;
}
<[stmtid: var1 = unop val ]>  {
    indata = delete_expr(get_exprs(var1), indata);
    indata = add_expr(create_expr(unop, val), indata);
    return indata;
}
<[stmtid: stmt1 ; stmt2]> {
    SET_TYPE out1,out2;
    out1 = avail_expr(stmt1, indata);
    out2 = avail_expr(stmt2, out1),
    return out2;
}
```

Figure 3.5: A Sample Specification File for the Analyzer Generator

```
case IF_STMT: {

    /** extra code generated to declare variables used in the **/
    /** pattern part of the specification                     **/
    tree   fi   =   node  ;
    tree   cond   = STMT_COND(  node  );
    tree   thenpart   = STMT_THEN( node  );
    tree   elsepart   = STMT_ELSE(  node  );

    if (STMT_ELSE(node) != NULL) {

    /** Extra Code generated to store the output in the     **/
    /** tree, because the STORE option is specified         **/
        if ((TREE_O_INFO(node))[O_TREE_FIELD_INDEX] == NULL)
             (TREE_O_INFO(node))[O_TREE_FIELD_INDEX] =
                 copy_avail(indata);
        else
           (TREE_O_INFO(node))[O_TREE_FIELD_INDEX] =
              merge_avail(copy_avail(indata),
                  (TREE_O_INFO(node))[O_TREE_FIELD_INDEX]);

/** code that is a part of the specification input **/
        {
        O_SET_TYPE out1, out2,out;
        out1 = avail_expr(thenpart, indata);
        out2 = avail_expr(elsepart, indata);
        out = merge_avail(out1, out2);
        return out;
        }
    }
}
```

Figure 3.6: A part of the Generator Output

in the templates and the actions corresponding to every statement/expression type. In the second phase, it makes one full pass over the table and spits out C-code.

The generator has built-in information about the structure of SIMPLE. In order to create the data-flow analyzer routine, additional code has to be added to traverse and access the right sub-parts of the SIMPLE tree. The generator defines and initializes the variable names used in the pattern-templates (which it has stored in the table), with the actual parts of the SIMPLE tree they represent. For example, in the case of the `if`-statement, the variables 'thenpart' and 'elsepart' used in the patterns are initialized to be the tree representing the then-part of the `if`-statement and else-part of the `if`-statement respectively.

Since the **STOREOPTION** is specified to be **STORE** in the specification input, additional code has to be generated to store the collected information in the tree. The TREE_O_INFO field has a pointer to the information stored. **TREE_AVAIL_INDEX** is the unique index of the information array which points to the list of available expressions. This is described in more detail in the next section. Notice that the actions performed are different if the tree already has some information stored in it. If the information in the tree is NULL, which means that this is the first time we are analyzing that subtree, we just store the new information. Otherwise, (this means we are in a loop or in a recursive procedure call where we are computing the fixed point), we merge this new information with the information already present in the tree.

The C code given in the action part of the rules is inserted as it is in the output. If a rule for a particular type of statement is not given, then the table entry corresponding to that pattern-template is empty and the default action specified in the input specification is used. However, if there is no default action in the specification file, the identity function is performed, i.e., the data-set which comes in as a parameter is returned as the output.

The entire output analyzer produced by the generator for the specification input for reaching definitions is shown in Appendix C.

3.2.3 Storing the Data-Flow Information in the Tree-Node

As indicated in Figure 3.1, each analysis is specified independently, and a unique index is included in each specification which is used to specify a unique position in

a tree-node for storing the appropriate computed values (refer to Figure 3.7). This provides a way of ensuring that each analysis has an appropriate place for storing its results. There is a field in all relevant tree-nodes that points to an array of pointers. Each element of this array points to the information computed by different data-flow analyzers. The array-index for a particular data-flow analysis is specified in the input specification file. Similarly, in the case of interprocedural analysis, we store the additional information in the call-graph.

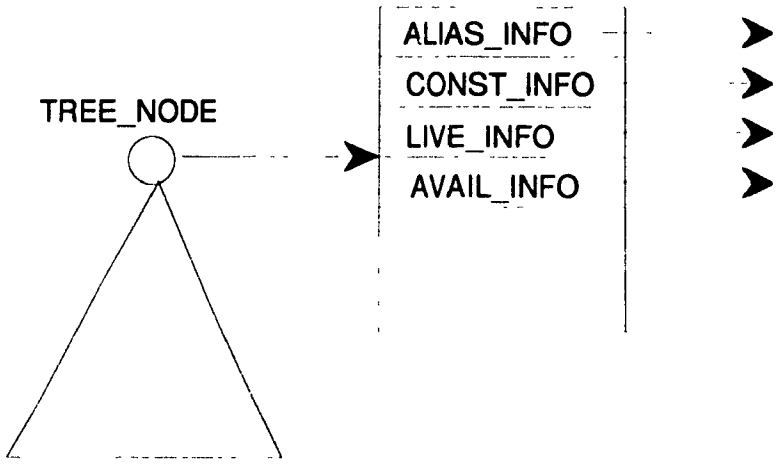


Figure 3.7: Data-flow information stored in a Tree-node

3.3 Parallelizing the Data-Flow Analyzers

In generating data-flow analyzers in a structured manner, parallelism can be exploited at two levels: one between various data-flow analyzers and one inside a specific data-flow analyzer.

- As shown in Figure 3.1, a number of data-flow analyzers could be produced using the generator tool. The places to store the results of these analyses are unique to the particular analysis. This provides an opportunity to exploit the coarse grain parallelism existing among these analyses. However, certain analyses have some kind of dependency with some other analyses and therefore enforce an order in the execution of these analyses. For example, alias information needs to be computed before computing live variables or reaching definitions (refer to

Chapter 4). Thus, except for the partial order enforced by the dependencies among analyses, parallelism at a coarse grain level can be exploited by executing the analyzers in parallel.

- Since we are building structured data-flow analyzers to work on SIMPLE, which itself is a structured compositional representation of the program, it is guaranteed that different subtrees will not interfere. As a consequence, they could be analyzed in parallel. For example, the two conditional parts of an if-statement could be analyzed in parallel. Similarly, all branches of a **`switch`**-statement could be analyzed in parallel.

3.4 Summary

In this chapter, we have presented a tool which helps us build structured, sophisticated analyzers in a relatively straightforward and elegant manner. The framework of the McTAG is based on SIMPLE, which is a structured and compositional representation of the entire program.

Chapter 4

Introductory Examples

In this chapter, we describe two analyzers that have been generated using the analyzer tool discussed in Chapter 3. The first analysis presented is a forward analysis for reaching definitions while the second analysis is a backward one for live variables. For each of the analyses, we first define the problem, the abstraction used, the operations performed on the abstraction, and then describe how the data-flow analysis problem is solved using the analyzer tool. Further, we extend the analyses to handle the `break` and `continue` constructs in a structured manner. Finally, in order to make our analyses interprocedural, we construct a special call graph that is employed to analyze both ordinary and recursive procedure calls.

4.1 Reaching Definitions

Reaching definition analysis helps us to determine what definitions **reach** a particular point in a program. This is a forward analysis, as discussed in [ASU86], where we analyze a program from top to bottom.

4.1.1 Problem Definition

Reaching definitions have been formally defined in [ASU86] as: A **definition of a variable x** is a statement that assigns, or may assign a value to **x**; A definition d

reaches a point p, if there is an execution path immediately following d to p, such that d is not 'killed' along that path.

For the purposes of our analysis, we extend this to define *definitely-reaching* and *maybe-reaching*. A definition d of a variable x is said to *definitely-reach* a point p if d is the only definition of the variable x that reaches the point p. The definition d of a variable x is said to *maybe-reach* the point p if there is more than one definition of the variable x that reaches point p. The following piece of code illustrates the above definitions.

```
(i)    y = 3;
        <-------- Definition (i) of y definitely-reaches.     (A)
(ii)   x = 4;
        <-------- Definition (i) of y definitely-reaches,
                  Definition (ii) of x definitely-reaches.   (B)
       if (cond)
(iii)     x = 2;
        <-------- After the if-statement,
                  the definition (i) of y definitely-reaches,
                  definition (ii) of x maybe-reaches,
                  definition (iii) of x  maybe-reaches.      (C)
```

We extend this definition to include structure references. Rather than computing the reaching definitions of structure references by enumerating all the subfields in the structure, we use the following conservative approximation. The definition of a structure s *definitely-reaches* a point p in the program if the definition of all the subfields of s definitely-reaches that point p in the program. If the definition of any field of s is redefined on any path starting at point p in the program, the definition of s *maybe-reaches* the point p.

We illustrate this with examples.

Example 1

```
(i)  a.b.c = ...
           <-- a.b.c defined in (i) definitely-reaches here    (A)
(ii) a.b = ...
           <-- a.b defined in (ii) definitely-reaches here     (B)
```

As we analyze the program from top to bottom, we first encounter statement (i). The definition (i) of `a.b.c` reaches point (A). In (ii), we are defining a larger part of the same structure, therefore, only (ii), defining `a.b`, definitely-reaches point (B).

Example 2:

```
(i)   a.b = ...
            <-- a.b defined in (i) definitely-reaches here    (A)
(ii)  a.b.c = ...
            <-- a.b defined in (i) maybe-reaches, and that defined
                in (ii) definitely-reaches                    (B)
```

The complete definition of all the subfields of the structure `a.b` defined at (i) does not reach point (B), because definition (ii) kills a part of it. Therefore we say (i) maybe-reaches point (B), but (ii) definitely-reaches (B).

This is where we make the choice between being as accurate as possible and being time and space efficient. We could have analyzed and could have been more accurate regarding which part of the definition (i) of `a.b` definitely-reaches (B). The following example brings out the difference between the two approaches.

```
struct {
  int a;
  int b;
} c, t;

  c = t;      --- (i)
  c.a = 1;    --- (ii)
  c.b = 2;    --- (iii)
     <------------ The definition (i) of c does not reach here,
                   because the structure c is completely defined
                   by (ii) and (iii). But our conservative analysis
                   will say definition (i) of c maybe-reaches.
```

Example 3.

```
(i)   t.b  = ...
           <-- t.b defined in (i) definitely-reaches here    (A)
(ii)  t.c  = ...
           <-- t defined in (i) maybe-reaches, and that defined
               in (ii) definitely-reaches                    (B)
```

This is also a conservative approximation, because we say definition (i) maybe-reaches point (B). This is a safe approximation.

Example 4.

```
if (cond)
   x.a = ... (i)
else
   x.b = ... (ii)
           <-- Here, x defined in (i) maybe-reaches, and that
               defined in (ii) also maybe-reaches.           (A)
```

Since definitions (i) and (ii) are on either branches of an `if`-statement, they both maybe-reach point (A).

However, in the case of pointers, we make use of alias information to be as accurate as possible. Consider the following example:

```
*t = 3;      (i)
     <----- (A)
```

When we reach the statement (i), we look at all the aliases of `*t`. If `*t` is aliased to a single scalar variable, say `a`, we say the definition (i) of `a` definitely-reaches (A). If `*t` is aliased to more that one scalar variable (these are 'maybe' aliases), say `a` and `b`, then we say that definition (i) of both `a` and `b` maybe-reach (A). If `*t` is aliased to structure references, then they have to be dealt with in a similar fashion. Lastly, if we have the reference `(*t).b.c`, we get the aliases of `*t`, say `a`, and then deal with the reference `a.b.c` in an appropriate manner.

4.1.2 Data Structure Abstraction

The data structure abstraction used to capture the information about reaching definitions is a set of `items`, where each `item` is of the form (`varname, list`), where `list` is a list of `definition pairs` of the form (`definition, flag`), and `flag` is either `may-reach` or `def-reach`. Thus at every point in the program we associate a set of `items` containing all the definitions that may or definitely-reach that point.

4.1.3 Operations on the Abstraction

We define the following terms: The *base-name* of a `varname` (defined in Figure 2.6) is defined as:

1. The base-name of a scalar variable of the form `x` is `x`.
2. The base-name of a structure reference of the form `a.b.c....` is `a`.
3. The base-name of a structure-pointer reference of the form `(*t).b.c...` is `t`.
4. The base-name of an array reference of the form `a[i][j]...` is `a`.

The *common-prefix* of two `varnames` is the same base-name as the largest prefix of these `varnames`. For example, the common prefix of two scalar variables `x` and `x` is `x`, of `a.b.c` and `a.b.d` is `a.b`, of `(*t).y.z` and `(*t).y` is `(*t).y`. We treat an entire array as a single scalar element: the common-prefix of two array references of the form `a[i][j]` and `a[i][j]` is just `a`, the common base-name of the two array references.

We define the function `common_prefix (a,b)` to return the common-prefix of two `varnames` having the same base-name. The merge operation on two sets of `items` (defined above) is defined in Figure 4.1. The routine `merge_reach` calls function `merge_item` to merge two `items` with the same `base-name`. The operation `concat_lists` concatenates two lists of definition pairs together. If the same definition is present in both the lists, it comes out as it is; if a definition appears on only one of the lists, it is always converted to a maybe-reach. The routine `change_flag` changes the flag of all the elements in a list to maybe-reach.

```
merge_item (⟨var1, list1⟩,⟨var2, list2⟩)=⟨prefix(var1,var2), concat_lists
(list1, list2)⟩

concat_lists(listA.listB):listC
{
   listC = {};
   For all pairs (def, flag) of listA do {
       if (pair (def, flag) exists in listB) {
           listA = listA - {(def, flag)};
   listB = listB - {(def, flag)};
           listC = listC U {(def, flag)};
       }
   }
   For all remaining pairs (def, flag) of listA and listB do
       listC = listC U {(def, maybe-reach)};

   return listC;
}
merge_reach(setA, setB) : setC
{
    setC = {};
    For all items itemA of the form <nameA, listA> of setA do {
       /** get corresponding item from setB and merge them          **/
       itemB  = get_item_with_same_base_name(itemA, setB);
       if (itemB != NULL) {
           setC = setC U {merge_item (itemA, itemB)};
           setB = setB - {itemB};      /** Delete itemB from setB  **/
       }
       else {
       /** setB does not have a item with the same basename as itemA **/
           setC = setC U {<nameA, change_flag(listA, maybe_reach)>};
       }
    }
    For all remaining items itemB of the form <nameB, listB> of setB do {
        setC = setC U {<nameB, change_flag(listB, maybe-reach)>};
    }
    return setC;
}
```

Figure 1.1: Merge Operation for Reaching Definitions

4.1.4 Specification to the Generator Tool to Create the Analyzer Module

McTAG, described in Chapter 3 takes a set of specifications describing the data-flow problem and creates an analyzer module that operates on SIMPLE. The main part of the input to the tool is a series of pattern-action pairs for every kind of statement type found in SIMPLE. We list the pattern-action pairs for some illustrative statement types for computing reaching definitions. This example also illustrates how structured flow analyses are implemented in general. We start with writing the rules for basic statement types, and then move on to sequences and other control constructs.

```
PROCEDURE reach_def(node:tree, indata:SET_TYPE) :SET_TYPE
CASE node OF
<[stmtid: val = var1 op var2]> {
   indata = delete_def (indata, val);
   indata = add_def (indata, stmtid, val, DEFINITELY_REACH);
   return indata;
}
```

This describes a pattern-action pair for a basic statement. We first delete the previous definitions of `val` from the input set, and add this new definition to it.

Let us now consider a rule for a slightly more complex statement, one with aliases in it.

```
<[stmtid: *val = var1 op var2 ]> {
   /** The routine 'get_alias' gets all the variables that are  **/
   /** aliases of '*val'                                        **/
   aliases = get_alias(val);
   indata = update_def (indata, stmtid, aliases);
   return indata;
}
```

The routine **update_def** does the job of the **delete_def** and **add_def**. It is more complicated because aliases are involved. It first checks if `val` is aliased to a single scalar variable. If it is, the previous definitions of this scalar variable can be deleted,

and the new definition can be added. If it is aliased to more than one scalar, then the new definition is added as a maybe reaching definition for each scalar to which it was aliased. Similarly, it has to deal with the cases when **val** is aliased to structures.

Next, let us consider a sequence of statements.

```
<[stmtid: stmt1; stmt2 ]> {
   SET_TYPE outdata1, outdata2;
   outdata1 = reach_def (stmt1, indata);
   outdata2 = reach_def (stmt2, outdata1);
   return outdata2;
}
```

Since this is a forward analysis, **stmt1** has to be analyzed first before **stmt2**. So we call the routine **reach_def** recursively first on **stmt1**, and then on **stmt2**. Notice that the input information for **stmt2** is the output information that is collected after analyzing **stmt1**.

We shall now illustrate the rules for an **if**-statement.

```
<[IF cond THEN thenpart ELSE elsepart ]> {
   SET_TYPE out1, out2;

   indata = reach_def(cond, indata);
   out1 = reach_def(thenpart, indata);
   out2 = reach_def(elsepart, indata);
   outdata = merge_reach (out1, out2);
   return outdata;
}
```

We first analyze the **cond** part of the **if**-statement. Notice that for reaching definitions, analyzing the **cond** part is not really necessary since all conditional expressions in SIMPLE are reduced to simple scalar variables and cannot have any side-effects. Therefore, after this step, **indata** is unchanged. We analyze the **thenpart** and the **elsepart** of the **if**-statement in turn. Notice that the input information to both of these is the information that enters the **if**-statement. After this, we merge the information obtained from both the branches of the **if**-statement.

In the above actions, we use a number of routines like `merge_reach` and `update_def` which should be defined in the C-code section of the specification file.

Switches are analyzed in a similar manner. Just as we have a two-way branch and a merge in an `if`-statement, we have an n-way branch and merge in a `switch`-statement.

We shall now demonstrate how `loops` are analyzed in a structured manner. We first look at loops that do not contain `break`, `continue` or `return` statements. Loops with `breaks`, `continues` or `returns` are dealt with in Section 4.3.

Consider the `while`-loop depicted in Figure 4.2. The code to analyze the `while`-loop is also listed in Fig 4.2. The `while`-loop body, depending on the condition `cond`, could be executed `n` times, where `n = 0, 1, 2, ...` This is what we are approximating when we analyze the loop. That is, we are approximating the information at the point * in the Figure 4.2. The first approximation of our output information is what we have after analyzing only `cond`. With this first approximation, we analyze `stmt`, the body of the `while`-loop, and `cond`. We now obtain our second approximation, and we continue till the last approximation equals the new approximation computed, i.e., a fixed point is reached.

```
<[l1: WHILE cond DO stmt]> {
   SET_TYPE next_approx,out2;
   SET_TYPE out1,last_approx;

   next_approx = reach_def(cond,indata);
   do {
      last_approx = next_approx;
      out1 = reach_def(stmt,last_approx);
      out2 = reach_def(cond,out1);
      next_approx = merge_reach(last_approx,out2);
   } while (next_approx !=last_approx);
   return next_approx;
}
```

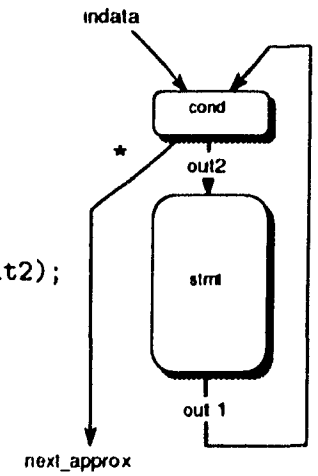


Figure 4.2: Forward Analysis of a WHILE Loop

A similar fixed-point computation is performed for **do-loop** and for **for-loops**. Do-loops are illustrated in Figure 4.3. We now approximate the information at the point * in the figure. Since the **do-loop** will be executed at least once, the first approximation to the **do-loop** is what we have after analyzing **stmt** and **cond** once. After this, we do a fixed-point calculation similar to that of the **while-loop**.

```
<[11: DO stmt WHILE cond]>
{
   SET_TYPE next_approx,out2;
   SET_TYPE out1,last_approx;

   out1 = reach_def(stmt,indata);
   next_approx = reach_def(cond,out1);
   do {
      last_approx = next_approx;
      out1 = reach_def(stmt,last_approx);
      out2 = reach_def(cond,out1);
      next_approx = merge_reach(last_approx,out2);
   } while (next_approx !=last_approx);
   return next_approx;
}
```

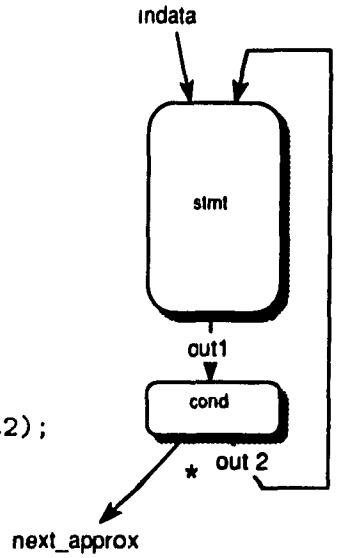


Figure 4.3: Forward Analysis of a DO Loop

For-loops are handled in a similar manner, except that the **start**, **end** and iteration conditions have to be handled carefully.

Notice that there are a number of ways to compute the fixed points of loops. We have presented one very general method. McTAG is flexible and general enough to allow the encoding of any of the methods to compute fixed points

We shall deal with procedure calls in detail in Section 4.4.

4.2 Live-Variable Analysis

Live variable analysis helps us to determine the live-ranges of variables used in the program. This is a backward analysis (the program is analyzed from bottom to top) and is absolutely essential for optimizations like register allocation. Our approach

here is similar to that of reaching definitions; the main difference is that this is a backward analysis.

4.2.1 Problem Definition

A variable **x** is *live* at a point **p** in the program if the value of **x** at **p** could be used along some execution path in the program starting at point **p** [ASU86]. Thus, during live-variable analysis, at every point in the program, we compute a set of variables that are live at that point in the program.

We now extend this notion of liveness to *definitely-live* and *maybe-live*. We first define these for scalar variables and later extend them for structure and pointer references. A scalar variable **x** is said to be definitely-live at a point **p** in the program if the value of **x** is definitely used along every execution path in the program starting at point **p**. Consider the following piece of code.

```
     <-- nothing is live                          (C)
x = 3;
     <-- x is definitely-live                     (B)
z = 4;
     <-- x is definitely-live and z is maybe-live (A)
if (cond)
   y = x + z;
else
   y = x;
```

As we do a backward analysis of the program, after analyzing the `if`-statement, at point (A) in the program, we find **x** is definitely-live and **z** is maybe-live. This is because x is used on both the branches of the `if`-statement, whereas **z** is used only in one branch. At point (B) in the program, only **x** is definitely-live, (**z** is 'killed' because it is defined), and at point (C) in the program, nothing is live.

We now extend the above definitions for structure, array and pointer references. We define a structure **s** to be *definitely-live* at a point **p** in the program if the values of all the subfields of **s** are definitely used along every execution path in the program starting at point **p**. Otherwise, if at least one field of structure **s** is used along some

execution path starting at point p in the program, the structure s is said to be *maybe-live*. We illustrate the above by some examples.

Example 1.

```
                <---nothing is live              (B)
(ii)    a = ...
                <---a.b is definitely-live       (A)
(i)   ... = a.b
```

As we analyze the program backwards, we encounter the statement (i). Since this is an use of a.b, at point (A), the structure a.b is definitely live. Statement (ii) completely defines the structure a. Thus, at point (B), nothing is live.

Example 2.

```
                <-- a.b is definitely-live       (B)
(ii)    ...= a b
                <--a.b.c is definitely-live      (A)
(i)     ..= a.b.c
```

Here, since statement (ii) uses a superset of the structure used in statement (i), we say the superset structure is live, i.e . a.b is live.

Example 3.

```
              <--a.b is definitely-live          (B)
(ii)    ...= a.b.c
              <--a.b is definitely-live          (A)
(i)     ...= a.b
```

Since statement (ii) uses only a part of the structure used in statement (i), we say a b is definitely-live at point (B) as well.

Example 4.

```
                <--a.b is maybe-live          (B)
(ii)    a.b.c = ...
                <--a.b is definitely-live     (A)
(i)     ... = a.b
```

At point (A), `a.b` is definitely live. However, at point (B), since a part of the structure is defined, we say `a.b` is maybe-live, since we are not sure, at that point, if the structure is completely defined or not.

Once again we make a compromise between being as accurate as possible, and being time and space efficient. This is illustrated in following example.

Example 4a.

```
struct {
int b;
int c;
} a, t;
    <--- At this point, nothing is live, but our conservative analysis
         will say that a is maybe-live.
a.b = 1;
a.c = 2;
t = a;
```

In the case of arrays, we treat each array as one object. Array `a` is *definitely-live* at a point `p` in the program if the value of at least one element of `a` is definitely used along every execution path in the program starting at point `p`. Similarly, an array `a` is defined to be *maybe-live* at a point `p` in the program if the value of at least one element of `a` is used on some execution path in the program starting at point `p`

For pointers, we make uses of aliasing information to determine liveness accurately Consider the following example:

```
      <--a is definitely-live
... = *t;   (i)
```

When we reach the statement (i), we look at all the aliases of `*t`. If `*t` is aliased to a single scalar variable, say `a`, we say both `t` and `a` are definitely-live at that point. If `*t` is aliased to more that one scalar variable (these are 'maybe' aliases), say `a` and `b`, then we say that `a` and `b` are maybe-live and `t` is definitely-live. If `*t` is aliased to structure references, they have to be dealt with in a similar fashion. Similarly, if we have the reference `(*t).b.c` we get the aliases of `*t`, say `a`, and then deal with the reference `a.b.c` in an appropriate manner.

4.2.2 Data Structure Abstraction

We now define the data structure abstraction used to capture the information about live variables. The representation used is a set of tuples of the form ⟨**name, flag**⟩, where `name` represents the variable reference, and `flag` is either definitely-live or maybe live. Thus, at every point in the program we associate a set of tuples containing all the variables that are either definitely-live or maybe-live at that point.

4.2.3 Operations on Sets

The algorithm for merging two sets and two tuples is given in Figure 4.4. The routine `merge_tuple` merges two tuples having the same basename. It is called by the `merge set`, to merge corresponding tuples. The routine `merge_set` goes through all the elements of the two sets and merges the corresponding elements one by one.

4.2.4 Specification to the Generator Tool to Create the Analyzer Module

This is very similar to what is described in Section 4.1.4. The main difference, of course, is that this is a backward analyses. This difference is especially seen when we analyze sequences and loop constructs, which is illustrated below.

```
/** merging of two tuples                                              **/
merge_tuple(tupA, tupB) : tupC
{
    nameA = extract_name_from_tuple(tupA);
    nameB = extract_name_from_tuple(tupB);
    flagA = extract_flag_from_tuple(tupA);
    flagB = extract_flag_from_tuple(tupB);
    if ((flagA == definitely_live) && (flagB == definitely_live)) {
       /** same_name is a routine which returns 1 if both nameA and   **/
       /** nameB are exactly the same, e.g., a.b and a.b              **/
       if (same_name(nameA,nameB))
          tupC = <nameA, definitely_live>;
       else
          tupC = <common_prefix(nameA,nameB), maybe_live>;
    } else
       tupC = <common_prefix(nameA,nameB), maybe_live>;
}
/** merging of two sets of live variables                              **/
merge_set(setA, setB) : setC
{
    setC = {};
    For all tuples tupA of setA do {
       /** get corresponding tuple from setB and merge them           **/
       tupB  = get_tuple_with_same_base_name(tupA, setB);
       if (tupB != NULL) {
           setC = setC U {merge_tuple (tupA, tupB)};
           setB = setB - {tupB};      /** Delete tupB from setB       **/
       }
       else {
       /** setB does not have a tuple with the same basename as tupA **/
           name = extract_name_from_tuple(tupA);
           setC = setC U {<name, maybe-live>};
       }
    }
    For all remaining tuples tupB of setB do {
        name = extract_name_from_tuple(tupB);
        setC = setC U {<name, maybe-live>};
    }
    return setC;
}
```

Figure 4.4: Merge Operation for Live Variable Analysis

```
PROCEDURE live_anal(node:tree, indata:SET_TYPE) :SET_TYPE
CASE node OF
<[stmtid: stmt1; stmt2 ]>
   {
   SET_TYPE  out1, out2;

   out1 = live_anal(stmt2, indata);
   out2 = live_anal(stmt1, out1);
   return out2;
   }
```

Notice that here, unlike in reaching definitions, we analyze **stmt2** first, and then analyze **stmt1**. This ordering is what that drives the backward analysis.

We illustrate the backward analysis of **while**-loops and **do**-loops below.

Figure 1.5 illustrates the **while**-loop during backward analysis. Notice here that the direction of all the arrows have been reversed when compared to Figure 4.2. Again, we try to approximate the information at point * in the figure. The first approximation we use here is the output we get after analyzing the conditional part, cond. Figure 4.6 illustrates the **do**-loop which is analyzed in a similar fashion.

```
<[11: WHILE cond DO stmt]>
{
   SET_TYPE next_approx,out2;
   SET_TYPE out1,last_approx;

   next_approx = live_anal(cond,indata);
   do {
      last_approx = next_approx;
      out1 = live_anal(stmt,last_approx);
      out2 = live_anal(cond,out1);
      next_approx = merge(last_approx,out2);
   } while (next_approx !=last_approx);
   return next_approx;
}
```

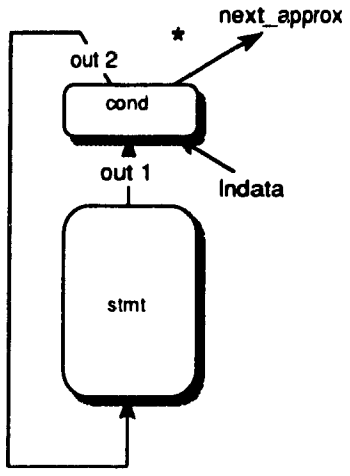


Figure 1.5: Backward Analysis of a WHILE Loop

We shall discuss procedure and function calls in detail separately in Section 4.4.

```
<[l1: DO stmt WHILE cond]>
{
   SET_TYPE next_approx,out2;
   SET_TYPE out1,last_approx;

   out1 = live_anal(cond,indata);
   next_approx = live_anal(stmt,out1);
   do {
      last_approx = next_approx;
      out1 = live_anal(cond,last_approx);
      out2 = live_anal(stmt,out1);
      next_approx = merge(last_approx,out2);
   } while (next_approx !=last_approx);
   return next_approx;
}
```

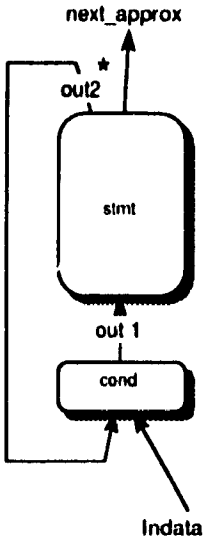


Figure 4.6: Backward Analysis of a DO Loop

4.3 Analyzing Break and Continue Constructs

This section describes how we analyze `break` and `continue` constructs of C in a structured manner. We shall extend both our forward and backward analysis techniques to handle `break` and `continue` constructs. We shall do this by extending our abstraction to include not only the information set, but also information about the data at the `break` and `continue` points of the program.

Forward Analysis:

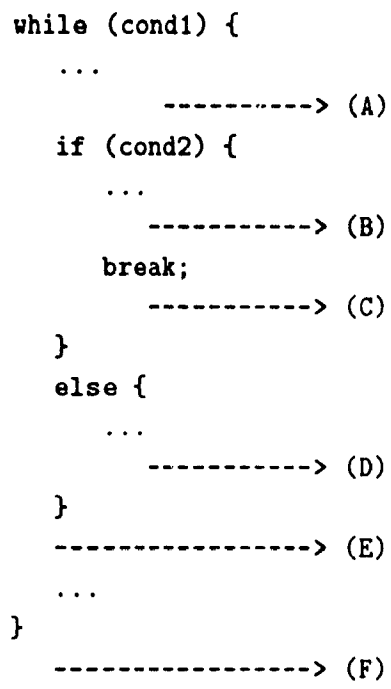
We shall first discuss forward analysis of a `while`-loop. As shown in Figure 4.7, when we encounter a `break` in the body of a loop, the control goes to the end of the loop. Hence, one of the valid outputs of the `while`-loop is the information collected just before the `break` statement. Similarly, a `continue` statement transfers control to the beginning of the loop. In this case, one of the valid inputs to the `while`-loop is the data collected just before the `continue` statement.

We shall therefore extend our abstraction to include two lists, a `breaklist` and a `contlist`. The code to analyze `while`-statements with `breaks` and `continues` is given in Figure 4.7. When we reach a `break` or a `continue`, we store the information

collected up to that program point in the **breaklist** or the **contlist** respectively, and return a new entity which we call as BOTTOM. BOTTOM essentially means that a path after the **break** or the **continue** is invalid a statement immediately following a **break** is never reached. Therefore, the output after analyzing any statement with BOTTOM as input is BOTTOM. But if we merge any set with BOTTOM, the result is the set.

After the first pass over the body of the loop, we have collected some information which is an approximation. We merge this with all the information sets present in the **contlist** and is used as an input to obtain the next approximation. We continue this way until we reach a fixed point. We now merge this fixed point output with all the information sets present in the **breaklist**. This newly merged information is returned as the result of analyzing the **while**-loop.

The following small example with a **break** statement gives an intuition behind how BOTTOM works. At point (A), we have collected some information, which enters both the branches of the **if-else** statement. As we analyze the **then** part of the **if**-statement, at point (B), just before the **break**, we have some valid data. This is stored in the **break-list** and BOTTOM is returned. Thus, the result of analyzing the **then**-part of the **if**-statement at point (C) is BOTTOM. After analyzing the **else**-part of the **if**-statement, at point (D), we get some valid data. This, merged with the output of the **then**-part (i.e.,BOTTOM) results in the valid data. This is used to as input to the statements after the **if**. After analyzing the **while**-loop completely, at point (F), we are going to merge the data collected in the **break-list**; at this point we include the information collected at point (C), which is what is required because after a **break** control goes to the end of the **while**-loop.



Backward Analysis:

In a backward analysis, we analyze a program from the bottom to the top. As shown in Figure 1.9, when we perform a backward analysis, **break** points act as entry points to the loop. At this point, the data coming in will be the information collected just outside the bottom of the loop. Therefore, the rule for the **break**-statement needs to be different for this case. Similarly, since `continue` transfers control to the beginning of the loop, the input information at the `continue` points will be what is collected each time at the top of the loop.

In a forward analysis, we extend the abstraction to hold data collected at **break** and `continue` points. Here, in a backward analysis, we extend the abstraction to store the input for the **break** and `continue` points in the **break**- and `continue`-lists. The code for a backward analysis of these statements is given in Figure 4.9.

Handling Return Statements:

These are handled in a manner similar to **break** and `continue` statements, by extending the abstraction and by having a `return-list`. The pieces of information collected

```
<[l1: WHILE cond DO stmt]> {
  blist = NULL;        /** break list **/
  next_app = reach(cond,indata);
  do {
    clist = NULL;        /** continue list **/
    last_app = next_app;
    out1 = reach(stmt,&blist,&clist,last_app);
    /** merge continue list **/
    out1 = merge_reach(out1, clist);
    out2 = reach(cond, &blist, &clist, out1);
    next_app = merge_reach (out2, last_app);
  } while (next_app!= last_app);
   /* merge break list */
  next_app = merge_reach(next_app, blist);
  return next_app;
}

<[l1: BREAK]> {
  store_data_in_break_list(indata);
  return BOTTOM;
}

<[l1: CONTINUE]> {
  store_data_in_cont_list(indata);
  return BOTTOM;
}
```

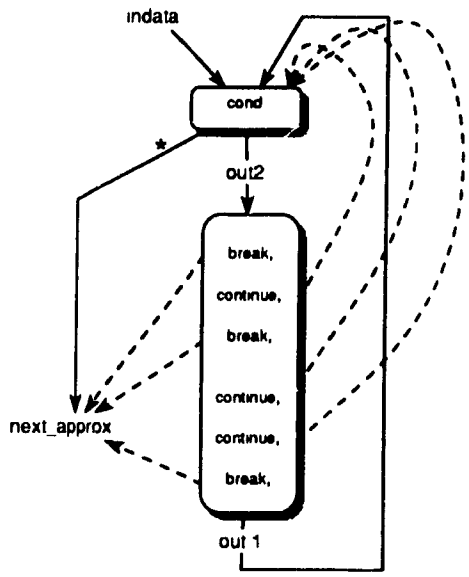


Figure 1.7: Breaks and Continues in WHILE Loops: Forward Analysis

```
<[l1: DO stmt WHILE cond]> {
  blist = NULL;          /** break list **/
  clist = NULL;          /** continue list **/
  out1 = reach(stmt, &blist, &clist, indata);
  next_app = reach(cond, &blist, &clist, out1);
  /** merge continue list **/
  next_app = merge(next_app, clist);
  do {
     last_app = next_app;
     out1 = reach(stmt,&blist,&clist,last_app);
     out2 = reach(cond,&blist,&clist, out1);
     next_app = merge_reach(last_app,out2);
     /** merge continue list **/
     next_app = merge(next_app, clist);
  } while (next_app!=last_app);
  /* merge break list */
  next_app = merge(next_app, blist);
  return next_app;
}

<[l1: BREAK]> {
  store_data_in_break_list(indata);
  return BOTTOM;
}

<[l1: CONTINUE]> {
  store_data_in_cont_list(indata);
  return BOTTOM;
}
```

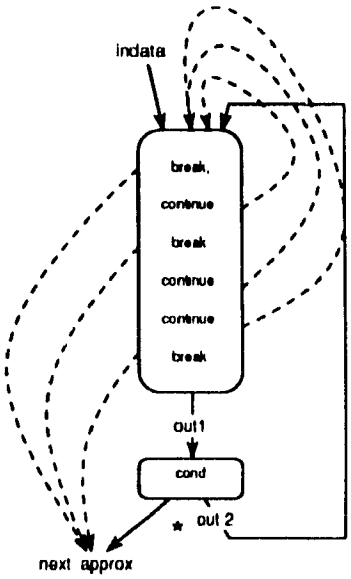


Figure 1.8. Breaks and Continues in DO Loops: Forward Analysis

```
<[11: WHILE cond DO stmt]> {
  /* set input at the break points */
  blist = copy_data(indata);
  next_app = reach(cond,indata);
  /*initial input at continue points is NULL*/
  clist = NULL;        /** continue list **/
  do {
    last_app = next_app;
    out1 = reach(stmt,&blist,&clist,last_app);
    out2 = reach(cond, &blist, &clist, out1);
    next_app = merge (out2, last_app);
    /** update input for continue points **/
    clist = copy_data(next_app);
  } while (next_app!= last_app);
  return next_app;
}

<[11: BREAK]> {
  /* new input at break point */
  indata = get_data_from_break_list(indata);
  return indata;
}

<[11: CONTINUE]> {
  /* new input at continue point */
  indata = get_data_from_cont_list(indata);
  return indata;
}
```

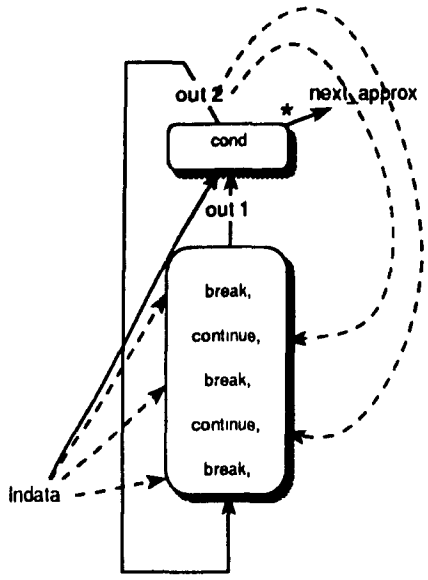


Figure 1.9. Breaks and Continues in WHILE loops: Backward Analysis

```
<[l1: DO stmt WHILE cond]> {
  /* initial input at break and cont. points*/
  blist = copy_data(indata);
  clist = NULL;
  out1 = reach(stmt, &blist, &clist, indata);
  next_app = reach(cond, &blist, &clist, out1);
  /* update continue input */
  clist = copy_data(next_app);
  do {
     last_app = next_app;
     out1 = reach(stmt,&blist,&clist,last_app);
     out2 = reach(cond,&blist,&clist, out1);
     next_app = merge(last_app,out2);
     clist = copy_data(next_app);
  } while (next_app!=last_app);
  return next_app;
}

<[l1: BREAK]> {
  /* new input at break point */
  indata = get_data_from_break_list(indata);
  return indata;
}

<[l1: CONTINUE]> {
  /* new input at continue point */
  indata = get_data_from_cont_list(indata);
  return indata;
}
```

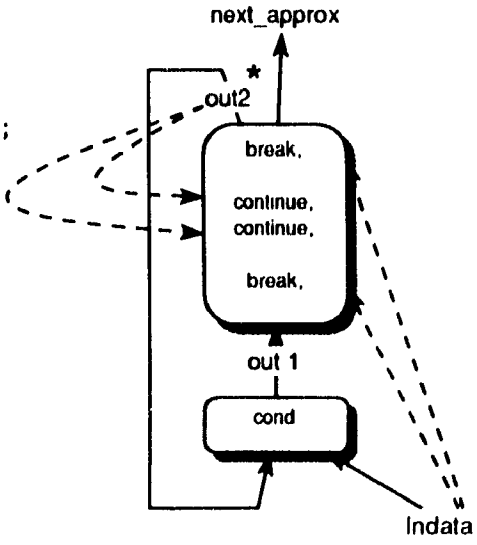


Figure 1.10: Breaks and Continues in DO loops: Backward Analysis

at the various **return** points in a procedure are all stored in the **return-list**. These are merged when we return from the procedure. Refer Figure4.11 for an illustration.

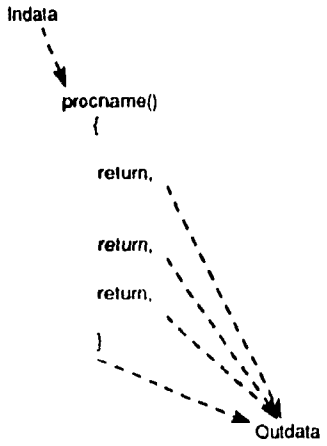


Figure 4.11: Handling Return Statements

4.4 Interprocedural Analysis

In the previous sections, we have not described what exactly we do when we encounter a procedure/function call. This section describes the special call-graph built to perform interprocedural analysis. The generator tool makes use of this call-graph by default when the analysis is specified as interprocedural.

4.4.1 Analysis of Nonrecursive Procedure Calls

In this subsection, we shall describe the call graph and the analysis for nonrecursive procedure calls, and in the next subsection, we shall extend this to handle recursive procedure calls. Figure 4.12 shows a non recursive program and its call graph. The call-graph is made up of nodes, which represent procedure calls, and edges, which specify the calling sequence. Every node in the call-graph (except **main**) corresponds to a calling site in the program. The nodes in this nonrecursive call-graph are called *ordinary nodes* and the edges are called *calling arcs*. Later, when we consider recursive call graphs, we shall introduce other different kinds of nodes and arcs.

```
main()
{
   f();      ---------(A)
   g();      ---------(B)
}

f()
{
   h();      ---------(C)
}

g()
{
   h();      ---------(D)
}

h()
{
}
```

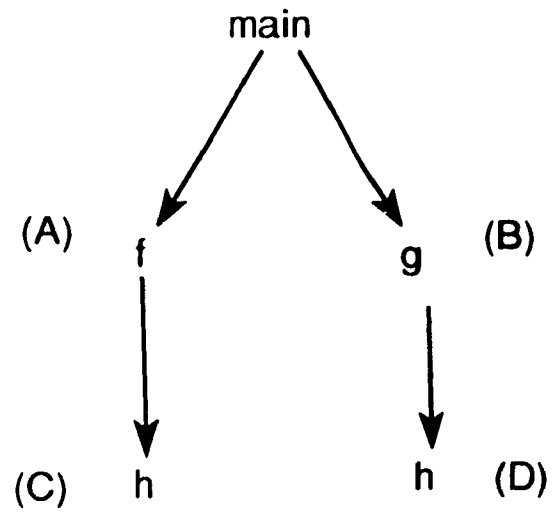


Figure 4.12: Nonrecursive Call Graph

When we do interprocedural analysis, we move back and forth between the call-node and the SIMPLE tree. Figure 4.11 illustrates what we do when we encounter a procedure call. When we encounter a procedure call in the SIMPLE tree, we analyze the arguments and then call the procedure `traverse_cg` with the corresponding `call_node`. `Traverse_cg` first calls a routine called `map` to map the information associated with the actual parameters to the formal parameters. `Map` and `unmap` are routines that map the information associated with actual parameters to the formal parameters, and vice-versa (refer to Figure 4.13). We then go back to SIMPLE to analyze the body of this new procedure. When we return after analyzing the procedure, we perform the unmapping of information

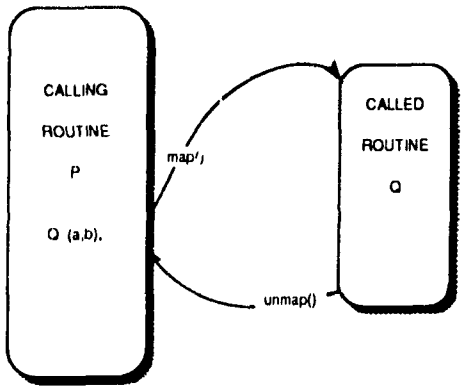


Figure 4.13: Map and Unmap Routines

In the next section, we shall see how the call-graph is built for recursive procedures, and the extensions made to the routine `traverse_cg` to traverse this new call-graph.

4.4.2 Analysis of Recursive Procedure Calls

In order to take care of recursion, we extend the ordinary call graph to contain special *recursive nodes* and *approximate nodes*. We illustrate a recursive call-graph through an example.

Call Graph Construction

We shall describe how the call-graph is constructed for the example shown in Figure 4.15. Since routine `main` calls `f`, we create *ordinary nodes* for `main` and `f`, and

```
<[l1: CALL (fn_name, arglist)]> {
{
  /* collect whatever information you want about the arguments */
  /* depending on the type of analysis we are performing       */
  collect_info_about_arguments (arglist);

  /** get the pointer to corresponding node in the call graph  */
  callnode = get_corresponding_call_node(fn_name);

  /** Call function to traverse  call graph                    */
  outdata = traverse_cg(call_node,indata);
  return outdata;
}

Procedure traverse_cg(cg_node, indata): outdata;
{
   /* map information onto the formal params of the new proc   */
   /*  before analyzing the new proc                           */
   data =  map(cg_node, indata);

   /*get the SIMPLE tree_node corr to the body of the new proc */
   tree_node = get_tree_node(cg_node);

   /*go to SIMPLE tree and start analyzing the new procedure   */
   data = analyze(tree_node, data);

   /*After returning from analyzing the new proc, perform     */
   /* unmapping */
   outdata = unmap(cg_node, data);

   return outdata;
}
```

Figure 4.14: Code for Analyzing Procedure Calls

```
main()
{
   f();      ---------(A)
}

f()
{
  if (cond)
     g();    ---------(B)
  else
     h();    ---------(C)
}

g()
{
  if (cond)
     f();    ---------(D)
}

h()
{
  if (cond)
     h();    ---------(E)
}
```

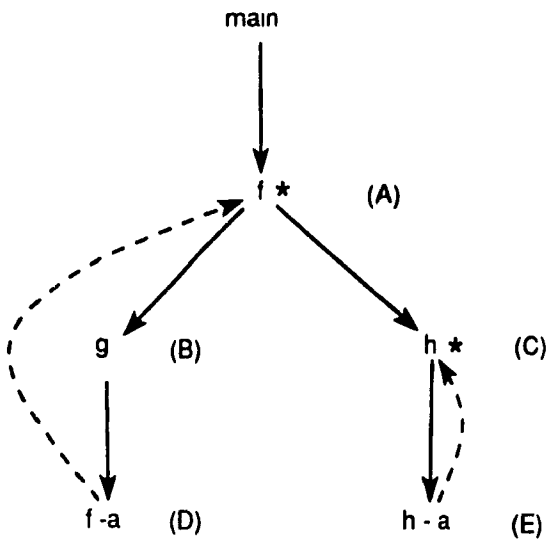


Figure 1.15: Call Graph Construction

connect them by a *calling arc*. Now, `f` calls routines `g` and `h`, and so we create two more ordinary nodes for `g` and `h` and connect them by calling arcs. Now `g` calls `f` recursively. Since `f` has already occurred once in the call-chain from the root, we perform three special actions: (i) we change the first occurrence of `f` to another kind of node, the *recursive node*, represented in Figure 4.15 as `f-*`; (ii) we create an *approximate node* for `f`, called `f-a`, and connect `g` and `f-a` by a calling arc, (iii) we connect the approximate node `f-a` and its corresponding recursive node `f-*` by an *approximate arc*. Approximate arcs are shown by dotted lines in the figure. Similarly, we see that the routine `h` calls itself recursively. So, we convert the first occurrence of `h` to `h-*`, create a new `h-A` node, and connect them by an approximate arc.

Thus, the call graph could be made up of three different kinds of nodes: an ordinary node, a recursive node or an approximate node, whereas edges could be either calling edges or approximate edges.

Interprocedural Analysis:

The recursive call-graph essentially represents the unrolling of the call-graph during recursion to an unbounded depth. We now extend the routine `traverse_cg` to take care of the different kinds of nodes found in the call-graph. Depending on the type of the call-node, routine `traverse_cg` performs different actions.

The modified routine `traverse_cg()` is summarized in the Figure 4.16.

If the call-node corresponding to the procedure call in the SIMPLE tree is an ordinary node, then we essentially do the same as we do for non-recursive procedure calls. We call routine `map` to map the information associated with the actual parameters to the formal parameters. We then go back to SIMPLE to analyze the body of this new procedure. When we return, we perform unmapping of information.

If the call-node corresponding to the procedure call is a recursive node, we have to iterate until a fixed point is reached. We have a list of input-output pairs of information stored at a recursive node. These correspond to all the different types of inputs and outputs possible during one iteration of the fixed-point calculation. During every iteration, we check if there is more than one input at this node. If there is, then this indicates that a recursive call was found with an input not included in the current input approximation. In this case, we merge all these inputs, store this

```
Procedure traverse_cg(cg_node, indata): outdata
{
   switch (typeof(cg_node)) {
      case Ordinary_node:
         /** map information                                          **/
         data =  map(cg_node, indata);
         /** go to SIMPLE tree and start analyzing the new procedure**/
         data = analyze(tree_node, data);
         /** perform unmapping                                        **/
         outdata = unmap(cg_node, data);
         return outdata;
      case Recursive_node:
         data =  map(cg_node, indata);
         data = analyze(tree_node, data);
         /** repeat until fixed point is reached                      **/
         while (more_than_one_input(call_node)||output(call_node) != data) {
            if (more_than_one_input(call_node)) {
               newin = merge_inputs(call_node);
               store_input_in_call_node(newin);
            } else {
               newout = merge(data, output(call_node));
               store_output_in_call_node(newout);
               newin = input(call_node);
            }
            data = analyze(tree_node, newin);
         }
         outdata = unmap(cg_node, data);
         return outdata;
      case Approximate_node:
         /** get the recursive node corresponding to this approx.node**/
         recur_node = get_recur_node(cg_node);
         newout = output(call_node, indata)
         if (newout != NULL) return unmap(newout);
         else {
            store_input_in_call_node(recur_node, indata);
            return BOTTOM;
         }
   }
}
```

Figure 1.16: Procedure to Traverse Call Graph

newly merged input in the tree, and start again with the new merged input. If there is a single input-output pair, but that output and the newly created output from the most recent iteration are not the same, we merge the two outputs, store the new merged output in the tree, and start again with the input present there. Finally, when a fixed point is reached, there is a single input-output pair which corresponds to a superset of all the input-output pairs possible for this procedure. Intuitively, this input-output pair summarizes all the possible unwindings of the call-graph.

If the call-node corresponding to the procedure call is an approximate node, we find its corresponding recursive node in the call-graph. We check the list of input-output pairs to see if an output exists for this particular input. If it does, we just return this output. Otherwise, we store this new input in the recursive node, and return `BOTTOM`. `BOTTOM` here means "I dont know", i.e., we still don't know the output for this particular input, but essentially has the same properties as the `BOTTOM` described in Section 4.3.

The routine `traverse_cg()` is fixed for this particular kind of call-graph. Once the user provides the routines `map()` and `unmap()`, MeTAG can be extended to automatically generate this routine.

4.5 Summary

In this chapter, we have presented two commonly performed analyses, live-variable analysis, which is a backward analysis and reaching definitions, which is a forward one. The analyses have been extended to handle `break` and `continue` constructs in a structured manner, and later made interprocedural. Though we have described two specific flow analysis problems in this chapter, these demonstrate in general, how other simple or complex interprocedural flow analyses, both forward and backward, can be implemented in a structured manner.

Chapter 5

An Advanced Example: Determination of Constants

In the previous chapter, we presented some traditional flow analysis problems and we showed how to make these analyses interprocedural. In this chapter, we give an overview of a completely different sort of analysis, the determination of constants. The purpose of this discussion is to illustrate the diversity of analyses that can be implemented with the analysis generator tool. A formal study of the analysis problem itself is outside the scope of this thesis.

The goal of constant propagation is to discover values that are constant in all possible executions of the program and to propagate the values as far as possible. We can easily implement constant propagation in the same way as we have implemented live variable analysis and reaching definitions, as shown in the previous chapter. Instead, we have approached the problem of finding all the constants in a program in a different manner. Instead of moving to and fro between the call graph and the SIMPLE tree while performing an interprocedural analysis, we make two passes. The first pass is an intraprocedural one, and this collects as much information as possible without analyzing procedure calls, and also sets up some dependence relations between variables and procedure calls. The second pass is an interprocedural one, where we resolve these dependencies, and determine all the constants at every point in the program. The main advantage of this method is that we perform a statement by statement analysis of every procedure body only once, and not as many times as the procedure is called.

The generator tool is employed to generate both the intraprocedural and the inter-procedural modules. In the input specification file for the intraprocedural module, in the action part for a procedure call, we do not have a jump to the call-graph; we just encode the dependencies of the variables on that procedure call. The tool generates the additional code to traverse all procedures once. In the input specification file for the inter-procedural module, in the action part for a procedure call, we have a jump to the call-graph, and then subsequently a jump to the called procedure.

5.1 An Overview

We shall introduce our method through an example. Figures 5.1 and 5.2 illustrate how the information of variables and their dependencies are encoded in the first pass, and later resolved in the second pass.

PASS 1:

Figure 5.1 shows the information collected during the first intraprocedural pass at each point in the program. The information is collected as a set of tuples. Since this pass is intraprocedural, the order in which we analyze the procedures is immaterial. We start with the first procedure, in this case, it is `main`.

- At point (A), we have variable `i` which is a constant. We represent this as a three-field tuple, ⟨`tuple-number, variable-name, value`⟩, and in this case it is ⟨`#1, i, constant[2]`⟩, which essentially means that this is the first tuple, the variable name is `i`, and it is a constant with a value 2.
- At point (B), we are within the `then`-part of the inner conditional, and we see that variable `j` is definitely a constant with a value 4 because `i` is a definitely a constant with a value 2. We represent this in a similar manner. The information set at point (B) now contains two tuples, `#1` and `#2`, representing variables `i` and `j`.
- At point (C), we are within the `else`-part of the inner conditional, and we see that the value of `j` could be a constant, but it depends on the procedure call `square`. We encode this information as shown in the figure, and later, during the second pass, resolve it to be either a constant or not a constant.

```
main()
{
   i = 2;
       -----------------> {<#1, i, constant[2]>}                          (A)
   if (con1) {
      if (con2)
         j = i + 2;
                ---------> {<#2, j, constant[4]>, <#1, i, constant[2]>} (B)
      else
         j = square(i);
                ---------> {<#3, j, call_depend[square]>,
                            <#1, i, constant[2]>}                        (C)
      -------------------> {<#4, j, (#3 @ #2)>, < #1, i, constant[2]> } (D)
      k = j + i;
      -------------------> {<#4, j, (#3 @ #2)>, <#1, i, constant[2]> ,
                            <#5, k, var-depend[(#4 + 2]> }               (E)
   }
   else {
      k = square(3);
                ---------> {<#6, k, call_depend[square]>,
                            <#1, i, constant[2]>}                        (F)
  j = k - 5;
                ---------> {<#6, k, call_depend[square]>,
                            <#1, i, constant[2]>
                            <#7, j, var-depend[#6 - 5]>}                 (G)
   }
   ----------------------> {<#8, j, (#7 @ #4)>, <#1, i, constant[2]>
                            <#9, k, (#6 @ #5)>}                          (H)
}
int square(int m)
{
   int n;
       ------------------> {<#10, m, proc_entry(square)>}                (I)
   n = m * m;
       ------------------> {<#10, m, proc_entry[square]>                 (J)
                            <#11, n, var-depend[#10 * #10]>}
   return n;
       ------------------> {<#10, m, proc_entry[square]>                 (K)
                            <#11, n, var-depend[#10 * #10]>}
}
```

Figure 5.1: Determination of Constants – Pass 1

- At point (D), we have to merge the `then-` and `else-` parts of the inner conditional. The variable `j` could be a constant provided the `then-` and `else-` parts of the conditional assign the same value for `j`. We thus represent the value of `j` as a new tuple `#4`, where the value of `j` depends on tuples `#2` and `#3`. The operator `@` essentially denotes two operands that have to be resolved into one in the second pass.

- At point (E), we find that the value of `k` depends on the value of `j` represented by the tuple `#4`, and `i`, which is definitely a constant. Thus, we represent `k` by tuple `#5` and the code `var-depend`, which says the value of `k` is the value of the variable represented by tuple `#4` plus the constant 2.

- Points (F) and (G) are straightforward and similar to the ones described above.

- At point (H), we have to merge the sets obtained at the end of the `then` and `else` parts of the outer `if-` statement, i.e., we have to merge the sets obtained at points (E) and (G). We then obtain the set illustrated in the figure at point (H). The variable `i` is still a constant, because it has the same constant value on both sides of the `if-then-else` statement. But we have to form new tuples `#8` and `#9`, for variables `j` and `k`, showing new merged values, which will be resolved later during the second pass.

- At point (I), we have just entered procedure `square`. We say the parameter `m` that comes in could be a constant, but it depends on the input to the procedure. We encode this information saying that `t` depends on `proc-entry`.

- At point (J), the local variable `n` is a constant, depending on `m`. We record this as a `var-depend`.

PASS 2:

During this pass, we go over the call-graph and the program, resolving the dependencies collected in pass 1. Since the information is already encoded, we don't have to analyze every statement again during this pass. Figure 5.2 shows the points at which the information gets resolved in the second pass while going over the call-graph.

During the second pass, we directly come to the point just before the first call to the procedure `square` (program point A') At this point, we have the information that

```
main()
{
   i = 2;
   if (con1) {
      if (con2)
         j = i + 2;
      else
               ---------> {<#1, i, constant[2]>}                        (A')
         j = square(i);
               ---------> {<#3, j, constant[4]>,
                           <#1, i, constant[2]>}                        (B')

      -------------------> {<#4, j, constant[4]>,<#1, i, constant[2]>} (C')
      k = j + i;
      -------------------> {<#4, j, constant[4]>,<#1, i, constant[2]>,
                           <#5, k, constant[6]> }                       (D')
   }
   else {
      k = square(3);
               ---------> {<#6, k, constant[9]>,
                           <#1, i, constant[2]>}                        (F')
  j = k - 5;
               ---------> {<#6, k, constant[9]>,
                           <#1, i, constant[2]>
                           <#7, j, constant[4]>}                        (G')
   }
   ----------------------> {<#8, j, constant[4]>, <#1, i, constant[2]>
                           <#9, k, not_a_constant>}                     (H')
}
int square(m)
int m;
{                        (First Call)            (Second Call)
   int n;
   n = m * m;                                |
   return n;                                 |
      -----------> {<#10, m, constant[2]>    |  {<#10, m, constant[3]>} (I')
                    <#11, n, constant[4]>}    |  {<#10, n, constant[9]>}
}
```

Figure 5.2: Determination of Constants - Pass 2

1, which is the actual parameter to procedure square, is a constant. We then jump to procedure square. Since procedure square does not call any other procedure, as can be seen from its call-graph, we can go directly to the end of procedure square, and check the information encoded there. We find that the return value n is a constant if the parameter m is a constant. Since m is a constant, equal to 2, during this call to square, we return to the calling procedure with the information that the return value is a constant equal to 4 (program point I'). Therefore, at point B', we have the information that both i and j are constants. At point (C'), we resolve the value of j. The information encoded there in the first pass, says that the value of j is #3 @ #2, i.e., we have to resolve tuples #2 and #3. At point (B'), we have resolved tuple #3 to be a constant 4. Since both the tuples #3 and #2 have the constant value 1, j is resolved to be a constant 4 at this point. This information, propagated down to point (D'), makes k a constant, with the value 6.

The second call to square has its actual parameter equal to a constant 3. We then jump to the routine square and return with a constant value 9, after resolving information at the end of the procedure square. When we return to the calling procedure, at point (F'), we have both k and i as constants. This information, propagated to the end of the else-part of the outer if, gives us k = 9, j = 4 and i = 1. If we now merge the then- and else- parts of the outer if- statement, we get that both i and j are constants and k is not a constant.

5.2 Pointer and Structure References:

We use aliasing information to deal with pointers. For example, if we have *t = 3, we determine the aliases of *t and track them down as constants. Different parts of a structure reference are treated as different entities: we could have a.b = 3 and a.c.d = 4; these are treated differently, independent of each other.

5.3 Summary

The above method has been implemented using the generator tool. The generator tool creates both the intraprocedural first pass module and the interprocedural second pass module, with the help of input specifications.

Chapter 6

Related Work

In this chapter, we shall classify and describe related research under three categories: intermediate representations used for analysis and optimizations, automating the analyses and optimization phases in a compiler, and other general data-flow analyses methods.

6.1 Intermediate Program Representations:

Traditional optimizations are implemented on control flow-graphs. The intermediate representations used were very close to the machine level code.

RTL is a register transfer language that compilers can use to represent programs during optimization [JM91]. It is used in the GCC compiler and in TS, an optimizing compiler for Smalltalk. RTL provides a typical set of numeric operations, as well as operations to read and write memory and to change the flow of control. The RTL System is a toolkit for constructing code-optimizers; it consists of a number of predefined algorithms that the compiler writer can customize. It is used in the Smalltalk compiler. This forms machine code by combining as many RTL instructions as it can; it never breaks them up into simpler ones. Thus, RTL represents the program at a very low-level, and is at a level even lower than machine code. The design philosophy behind the design of RTL is that a program should be represented with the simplest possible instructions for optimization because a complex instruction

may possibly hide an optimization. Thus RTL has no way of representing high-level structures. Types of loops and loop structures are completely lost; these are all converted to branches and labels in the RTL representation. Array and structure references are broken down to address arithmetic.

Another intermediate representation is U-code [Nye81] which is for Pascal and FORTRAN. It is also a low-level representation, consisting of a linear list of instructions with labels and jumps to change the flow of control. It is also similar to RTL in the sense that it has no way to represent high level program structures and array references.

SUIF (Stanford University Intermediate Format) [TWL+91] is an intermediate form that integrates both high- and low-levels of program representation. It has the ability to represent high-level constructs such as `for`-loops; thus it can maintain program structure while exposing low-level details such as array reference address calculations. SUIF is derived from a bottom-up perspective; the program is represented in low-SUIF, with high-SUIF instructions added at critical points. Thus, SUIF is one intermediate representation that is used for all optimization phases. In our McCAT compiler (refer Figure 1.1), we have both SIMPLE and LAST intermediate representations, and we perform analysis and apply optimization transformations at the appropriate representation level. For example, alias analysis is best done at the SIMPLE level, while instruction scheduling is best done at the LAST level.

6.2 Automating the Analysis and Optimization Phases:

Research in automating the analysis and optimization phases of a compiler is still in its early stages. Few tools exist to help build optimizers, which are usually large and complex, since they must perform many program transformations to get the best code. Sharlit[Tji92] is a system which is designed to simplify building of optimizers in compilers. Sharlit merges the data-flow collection phase and the optimization phases; it takes in a specification and performs *one* type of data-flow analysis and a code-transformation that relies on that analysis. This works on the traditional flow-graphs and basic-blocks. Sharlit uses the following abstractions to develop global analyses and optimizations in a modular fashion

- The nodes of the flow graph.
- Values that flow through the flow graph.
- Flow functions that represent the effect of flow graph nodes and paths on the flow values.
- Action routines to perform the optimization.
- Rules to combine the flow functions to other flow functions for path simplification.

The data-flow analyzer consists of four major components: (i) the control flow analyzer that summarizes the structure of the flow graph, (ii) the path simplifier, generated from the path-simplification rules in the input description and uses control flow information to eliminate some flow nodes, (iii) the iterator makes use of the flow functions and iterates to find a solution for the data-flow equations, and (iv) the propagator that uses the action routines to perform the optimization. Thus, Sharlit does not consider the cases when a particular data-flow analysis is required for more than one optimization, or when an optimization requires more than one data-flow analysis. Furthermore, Sharlit performs intraprocedural analyses and optimizations on flow-graphs, whereas we are looking at interprocedural analyses and optimizations on structured intermediate representations.

Whitfield and Soffa[WS91] describe the automatic generation of global optimizers. They have introduced a General Optimization SPecification Language (GOSpeL) and an optimizer generator (GENesis) that is used to create global optimizers from compact, declarative specifications made in GOSpeL. The specifications mainly consist of a set of preconditions and the actions to optimize the code. The preconditions, in turn, consist of the code pattern to match and the global dependence information (i.e., the control and data dependencies that are required for the specific optimization). The actions take the form of primitive operations that make up the optimization transformation. GENesis analyzes GOSpeL specifications and produces the optimizer. It first produces code (i) for the data structures defined, (ii) for matching the required code pattern, (iii) for checking if the particular data dependences hold, and (iv) for performing the required optimizing transformations. Thus, unlike Sharlit, they do not generate the data-flow analyzer along with the optimizer, but they do assume the data flow information such as anti, output and flow dependence relations are already

computed and available. Further, they also work on flow-graphs, and do not perform any interprocedural optimization.

In the MUG2 Compiler Generating System[Wil81], Wilhelm describes separate analyses and optimization phases which work on a structured abstract syntax tree intermediate representation. Global data-flow analysis is specified using modified attribute grammars and the abstract syntax tree, decorated with the data-flow information, is called an *attributed program tree*. In a single analysis pass, which may be made up of several semantic analysis passes, global data-flow information is collected as attributes associated with nodes in a program tree. Attributes are classified as either *derived* or *inherited* and they are evaluated according to the rules specified for every different kind of node. The optimization passes are implemented as tree transformations, and could also update the data-flow information present in the tree nodes. The system which we are trying to build is similar to this, in the sense that we too work on syntax trees; but we do not use attribute grammars to collect data-flow information. Further, in MUG2, interprocedural analysis is not performed. Moreover, it is complicated to describe or classify more complex analyses like alias-analysis in the form of attribute grammars.

6.3 General Data-flow Analyses Methods:

Traditionally, data-flow analyses are intraprocedural and are implemented on flow-graphs. Each procedure is analyzed independently, and the optimization transformations are performed on the flow graph.

Constant propagation is a well-known global flow analysis and optimization problem that has been approached and solved in several different ways. The first global constant propagation algorithm was developed by Kildall [Kil73]. Several variations of this algorithm have also been published, and a generalization was also published by Kam and Ullman [KU77, WZ91]. The Conditional Constant algorithm [Weg75], is a variant of Wegbreit's algorithm [WZ85], and this finds all constants that can be found by evaluating all conditional branches with all the constant operands. Thus it performs a combination of dead code elimination and constant propagation. The Sparse Conditional Constant algorithm, developed by Wegman and Zadeck [WZ91], finds the same class of constants as the CC algorithm, but runs much faster since it works on a sparse representation (the SSA graph). Thus, almost all the well-known constant

propagation algorithms work on flow graphs. Further, these algorithms are all intraprocedural. Our implementation of the constant propagation is basically different since it is a structure-based interprocedural algorithm. We can extend our implementation very easily to find the same class of constants as that of the CC algorithm: we need to do additional tests on the conditional expressions of the `if`-statements and loop constructs, to see if they could be completely evaluated to either true or false, so that we never evaluate the sections of the program that are never executed. Further, we use a rule-based method and a tool to generate the actual analyzer.

Soffa and Harrold [HS90] give a method to compute interprocedural *definition and use* dependencies. First, they abstract out the definition and use information for each procedure and then propagate the information throughout an interprocedural flow graph. This method is similar to what we do with constant propagation, except that they work on flow graphs, and ours works in a structured, rule-based manner.

Chapter 7

Conclusions

The design of an experimental compiler/architecture testbed helps to coordinate the efforts of compiler writers and architecture designers when designing high-performance computer systems. In the compiler component of a testbed, the optimization phase is, perhaps, the most crucial one, which completely determines the quality of the machine code produced. In order to do a good job of optimization, we need sophisticated analyzing techniques to gather accurate information about the various variables and structures used in a program. This thesis deals with the design of a general analyzer framework for the McCAT (McGill Compiler Architecture Testbed) compiler. In order to experiment with both high-level and low-level optimization techniques and their effects on the underlying architecture, we needed to design suitable intermediate representations of the program. The first part of this thesis deals with design and implementation of the front-end of the compiler with its two intermediate forms, FIRST and SIMPLE.

The FIRST intermediate form provides a complete high-level abstract representation of an entire module or program. It cleanly separates the front-end processing of parsing and type-checking from the back-end processing of analysis, transformation, and code-generation.

The next intermediate form, SIMPLE , forms the right level of program representation on which sophisticated high-level analyses and optimizations could be implemented. As the name suggests, its grammar is simple, yet powerful enough to represent all constructs of C. In SIMPLE , the control flow is structured, complex statements are broken down to a series of simpler statements, complex variable names

are split whenever possible, and all conditionals, loops and switches are transformed to adhere to a fixed format. Complex alias analysis and dependence analysis techniques have been implemented on SIMPLE [Ema92, HDG+92].

The second part of the thesis deals with the development of an analyzer-generator, McTAG, which takes advantage of the compositional nature of SIMPLE to provide a straight-forward, rule-based way of specifying new analyses. The generator takes in a set of specifications which describe the analysis completely independent of the SIMPLE tree, and produces an analyzer that works on SIMPLE. This allows the development of both intraprocedural and interprocedural analyzers. With this tool, we have developed modules for interprocedural live-variable analysis, reaching definitions and constant propagation. The constant propagation algorithm works in two passes; in the first intraprocedural pass, it collects all information and dependencies, while in the second interprocedural pass, it resolves these dependencies to determine constants across procedure boundaries

Future Work:

We now have a solid foundation to experiment with a number of new analyses and optimization techniques. Other classical analyses like available expressions, common subexpressions, first-use and last-use information, etc, can be implemented on SIMPLE , using the analyzer generator. SIMPLE facilitates the development of other new and relatively complex analyses like analyzing dynamically allocated pointers.

Detailed array dependence analysis is currently being implemented on SIMPLE [Jus92]; once this is done, various loop and array optimizations can be experimented with. It would be very interesting to think about an optimizer generator tool that works on SIMPLE , so that different optimizers can be implemented with relative ease.

SIMPLE can be targeted towards different architectures; the third and the lowest level intermediate representation LAST [Don92] currently being developed is targeted towards generating code for RISC machines. We expect that other low-level intermediate representations suitable for other new and different classes of architectures like superscalar and multi-threaded design models can also be designed and studied.

Appendix A

The SIMPLE Grammar

```
all_stmts : stmtlist stop_stmt
          | stmtlist

stmtlist : stmtlist stmt
         | stmt

stmt : compstmt
     | expr ';'
     | IF '(' condexpr ')' stmt
     | IF '(' condexpr ')' stmt ELSE stmt
     | WHILE '(' condexpr ')' stmt
     | DO stmt WHILE '(' condexpr ')'
     | FOR '('exprseq ',' condexpr ';'exprseq ')' stmt
     | SWITCH '(' val ')' casestmts
     | ';'

compsmt . '{' all_stmts '}'
        | '{' '}'
        | '{' decls all_stmts '}'
        | '{' decls '}'

/** decls denotes all possible C declarations. The only difference is that**/
/** the declarations are not allowed to have initializations in them.     **/

exprseq : exprseq ',' expr
        | expr

stop_stmt : BREAK ';'
          | CONTINUE ';'
          | RETURN ';'
          | RETURN val ';'
```

```
          | RETURN '(' val ')' ';'

casestmts : '{' cases default'}'
          | ';'
          | '{' '}'

cases : cases  case
      | case

case  : CASE INT_CONST':' stmtlist stop_stmt

default : DEFAULT ':' stmtlist stop_stmt

expr : rhs
     | modify_expr

call_expr : ID '(' arglist ')'

arglist : arglist ',' val
        | val
        |

modify_expr : varname '=' rhs
            | '*' ID '=' rhs

rhs : binary_expr
    | unary_expr

unary_expr : simp_expr
           | '*' ID
           | '&' varname
           |call_expr
           | unop val
           | '(' cast ')' varname

/** cast here stands for all valid C typecasts **/

binary_expr : val binop val

unop : '+'
     | '-'
     | '!'
     | '~'

binop :  relop
      | '-' | '+' | '/' | '*' | '%'
      | '&' | '|' | '<<' | '>>' | '^'
```

```
relop   '<' | '<=' | '>' | '>=' | '==' | '!='

condexpr . val
         | val relop val

simp_expr . varname
          | INT_CONST
          | FLOAT_CONST
          | STRING_CONST

val   ID
    | CONST

varname   arrayref
        | compref
        | ID

arrayref   ID reflist

reflist : '[' val ']'
        | reflist '[' val ']'

idlist . idlist '.' ID
       | ID

compref : '(' '*' ID ')' '.' idlist
        | idlist
```

Appendix B

The Generator Specification Grammar

```
program:    /*empty*/
       | C_CODE set_all_specs routine
       ;
set_all_specs : set_all_specs set_specs
              |
              ;
set_specs : DATA_NAME COLON IDENTIFIER
          | DATA_TYPE COLON IDENTIFIER
          | TREE_FIELD_INDEX COLON IDENTIFIER
          | CALL_GRAPH_INDEX COLON IDENTIFIER
          | set_anal
          | set_proc
          | set_store
          | set_merge
          | set_copy
          | set_print
          | set_cg
          ;
set_anal  : ANAL_TYPE COLON FORWARD
          | ANAL_TYPE COLON BACKWARD
          ;
set_proc  : PROC_TYPE COLON INTER
          | PROC_TYPE COLON INTRA
          ;
set_store : STORE_OPTION COLON STORE
          | STORE_OPTION COLON NOSTORE
          ;
set_merge : MERGER COLON IDENTIFIER
          ;
```

```
set_copy    COPIER COLON IDENTIFIER
            ;
set_print : PRINTER COLON IDENTIFIER
            ;
set_cg      : CGBUILDER COLON IDENTIFIER
            ;
routine. PROCEDURE identifier '(' paramlist ')' ':' type body
       ;
paramlist: param
         | paramlist ',' param
         |
         ;
param    identifier ':' type
       ;
type   . identifier
       ;
body   : CASE identifier OF cases
       ;
cases  : case
       | cases case
       ;
case   . '<' caseheadlist '>' C_CODE
       ;
caseheadlist · caseheadlist casehead
             | casehead
             ;
casehead . '[' identifier ':' stmttype ']'
         ;
stmttype : WHILE identifier DO identifier
       | DO identifier WHILE identifier
       | FOR identifier identifier identifier DO identifier
       | RETURN
       | BREAK
       | CONTINUE
       | BREAK identifier
       | RETURN identifier
       | SWITCH identifier DO identifier
       | CASE identifier DO identifier
       | DEFAULT DO identifier
       | IF identifier THEN identifier ELSE identifier
       | IF identifier THEN identifier
       | CALL '(' identifier ',' identifier ')'
       | identifier '=' identifier identifier identifier
       | '*' identifier '=' identifier identifier identifier
       | identifier '=' identifier
       | identifier '=' '*' identifier
       | identifier '=' '&' identifier
       | identifier '=' identifier  identifier
```

```
| identifier '=' CALL '(' identifier ',' identifier ')'
| identifier '=' CAST '(' identifier ',' identifier ')'
| '*' identifier '=' identifier
| '*' identifier '=' '*' identifier
| '*' identifier '=' '&' identifier
| '*' identifier '=' identifier  identifier
| '*' identifier '=' CALL '(' identifier ',' identifier ')'
| '*' identifier '=' CAST '(' identifier ',' identifier ')'
| identifier ';' identifier
| identifier identifier identifier
| identifier
| DEFAULTACTION
;
```

Appendix C

A Sample Input and Output for McTAG

C.1 Generator Input for Reaching Definitions

```
%{
O_DATA process_while_loop();
O_DATA process_do_loop();
O_DATA process_for_loop();
O_DATA process_return();
O_DATA process_break();
O_DATA process_continue();
O_DATA process_switch();
O_DATA process case();
O_DATA process_default();
O_DATA process_if();
O_DATA process_call();
O_DATA process_ORD_BINOP();
O_DATA process_STAR_BINOP();
O_DATA process_ORD_ADDR();
O_DATA process_ORD_STAR();
O_DATA process_ORD_UNOP();
O_DATA process_ORD_FUNC_CALL();
O_DATA process_ORD_CAST();
O_DATA process_ORD_ASSG();
O_DATA process_STAR_ADDR();
O_DATA process_STAR_STAR();
O_DATA process_STAR_UNOP();
O_DATA process_STAR_FUNC_CALL();
O_DATA process_STAR_CAST();
```

```
O_DATA process_STAR_ASSG();
O_DATA process_seq();
O_DATA process_expr();
O_DATA process_var();
%}
DATA_NAME : indata
DATA_TYPE : O_DATA
ANAL_TYPE : BACKWARD
PROC_TYPE : INTER
STORE_OPTION: NOSTORE

PROCEDURE reach_def (node:tree, indata:O_DATA) :O_DATA
CASE node OF
<[l1: WHILE cond DO stmt]> {
        return  process_while_loop(l1, cond,stmt,indata);
}
<[l1: DO stmt WHILE cond]> {
        return process_do_loop(l1, cond,stmt,indata);
}
<[l1: FOR init final iter DO  stmt]> {
        return process_for_loop(l1, init,final,iter,stmt,indata);
}
<[l1: RETURN val]> {
        return process_return(l1,val, indata);
}
<[l1: BREAK ]> {
        return process_break(l1,indata);
}
<[l1: CONTINUE]> {
        return process_continue(l1,indata);
}
<[l1: SWITCH val DO stmt]> {
        return process_switch(l1,val, stmt,indata);
}
<[l1: CASE expr DO stmt ]> {
        return process_case(l1,expr,stmt,indata);
}
<[l1: DEFAULT DO stmt ]> {
        return process_default(l1,stmt,indata);
}
<[l1: IF cond THEN thenpart ELSE elsepart]> {
        return process_if(l1,cond,thenpart,elsepart,indata);
}
<[l1. IF cond THEN thenpart ]> {
        return process_if(l1,cond,thenpart,NULL,indata);
}
<[s1: CALL (procname, arglist)]> {
        return process_call(s1,procname,arglist,indata);
```

```
}
<[s1: var1 = val1 binop val2]> {
        return process_ORD_BINOP(s1,var1,val1,binop,val2,indata);
}
<[s1: STAR id = val1 binop val2]> {
        return process_STAR_BINOP(s1,id,val1,binop,val2,indata);
}
<[s1: var1 = ADDR var2 ]> {
        return process_ORD_ADDR(s1,var1,var2,indata);
}
<[s1. var1 = STAR id ]> {
        return process_ORD_STAR(s1,var1, id,indata);
}
<[s1· var1 = unop val ]> {
        return process_ORD_UNOP(s1,var1, unop,val,indata);
}
<[s1. var1 = CALL (procname, arglist)]> {
        return process_ORD_FUNC_CALL(s1,var1, procname,arglist,indata);
}
<[s1: var1 = CAST (var2, type)]> {
        return process_ORD_CAST(s1,var1,var2,type ,indata);
}
<[s1  var1 = var2 ]> {
        return process_ORD_ASSG(s1,var1,var2 ,indata);
}
<[s1: STAR val1 = ADDR var2 ]> {
        return process_STAR_ADDR(s1,val1,var2 ,indata);
}
<[s1: STAR val1 = STAR id ]> {
        return process_STAR_STAR(s1,val1,id ,indata);
}
<[s1· STAR val1 = unop val ]> {
        return process_STAR_UNOP(s1,val1,unop, val ,indata);
}
<[s1: STAR val1 = CALL (procname, arglist)]> {
        return process_STAR_FUNC_CALL(s1,val1, procname,arglist,indata);
}
<[s1: STAR val1 = CAST (var2, type)]> {
        return process_STAR_CAST(s1,val1,var2,type ,indata);
}
<[s1: STAR val1 = var2 ]> {
        return process_STAR_ASSG(s1,val1,var2 ,indata);
}
<[s1  stmt1 ; stmt2]> {
        return process_seq(s1,stmt1,stmt2 ,indata);
}
<[s1:val1 op val2]> {
        return process_expr(s1,val1,op,val2,indata);
```

```
}
<[s1: var]> {
        return process_var(si,var,indata);
}
<[s1: DEFAULTACTION ]> {
return indata;
}
```

C.2 Generator Output for Reaching Definitions

```
#include "stdio.h"
#include "/labs/acaps/acaps2/dlx/compilerwork/bhama/c-ast-c/tree.h"
#include "struct.h"
O_DATA process_while_loop();
O_DATA process_do_loop();
O_DATA process_for_loop();
O_DATA process_return();
O_DATA process_break();
O_DATA process_continue();
O_DATA process_switch();
O_DATA process_case();
O_DATA process_default();
O_DATA process_if();
O_DATA process_call();
O_DATA process_ORD_BINOP();
O_DATA process_STAR_BINOP();
O_DATA process_ORD_ADDR();
O_DATA process_ORD_STAR();
O_DATA process_ORD_UNOP();
O_DATA process_ORD_FUNC_CALL();
O_DATA process_ORD_CAST();
O_DATA process_ORD_ASSG();
O_DATA process_STAR_ADDR();
O_DATA process_STAR_STAR();
O_DATA process_STAR_UNOP();
O_DATA process_STAR_FUNC_CALL();
O_DATA process_STAR_CAST();
O_DATA process_STAR_ASSG();
O_DATA process_seq();
O_DATA process_expr();
O_DATA process_var();


O_DATA  reach_def  (  node  ,  indata  )
tree  node  ;
O_DATA  indata  ;

{
  if (  node  == NULL)
    return  indata  ;

  switch (TREE_CODE(  node  ))   {

    case WHILE_STMT: {
    tree   l1   =   node  ;
    tree   cond   = STMT_WHILE_COND(  node  );
```

```
  tree   stmt   = STMT_BODY(  node  );
  {
  return  process_while_loop(l1, cond,stmt,indata);
  }
}
break;
case FOR_STMT: {
  tree   l1   =   node  ;
  tree   init   = STMT_START(  node  );
  tree   final   = STMT_END(  node  );
  tree   iter   = STMT_ITER(  node  );
  tree   stmt   = STMT_BODY(  node  );
  {
  return process_for_loop(l1, init,final,iter,stmt,indata);
  }
}
break;
case DO_STMT: {
  tree   l1   =   node  ;
  tree   stmt   = STMT_DO_COND(  node  );
  tree   cond   = STMT_BODY(  node  );
  {
  return process_do_loop(l1, cond,stmt,indata);
  }
}
break;
case SWITCH_STMT: {
  tree   l1   =   node  ;
  tree   val   = STMT_SWITCH_EXPR(  node  );
  tree   stmt   = STMT_SWITCH_STMT(  node  );
  {
  return process_switch(l1,val, stmt,indata);
  }
}
break;
case RETURN_STMT:  {
  tree   l1   =   node  ;
  tree   val   = STMT_BODY(  node  );
  if (STMT_BODY(  node  ) != NULL) {
    {
    return process_return(l1,val, indata);
    }
  }
}
break;
case BREAK_STMT:  {
  tree   l1   =   node  ;
  if (STMT_BODY(  node  ) == NULL) {
```

```
    {
    return process_break(l1,indata);
    }
  }
}
break;
case CONTINUE_STMT: {
  tree   l1   =   node  ;
  {
  return process_continue(l1,indata);
  }
}
break;
case CASE_STMT: {
  tree   l1   =   node  ;
  tree   expr   = STMT_CASE_EXPR(  node  );
  tree   stmt   = STMT_CASE_STMT(  node  );
  {
  return process_case(l1,expr,stmt,indata);
  }
}
break;
case DEFAULT_STMT: {
  tree   l1   =   node  ;
  tree   stmt   = STMT_DEFAULT_STMT(  node  );
  {
  return process_default(l1,stmt,indata);
  }
}
break,
case IF_STMT: {
  tree   l1   =   node  ;
  tree   cond   = STMT_COND(  node  );
  tree   thenpart   = STMT_THEN(  node  );
  if (STMT_ELSE(  node  ) == NULL) {
    {
    return process_if(l1,cond,thenpart,NULL,indata);
    }
  }
}
{
  tree   l1   =   node  ;
  tree   cond   = STMT_COND(  node  );
  tree   thenpart   = STMT_THEN(  node  );
  tree   elsepart   = STMT_ELSE(  node  );
  if (STMT_ELSE(  node  ) != NULL) {
    {
    return process_if(l1,cond,thenpart,elsepart,indata);
```

```
    }
  }
}
break;
case TREE_LIST: {
  tree   s1   =   node  ;
  tree   stmt1   = TREE_VALUE(  node  );
  tree   stmt2   = TREE_CHAIN(  node  );
  {
  return process_seq(s1,stmt1,stmt2 ,1ndata);
  }
}
break;
case EXPR_STMT:
  switch (TREE_CODE(STMT_BODY(  node  ))) {
    case CALL_EXPR:  {
    tree s1  =   node  ;
    tree procname = TREE_OPERAND(TREE_OPERAND(STMT_BODY(s1),0),0);
    tree   arglist   = TREE_OPERAND(STMT_BODY(s1),0);
      {
      return process_call(s1,procname,arglist,1ndata);
      }
    }
    break;
    default:
    switch (TREE_CODE(TREE_OPERAND(STMT_BODY(  node  ),0))){
      case INDIRECT_REF:  {
      tree mod_expr = STMT_BODY(  node  );
      switch (TREE_CODE(TREE_OPERAND( mod_expr,1))){
        case ADDR_EXPR: {
        tree   s1   =   node  ;
        tree val1=TREE_OPERAND(TREE_OPERAND(STMT_BODY(s1),0),0);
        tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
          {
          return process_STAR_ADDR(s1,val1,var2 ,1ndata);
          }
        }
        break;
        case INDIRECT_REF: {
        tree s1   =   node  ;
        tree val1 = TREE_OPERAND(TREE_OPERAND(STMT_BODY(s1),0),0);
        tree   1d   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
          {
          return process_STAR_STAR(s1,val1,1d ,1ndata);
          }
        }
        break;
        case CALL_EXPR: {
```

```
tree   s1   =   node  ;
tree val1=TREE_OPERAND(TREE_OPERAND(STMT_BODY(s1),0),0);
tree procname=TREE_OPERAND(TREE_OPERAND(
TREE_OPERAND(mod_expr, 1),0),0);
tree arglist-TREE_OPERAND(TREE_OPERAND(mod_expr,1),1);
  {
  return process_STAR_FUNC_CALL(s1,val1, procname,
  arglist,indata);
  }
}
break;
case NOP_EXPR: {
tree   s1 = node  ;
tree val1=TREE_OPERAND(TREE_OPERAND(
STMT_BODY(  s1  ),0),0);
tree var2=TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
tree   type = TREE_TYPE(TREE_OPERAND(mod_expr, 1));
  {
  return process_STAR_CAST(s1,val1,var2,type,indata);
  }
  }
break;
default:
if (is_binary(TREE_OPERAND(mod_expr,1) )) {
  tree   s1   =   node  ;
  tree   id  = TREE_OPERAND(TREE_OPERAND(STMT_BODY(  s1  ),0),0);
  tree   val1   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
  tree   binop   = TREE_OPERAND(mod_expr, 1);
  tree   val2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
  {
  return process_STAR_BINOP(s1,id,val1,binop,val2,indata);
  }
  break,
}
if (is_unary(TREE_OPERAND(mod_expr,1) )) {
  tree   s1   =   node  ;
  tree   val1  = TREE_OPERAND(TREE_OPERAND(STMT_BODY(  si  ),0),0);
  tree   unop   = TREE_OPERAND(mod_expr, 1);
  tree   val   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
  {
  return process_STAR_UNOP(s1,val1,unop, val ,indata);
  }
break;
}
{
  tree   s1   =   node  ;
  tree   val1  = TREE_OPERAND(TREE_OPERAND(STMT_BODY(  s1  ),0),0);
  tree   var2   = TREE_OPERAND(mod_expr, 1);
```

```
    {
    return process_STAR_ASSG(s1,val1,var2 ,1ndata);
    }
  break;
  }
  return   1ndata  ;
}
break;
default:
  {
  tree mod_expr = STMT_BODY(  node  );
  switch (TREE_CODE(TREE_OPERAND( mod_expr,1))){
    case ADDR_EXPR: {
      tree   s1   =   node  ;
      tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
      tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
      {
      return process_ORD_ADDR(s1,var1,var2,1ndata);
      }
    }
    break;
    case INDIRECT_REF: {
      tree   si   =   node  ;
      tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
      tree   1d   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
      {
      return process_ORD_STAR(s1,var1, 1d,1ndata);
      }
    }
    break,
    case CALL_EXPR: {
      tree   s1   =   node  ;
      tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
      tree   procname   = TREE_OPERAND(TREE_OPERAND(
               TREE_OPERAND(mod_expr, 1),0),0);
      tree   arglist   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
      {
      return process_ORD_FUNC_CALL(s1,var1, procname,arglist,1ndata);
      }
    }
    break;
    case NOP_EXPR: {
      tree   s1   =   node  ;
      tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
      tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
      tree   type   = TREE_TYPE(TREE_OPERAND(mod_expr, 1));
      {
      return process_ORD_CAST(s1,var1,var2,type ,1ndata);
```

```
        }
      }
      break;
      default:
        if (is_binary(TREE_OPERAND(mod_expr,1) )) {
          tree   s1   =   node  ;
          tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
          tree   val1   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
          tree   binop   = TREE_OPERAND(mod_expr, 1);
          tree   val2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
          {
          return process_ORD_BINOP(s1,var1,val1,binop,val2,indata);
          }
        break;
        }
        if (is_unary(TREE_OPERAND(mod_expr,1) )) {
          tree   s1   =   node  ,
          tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
          tree   unop   = TREE_OPERAND(mod_expr, 1);
          tree   val   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
          {
          return process_ORD_UNOP(s1,var1, unop,val,indata);
          }
        break;
        }
        {
        tree   s1   =   node  ;
        tree   var1  = TREE_OPERAND(STMT_BODY(  s1  ),0);
        tree   var2   = TREE_OPERAND(mod_expr, 1);
        {
        return process_ORD_ASSG(s1,var1,var2 ,indata);
        }
        break;
        }
      return   indata  ;
    }
    break,
    }
  }
  break,
}
break,
default:
  if (TREE_CODE(  node  ) == MODIFY_EXPR) {
  switch (TREE_CODE(TREE_OPERAND(  node  ,0))){
    case INDIRECT_REF:  {
    tree mod_expr =   node  ;
    switch (TREE_CODE(TREE_OPERAND( mod_expr,1))){
```

```
case ADDR_EXPR: {
tree   s1   =   node  ;
tree   val1  = TREE_OPERAND(TREE_OPERAND(  s1  ,0),0);
tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0),
  {
  return process_STAR_ADDR(s1,val1,var2 ,indata);
  }
}
break;
case INDIRECT_REF: {
  tree   s1   =   node  ;
  tree   val1  = TREE_OPERAND(TREE_OPERAND(  s1  ,0),0);
  tree   id   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
  {
  return process_STAR_STAR(s1,val1,id ,indata);
  }
}
break;
case CALL_EXPR: {
  tree   s1   =   node  ;
  tree   val1  = TREE_OPERAND(TREE_OPERAND(  s1  ,0),0);
  tree   procname   = TREE_OPERAND(TREE_OPERAND(
                TREE_OPERAND(mod_expr, 1),0),0);
  tree   arglist   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
  {
  return process_STAR_FUNC_CALL(s1,val1, procname,arglist,indata);
  }
}
break;
case NOP_EXPR: {
  tree   s1   =   node  ;
  tree   val1  = TREE_OPERAND(TREE_OPERAND(  s1  ,0),0);
  tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
  tree   type   = TREE_TYPE(TREE_OPERAND(mod_expr, 1));
  {
  return process_STAR_CAST(s1,val1,var2,type ,indata);
  }
}
break;
default:
if (is_binary(TREE_OPERAND(mod_expr,1) )) {
  tree   s1   =   node  ;
  tree   id  = TREE_OPERAND(TREE_OPERAND(  s1  ,0),0);
  tree   val1   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
  tree   binop   = TREE_OPERAND(mod_expr, 1);
  tree   val2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
  {
  return process_STAR_BINOP(s1,id,val1,binop,val2,indata);
```

```
        }
      break
      }
      if (is_unary(TREE_OPERAND(mod_expr,1) )) {
        tree   s1   =   node  ,
        tree   val1  = TREE_OPERAND(TREE_OPERAND(  si  ,0),0);
        tree   unop   = TREE_OPERAND(mod_expr, 1);
        tree   val   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
        {
        return process_STAR_UNOP(s1,val1,unop, val ,indata);
        }
      break;
      }
      {
      tree   s1   =   node  ;
      tree   val1  = TREE_OPERAND(TREE_OPERAND(  s1  ,0),0);
      tree   var2   = TREE_OPERAND(mod_expr, 1);
        {
        return process_STAR_ASSG(s1,val1,var2 ,indata);
        }
      break,
      }
    return   indata  ;
    }
}
break,
default   {
  tree mod_expr =   node  ;
  switch (TREE_CODE(TREE_OPERAND( mod_expr,1))){
  case ADDR_EXPR: {
    tree   s1   =   node  ;
    tree   var1  = TREE_OPERAND(  s1  ,0);
    tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
    {
    return process_ORD_ADDR(s1,var1,var2,indata);
    }
  }
  break;
  case INDIRECT_REF: {
  tree   s1   =   node  ;
  tree   var1  = TREE_OPERAND(  s1  ,0);
  tree   id   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
    {
    return process_ORD_STAR(s1,var1, id,indata);
    }
  }
  break;
  case CALL_EXPR: {
```

```
  tree   s1   =   node  ;
  tree   var1  = TREE_OPERAND(  s1  ,0);
  tree   procname   = TREE_OPERAND(TREE_OPERAND(
                TREE_OPERAND(mod_expr, 1),0),0);
  tree   arglist   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
  {
  return process_ORD_FUNC_CALL(s1,var1, procname,arglist,indata);
  }
}
break;
case NOP_EXPR: {
  tree   s1   =   node  ;
  tree   var1  = TREE_OPERAND(  s1  ,0);
  tree   var2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0);
  tree   type   = TREE_TYPE(TREE_OPERAND(mod_expr, 1));
  {
  return process_ORD_CAST(s1,var1,var2,type ,indata);
  }
}
break;
default:
  if (is_binary(TREE_OPERAND(mod_expr,1) )) {
    tree   s1   =   node  ;
    tree   var1  = TREE_OPERAND(  s1  ,0);
    tree   val1   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0),
    tree   binop   = TREE_OPERAND(mod_expr, 1);
    tree   val2   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),1);
    {
    return process_ORD_BINOP(s1,var1,val1,binop,val2,indata);
    }
  break;
  }
  if (is_unary(TREE_OPERAND(mod_expr,1) )) {
    tree   si   =   node  ;
    tree   var1  = TREE_OPERAND(  s1  ,0);
    tree   unop   = TREE_OPERAND(mod_expr, 1);
    tree   val   = TREE_OPERAND(TREE_OPERAND(mod_expr, 1),0),
    {
    return process_ORD_UNOP(s1,var1, unop,val,indata);
    }
  break;
  }
  {
  tree   s1   =   node  ;
  tree   var1  = TREE_OPERAND(  s1  ,0);
  tree   var2   = TREE_OPERAND(mod_expr, 1);
    {
    return process_ORD_ASSG(s1,var1,var2 ,indata);
```

```
            }
          break;
          }
        return   indata  ;
      }
      break;
    }
    break;
  }
  if (is_var_or_const(  node  )) {
    tree   s1   =   node  ;
    tree   var   =   node  ;
    {
    return process_var(s1,var,indata);
    }
  break;
  }
  if (is_binary(  node  )) {
    tree   s1   =   node  ;
    tree   val1   = TREE_OPERAND(  node  ,0);
    tree   op   =   node  ;
    tree   val2   = TREE_OPERAND(  node  ,1);
    {
    return process_expr(s1,val1,op,val2,indata);
    }
  break;
  }
return   indata  ;
}
```

Bibliography

[ACK87] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[Alt90] E. R. Altman. Minimizing Pipeline Interlocks through Instruction Scheduling. Course Project for Spring 1990 Csc 308-762B with Prof. G. Gao, 1990.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237 251, 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.

[Bak77] B. Baker. An Algorithm for Structuring Flowgraphs. *JACM*, 24(1):98 120, 1977.

[Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 29 41, 1979.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm *Communications of the ACM*, 21:724 736, 1978.

[BEH91] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 122-131, April 1991.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 19-59, January 11 13 1989. Austin, TX.

[Coo85] Keith Cooper. Analyzing aliases of reference formal parameters. *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1985.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[DH179] J.J. Dongarra and A.R. Hinds. Unrolling loops in FORTRAN. *Software-Practice and Experience*, 9:219–226, 1979.

[Don92] Christopher Donawa. The LAST McCAT Intermediate Representation. McCAT ACAPS Design Note 3, McGill University, School of Computer Science, 1992.

[Ema92] Maryam Emami. An Alias Analysis for Stack-allocated Data Structures. Master's thesis, McGILL University, expected December 1992.

[Ero92] A. Erosa. Restructuring SIMPLE. McCAT ACAPS Design Note 9, McGill University, School of Computer Science, 1992.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 11–16, Palo Alto, CA, June 1986.

[HDG+92] Laurie Hendren, Chris Donawa, Guang Gao, Justiani, Maryam Emami, and Bhama Sridharan. Designing the McCAT Compiler based on a Family of Structured Intermediate Representations. ACAPS Design Memo 46, McGill University, School of Computer Science, 1992.

[HN89] Laurie J. Hendren and Alexandru Nicolau. Interference analysis tools for parallelizing programs with recursive data structures. In *Proceedings of the International Conference on Supercomputing*, pages 205–214, June 1989.

[HS90] Mary Jean Harrold and Mary Lou Soffa. Computation of Interprocedural Definition and Use Dependencies. *Proceedings of the 1990 International Conference Computing Languages, IEEE*, pages 297–306, 1990.

[JM91] Ralph E. Johnson and Carl McConnell. The RTL System: A Framework for Code Optimization. Technical report, University of Illinois at Urbana-Champaign, 1991.

[Joh75] S.C. Johnson. YACC: Yet Another Compiler Compiler. Computing Science Technical Report 32, Bell Laboratories, 1975.

[Jus92] J. Justiani. Array Dependence Analysis on SIMPLE. McCAT ACAPS Design Note 10, McGill University, School of Computer Science, 1992.

[Kil73] G. Kildall. A unified approach to global program optimization. *Conference Record of First ACM Symposium on Principles of Programming Languages*, pages 194-206, January 1973.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305-317, 1977.

[Lam90] Monica S. Lam. Instruction scheduling for superscalar architectures. *Annual Review of Computer Science*, 4:173-201, 1990.

[Lan92] William A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, 1992.

[LR92] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation*, pages 235-248, June 1992.

[LS75] M.E. Lesk and E. Schmidt. LEX - a Lexical Analyzer Generator. Computing science technical report, Bell Laboratories, 1975.

[Muk91] Chandrika Mukerji. Instruction scheduling at the RTL level. Technical Report ACAPS Note 28, McGill University, 1991.

[Nye81] P. Nye. S-1 U-code: An Intermediate Language for Pascal and FORTRAN. S-1 Project Document PAIL-8, Computer System LAB, Stanford University, 1981.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184-1201, December 1986.

[Ros85] David S. Rosenblum. A Methodology for the Design of Ada Transformation Tools in a DIANA Environment. *IEEE Software*, pages 24-33, 1985.

[Sre92] V. Sreedhar. Unnesting Nested Blocks in SIMPLE. McCAT ACAPS Design Note 7, McGill University, School of Computer Science, 1992.

[Sri91] Bhama Sridharan. Creation and transformations of the abstract syntax tree. ACAPS Design Note 27, School of Computer Science, McGill University, 1991.

[Sri92] Bhama Sridharan. The SIMPLE AST - McCAT Compiler. ACAPS McCAT Design Note 2, School of Computer Science, McGill University, 1992.

[Sta90] R. M. Stallman. Using and porting the GNU CC. Technical report, Free Software Foundation, Cambridge, MA, 1990.

[Tji92] Steve W. K. Tjiang and John L. Hennessy. Sharlit — a tool for building optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, June 1992.

[TWL+91] Steven W.K. Tjiang, Micheal E. Wolf, Monica S. Lam, Karen L. Pieper, and John L. Hennessy. Integrating Scalar Optimization and Parallelization. In *Proceedings of the Fourth International Workshop on Languages and COmpilers for Parallel Computing*, pages 137–151, Santa Clara, California, USA, August 1991. Springer-Verlag, Lecture Notes in Computer Science.

[Weg75] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, 1:270–285, 1975.

[Wil81] Reinhard Wilhelm. *Global Flow Analysis and Optimization in the MUG2 Compiler Generating System*, pages 132–159. Prentice-Hall, Inc., 1981.

[WO75] M. H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Comput. J.*, 21(2), 1975.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.

[WS91] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 120–129, June 1991.

[WZ85] Mark Wegman and Ken Zadeck. Constant propagation with conditional branches. *Conference Rec. Twelfth ACM Symposium on Principles of Programming Languages*, pages 291-299, January 1985.

[WZ91] Mark Wegman and Ken Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, pages 181–210, April 1991.