Towards Automated Construction of Tabular Expressions

By
Yazhi Wang, B.S.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2006) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
Towards Automated Construction of Tabular Expressions

AUTHOR: Yazhi Wang, B.S.
(Beijing University of Posts and Telecommunications, China)

SUPERVISOR: Dr. Alan Wassyng

NUMBER OF PAGES: xiii, 117

Abstract

Deriving precise descriptions of existing programs using automated actions plays a significant role in software engineering, especially in projects that are not well documented. Tabular expressions (tables) are practical formalized specification notations that can be used in place of conventional mathematical expressions. Building function tables from source code is a tremendous aid to understand the behavior of target programs for inspectors and maintainers. However, generating those tables manually is tedious and time consuming.

This thesis presents an automated method that will help extract vector function tables from imperative programs in C. By dealing with the three primitive constructs (assignments, alternations, iterations) we aim to translate the target programs into functional documentation using tabular expressions. We discuss the difficulties we encountered and the methods we chose to overcome those difficulties. Loop termination and pattern matching are also discussed in our analysis. Currently, we stop short of producing the tabular expressions, but it is easy to see that tables can be generated from the expressions produced by our tool.

Acknowledgments

The completion of this degree was made possible through the support and cooperation of many people. First of all, I thank my supervisor, Dr. Alan Wassyng. I am grateful for his guidance. Above all, I appreciate his time and patience as he reviewed the numerous drafts of this thesis. I am also grateful to Dr. William Farmer and Dr. Ridha Khedri for serving on my committee. I appreciate their comments and their willingness to sacrifice their time and energy to help me. I thank Dr. Jeffery Zucker for giving so many comments on my academic writing in English. I appreciate the research support that I received from Dr. Jacques Carette, Dr. Wolfram Kahl and Dr. Emil Sekerinski. I would also like to thank the administrative staff in the Department of Computing and Software for their help. I am particularly grateful to Laurie LeBlanc for her assistance. Finally, I thank my family for their love and support over the years. In particular, I thank my loving girl friend, Jie Wu. Without her unfailing love and emotional support, I doubt that I would ever have finished. Furthermore, she helped by reviewing numerous drafts of this thesis. Above all, I realize just how many people have supported me and contributed to this degree in one form or another. I am deeply grateful to everyone that has helped and supported me over the years of my graduate study.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Deriving precise descriptions of existing behaviour from code using automated actions plays a significant role in software engineering, especially in projects that are not well documented. In this reverse engineering step people want to extract intended behaviours from code. To describe those programs in a more readable way, tabular expressions are used more and more in critical projects, starting from the 1970s when David Lorge Parnas and others at U.S. Naval Research Laboratories used them to document requirements for the A-7E aircraft [16]. Later in the Darlington Shutdown Systems of Ontario Hydro, now Ontario Power Generation (OPG), tables were extracted from the code manually to help evaluate those programs. Although it is more than two decades from the time that tabular expression were first used, there are not many tools which support this readable and precise expression. As Wassyng and Lawford mentioned in [53], although UML had no semantic basis, it has proved to be extremely successful in industry. The success of UML, to a large extent, can be attributed to the comprehensive tool support that was available for it. Therefore we can confidently say tabular expressions could become more accepted in industry if we make a full toolkit to support them.

This thesis presents a proof-of-concept prototype tool for software maintainers to generate function tables in an automated way from C language for further analysis.

Inspection and verification can be made easier with help such a tool. Based on static program analysis this tool produces expressions that describe the behaviour of the code. The expressions are intended to be at a high level of abstraction than the code, and in a form that can be used to populate the cells of a vector function table.

1.2 Our Approach

Through decades of endeavor in reverse engineering researchers have invented several different ways to derive high level specifications from programs. For example, Ward built a language called the "Wide Spectrum Language" [50] [48] [47] which includes low-level programming constructs and high-level abstract specifications within a single language. Although people generally regard reverse engineering as a method for deriving the design from source code, different specifications are used related to particular reverse engineering projects.

So, in our particular context we want to create function tables from high-level imperative languages. A major difficulty in reverse engineering is that we often lack the big picture of the system as developed at the design stage. One way to reconstruct this picture is to extract the relations that describe the behaviour of individual functions in the code, so that the mathematical composition of the functions describes the overall behaviour. Our goal is to structure the mathematical expressions for each code function so that we are able to describe each code function using tabular expressions.

The approach described in this thesis applies to a single code function at a time. One important thing is the variable set of monitored and controlled variables which should be caught before our analysis. All controlled variables will be represented in formula expressed by vector function tables. We know that the function of the program is combined by all partial relations which are represented in every tabular cell. For those relations we have one branch of execution related to it. Combining all the simple assignments in those respective execution branches, we can get the final results of every output variable by symbolic evaluation. Things become more complex when we encounter loop structures. This thesis presents a variable's function in recurrence equations which can help us to deal with analysis in automatic actions.

Also by using pattern matching technology we can simplify our results into more readable forms.

1.3 Contributions and Thesis Scope

1.3.1 Contributions

Our major contributions are:

- We provided an automated table generation tool that will support the creation of tabular expressions.
- We showed one way in which we can extract functions that describes the behaviour of a code variable by recurrence equations, which can help us understand programs in an easier way and make it possible for the analysis to be handled by tools.
- We developed a method of pattern matching to deal with loop statements in static program analysis.
- We described a good experience of how to use formal semantics in real industry world and how to implement them in a tabular expression toolkits.

1.3.2 Thesis Scope

We discuss relevant literature concerning previous reverse engineering, tabular expression in Chapter 2 and 3. Chapter 4 presents our analysis and the methods we used to derive functional descriptions of code written in a high level language. Chapters 5 and 6 include the requirements of our tool and how we implemented this tool. Then we discuss testing results for our tool are shown in Chapter 7. Finally, we present our conclusions about contribution and future work in Chapter 8.

Chapter 2

Specification Recovery from Code

This chapter briefly introduces existing methods which can derive specifications from source code, and references relevant literature.

2.1 Overview of Reverse Engineering

The objective of our Table Generation Tool is to produce an abstract specification from an imperative program written in C. A number of concepts and useful ideas in reverse engineering have emerged through recent research and experience. A survey in this valuable literature will tremendously help us in our analysis and implementation.

Software reverse engineering as defined in [5], also known as both renovation and reclamation, is the examination and alteration of a software system to reconstitute it in a new form, and the subsequent implementation of the new form.

The goal of software re-engineering is to take an existing system and generate from it a new system which is called the target system, that has the same properties as a system created by modern software development methods. These desired software properties include: maintainability, portability, reliability, reusability, quality of documentation, testability, and usability [4].

Figure 2.1 contains a graphical depiction of a process model for reverse and re-engineering [4]. In the figure, two triangles are used to represent the different levels of abstraction. The arrows show the direction of the software process steps. In

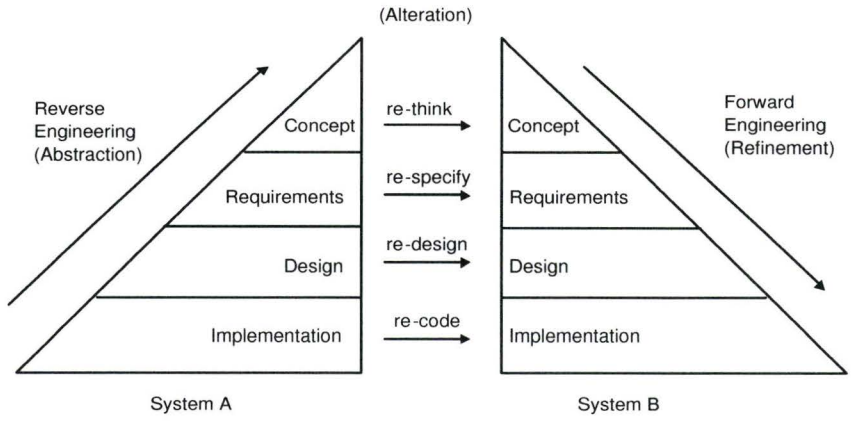

Figure 2.1: General model for software re-engineering

the triangle for System B the process of refinement is performed from Concept to Implementation. In contrast with B System, the triangle for A shows the figure of performance of abstraction.

2.2 Major methods of Specification Recovery

Specification Recovery from code is the process of deriving a higher level abstraction from target programs which is within the domain of Software Re-Engineering. A number of methods have been developed to achieve this mission in recent research. These methods include both informal methods and formal methods.

Relevant formal methods are presented below.

2.2.1 FermaT

In his paper [50], Ward presents an approach to extract high-level specifications from unstructured source code. This method is based on a theory of program refinement and transformation, which is used as the basis for the development of a catalogue of powerful semantics-preserving transformations. Ward's transformations are based on

a rigorous mathematical foundation. Without such a foundation, it is all too easy to assume that a particular transformation is correct, and come to rely upon it , only to discover that there are certain special cases where the transformation is not correct.

Foundations

In FermaT project a new formal language, Wide Spectrum programming Language (called WSL), is used as a lower level programming language and high level specification language at the same time. All the transformation techniques between these two levels have been proven correct and have mechanically checkable applicability conditions [50]. This method helps the user do the transformations with confidence. Infinite first order logic has been used to express the weakest precondition of programs in the kernel language of WSL.

Notation $P\{S\}Q$ (called a Hoare Triple) presents a partial correctness model of a program's execution, which means if S starts in a state satisfying P and terminates then its terminating state satisfies Q. We call P the precondition, which describes the set of initial states, and we call Q postcondition, which describes the set of final states. So the *weakest precondition* $wp(S, Q)$ describes the set of all states in which statement S starts and terminates with postcondition Q true. We sometime use $wlp(S, Q)$ to describes the weakest liberal precondition, which indicates that it refers to partial correctness and includes the non-termination cases.

There are also some theorems which are the foundations of FermaT project [7]:

Theorem1: If $P \Rightarrow W$, then

$$W\{S\}Q \Rightarrow P\{S\}Q$$

Theorem2: $wp(S, False) \Leftrightarrow False$

Theorem3: For any mechanism (program) S, and any postconditions Q and R, we have

$$wp(S, Q) \wedge wp(S, R) \Leftrightarrow wp(S, Q \wedge R)$$

Theorem4: For any mechanism (program) S, and any postconditions Q and R, we have

$$wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q \vee R)$$

Theorem5: For any deterministic mechanism (program) S, and any postconditions Q and R, we have

$$wp(S, Q) \vee wp(S, R) \Leftrightarrow wp(S, Q \vee R)$$

Since the application of the weakest precondition predicate transformer [8], weakest precondition techniques are primarily used for program derivation and specification. In reverse engineering projects led by M.Ward, *wp* plays a role as a guideline for constructing formal specifications.

Major Stages of Specification Recovery with FermaT

FermaT is a program transformation system based on the theory of program refinement as equivalence developed in [45] and applied to Reverse Engineering in [49]. This transformation system is intended as a practical tool in software maintenance and programming comprehension.

In [49], four stages are adopted to extract a formal specification from given program. The first three stages are carried out with a prototype of FermaT, starting with the original program and applying general purpose transformations. However, the last stage involves user intervention. During the four stages, *wp* has been used to prove the correctness in the transformation.

First Stage: Restructure and Simplify

In the first stage some structure-like switches are re-expressed as primary structures in a kernel language. Users do not have to understand the semantics of the target programs before transforming them. The system takes care of all the correctness conditions and the details of those transformations. Commonly what are "cleaner semantics"? the stage of restructuring to re-express target programs with more clear semantics is widely adopted.

Second Stage: Abstract Data Types

After the restructuring stage, high level abstract data types are extracted from the target programs. A semi-automated method of this abstraction involves human

input in selecting abstract equivalents. Some simple types can be transformed automatically.

Third Stage: Restructure and Simplify Again

This stage is similar to the first stage. However, the simplification targets are the abstract data types produced during the previous stage.

Fourth Stage: Specification Level

To date, totally automated methods can not be implemented in this final stage of abstracting the specification. Two methods are introduced in specification abstraction. One uses loop invariants which are conditions preserved by a loop throughout its execution. Loop invariants are really significant in specification recovery. We will talk about them more in later sections. The other method involves changing the data structure. A *list* data structure is introduced in high level specification language and related operations can be used to describe equivalent operations which are implemented by loops in low level programming languages.

Conclusions

The methods and techniques we discussed above present a new approach which is based on a wide spectrum language including both a lower level programming language and a high level specification language. Most of this work focuses on the programming transformation which is proven correct by using the concept of weakest precondition. The disadvantage of this approach is that the abstraction of the specification still depends on human intervention.

2.2.2 Strongest Postcondition

There is another different analysis method emerging in the project AUTOSPEC [12], in which the Strongest Postcondition is adopted to construct a high level specification from programs written in imperative languages like C. In this section we will discuss this approach and its related support tools.

Background

As M. Ward did in his approach, G. Gannod also uses Hoare triples as the formal notation in his research. However, he uses the strongest postcondition in contrast with the weakest precondition.

We already discussed the Hoare triple $P\{S\}Q$ to present the partial correctness of a program. $wp(S, Q)$ can identify the weakest precondition of statement S and postcondition Q. G. Gannod uses this triple differently, so that $sp(S, P)$ represents the strongest postcondition, meaning that if a program starts in state P, then the execution of S will place the program in state $sp(S, P)$ if S terminates. Figure 2.2 shows the difference between these two methods, as depicted in [10].

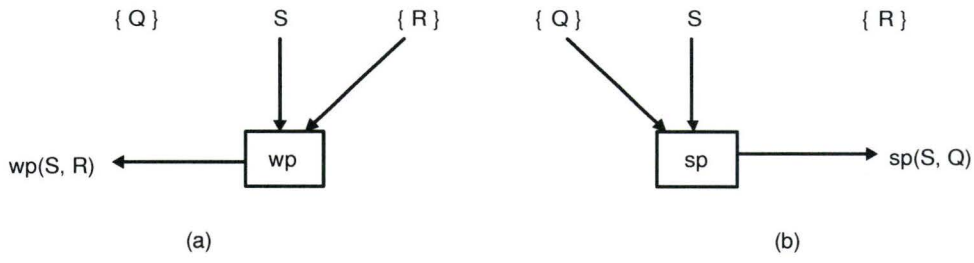

Figure 2.2: Black box representation of (a) wp (b) sp

This forward derivation rule which is shown in Figure 2.3 can be used as a predicate transformer to extract high level specifications. The use of these predicate transformers for reverse engineering have different implications compared with *wp*. Using *wp* means that the postcondition is known. Nevertheless, the postcondition is always what we want when we try to derive the specification. So we notice that the approach using *wp* can only be used as a guideline on which all proof about the transformation is based. As such, it seems that *sp* is more applicable to reverse engineering.

Analysis to Primitive Constructs

In order to derive the strongest postcondition of target programs, methods that deal with the primitive constructs such as assignment, alternation, sequence, iteration and procedure are needed. In his paper [10], G. Gannod describes the semantics of the predicate transformers *wlp* and *sp* as they apply to each primitive and then, for reverse engineering purposes, describes specification recovery in terms of Hoare triples.

Construct	*sp* Semantics
$sp(x := e, Q)$	$\equiv (\exists v :: Q_v^x \wedge x = e_v^x)$
$sp(IF, Q)$	$\equiv sp(S_1, B_1 \wedge Q) \vee ... \vee sp(S_n, B_n \wedge Q)$
$sp(DO, Q)$	$\equiv \neg B \wedge (\exists i : 0 \leq i : sp(IF^i, Q))$
$sp(S_1; S_2, Q)$	$\equiv sp(S_2, sp(S_1, Q))$

Figure 2.3: Strongest postcondition semantics

Gannod gives the semantics for these primitive constructs (shown in Figure 2.3), by which the strongest postcondition is used directly as a predicate transformer.

A suite of Tools to support

To make these research methods more practical to use, Gannod also provides a suite of tools to support them [12]. The tools include:

AUTOSPEC: supports the construction of specifications using the semantics of the strongest postcondition predicate transformer;

SPECGEN: derives abstract specifications from as-built specifications;

SPECEDIT: a specification editor with a graphical user interface front-end that supports the construction of syntactically correct specifications;

TPROVER: a tableau theorem prover that verifies the consistency of specifications that are modified by a user.

Gannod also describes the relationship between the tools, as shown in Figure 2.4. In this figure, circles are used to represent processes, parallel lines represent data stores, rectangles represent actors, and arrows represent flow of data.

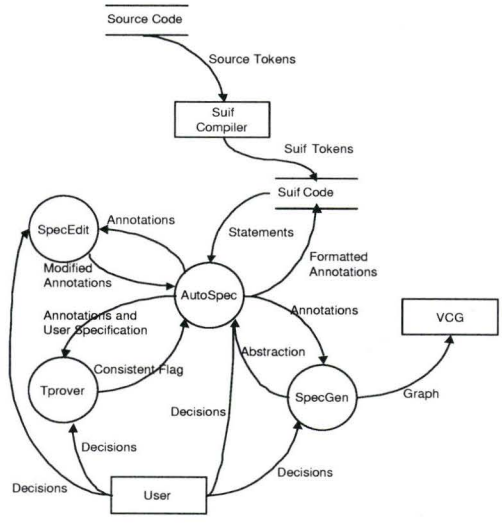

Figure 2.4: Suite of tools for AutoSpec

Conclusion

In project AutoSpec, Gannod describes a suite of tools which can help an inspector extract specifications from source code. His method includes one prototype tool and abstraction process. The specifications produced by these tools make the behavior of the programs more understandable than if the inspectors simply reviewed source code. However, invariants are necessary in the process of this specification recovery. His method can be classified as semi-automatic for this reason.

2.2.3 Loop Invariant

In the previous section, we mentioned that invariants are used in recent research work. Actually, invariants play a very important role in software verification and inspection. Especially in the analysis of source code involved with iterations, understanding the related loop invariants is a tremendous aid to inspectors and maintainers.

A loop invariant for a loop in a program is a proposition composed of variables from the program that is true before the loop, during each iteration of the loop, and

after the loop completes (if it completes). There is always great interest in finding loop invariants by automatic method in reverse engineering. Significant work in this field using compiler techniques was done in the 1970's. However, since then, novel methods concerning automatic loop invariant detection methods were absent until theorem provers and artificial intelligence work were adopted in this research area. Also, dynamic analysis, which means invariants are extracted from execution results, has been implemented successfully in the project Daikon. In this section we will concentrate on recent methods.

The Induction-Iteration Method

The method of induction-iteration was introduced originally by Suzuki and Ishihata in their paper [43] about array boundary checking. In this method they attempt to find the weakest liberal precondition (wlp) of the source code being analyzed. The weakest liberal precondition shows the partial correctness of target programs. For the weakest liberal precondition of specific loops Suzuki and Ishihata present a recursive predicate:

$$W(0) = wlp(\text{Loop-body}, Q)$$
$$W(i+1) = wlp(\text{Loop-body}, W(i))$$

where Loop-body represents the statements in the loop body, and Q is the postcondition of the target program. Then the weakest liberal precondition of one loop is the conjunction of all $W(i)$. Their Algorithm can be described by the following pseudo code (from [43]) in Figure 2.5:

The major concept in this algorithm is to derive an $L(j)$, where $L(j) = \wedge_{j \geq i \geq 0} W(i)$, and if this $L(j)$ is true it implies $W(i+1)$.

Suzuki and Ishihata noticed that for some particular programs this algorithm can not terminate and the set of W(i) can be increased exponentially. For this reason this method can only deal with relatively simple loops.

```
Induction_Iteration() : Success | Failure
{
    i = 0; Create formula W(0);
    while ( i < maximum number of iterations)
    {
        switch
        (TheoremProver ((L(i-1) implies W(i))){
        True: return Success;
        Otherwise: {
            switch (TheoremProver(wlp(<on-entry-to-loop>,W(i)))) {
                True: W(i + 1) = wlp(S,W(i)); i = i + 1;
                Otherwise: return Failure;
                }
            }
        }
    }
}
```

Figure 2.5: Algorithm of generating W(i+1)

Proof Attempts

Researchers noticed that automated methods that blindly search for invariants can result in many failed attempts. One possible way to improve the search is to analyze the failed proof attempts. The basic concept is to direct a successful approach by manually analyzing failed attempts.

In paper [18] Ireland and Stark made considerable steps in proof planning by using a proof approach called *rippling*, a heuristic used often in guiding inductive proof plans. Actually, rippling is the process of rewriting, which converts the target into some known proper form. There rewrite rules are called "wave rules".

The rippling approach performs really well on some simple loops, However, for larger programs, the method needs to be improved. Also nested loops will be dealt with only in the future work in their project.

Predicate Abstraction

Another popular method to derive invariants is based on predicate abstraction, an abstract interpretation technique [6] in which the abstract domain is constructed from a set of predicates over the program variables [14]. One advantage of this method is that it can infer universally-quantified loop invariants, which are important when verifying programs with data types like arrays.

In this method, those predicates are generated from source code using a heuristic method. Given a set of predicates for loops, the process of deriving loop invariants can be reduced to an easier problem of guessing a relevant set of simple predicates. The pseudo code (from [14]) for inferring loop invariants is described in Figure 2.6:

```
<Formula, Stmt> infer (Stmt C, Stmt S) {
    let "{P, I} while e do B" = S;
    Stmt H = havoc(targets(B));
    AbsDomain r = Abstraction(Norm(true, C));
    while (true)
    {
        Formula J = (r);
        Stmt A = "assume e  I  J";
        Stmt B' = traverse("C ; H ; A",B);
        Formula Q = Norm(true, "C; H ; A ; B' ");
        AbsDomain next = r union Abstraction(Q);
        if (next = r) return <J,B'>;
        r = next;
       }
}
```

Figure 2.6: Algorithm of predicate abstraction

We refer the reader to [14] for a full understanding of this algorithm. The basic idea of this algorithm is that new invariants are calculated by original invariants and an Abstraction (Q). Flanagan and Qadeer also discussed some optimization methods in algorithms of the abstraction process, which significantly reduce the number of

predicate clauses that need to be enumerated.

There still are some shortcomings for the method of predicate abstractions by Flanagan and Qadeer. As they said in their paper, the target predicate needs to be designated before abstraction. Also this method is limited by the annotation language. However, their method is a good example of using an advanced artificial intelligence technique for deriving invariants. Their capability of dealing with universally-quantified invariants is another novel feature of this method.

Dynamic Invariant Detection

After reviewing these methods of static analysis to derive loop invariants, we noticed that there is a dynamic invariant detection method as well, by which different invariants are based on different test suites. This procedure is not just for loop invariants, other program invariants can also be detected.

Daikon [9], a prototype tool created by Ernst and his colleagues, demonstrates the feasibility of dynamically detecting invariants. Their approach is to run the target program, examine the values the program computes, check the potential invariants over these values, and report those that are true for the test suite. The major process is described in Figure 2.7 taken from [9].

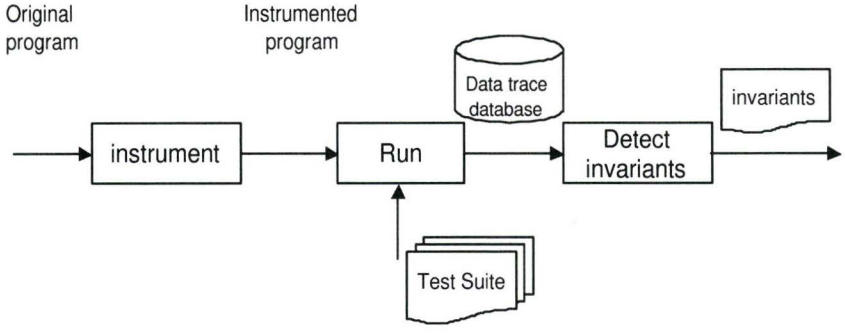

Figure 2.7: Architecture of dynamical detection of invariant

In Daikon project, several techniques are discussed in four major fields. These are:

- Polymorphism elimination
- Redundant invariants
- Comparability
- Return values

Dynamically detecting program invariants expands a programmer's ability to gather information pertinent to software evolution tasks. By combining this approach with existing static analysis techniques, a programmer may be able to gain the best of both the static and the dynamic worlds. Static analysis tends to be sound, but the state of the art does not accurately handle very large programs or all programming languages and features. In contrast, dynamic techniques tend to be more practical in terms of applicability to arbitrary programs and often seem to provide useful information despite their inherent unsoundness [9].

Conclusion

From the 1970's to the present, automatic methods to detect invariants continue to expand. These invariants cannot only be used to understand the behavior of target programs, but they can benifit other related reverse engineering research.

2.3 Conclusion

In this chapter, we discussed several techniques for specification recovery. In those methods we notice that most of it still involves human intervention in the process. Some totally automated methods limits in relatively simple examples. However, the success of this research work demonstrates the feasibility of automatic methods applyingq to extract high level specification from source code. Also they can trigger ideas that may lead us to invent novel methods.

Chapter 3

Literature Survey of Tabular Specifications

This chapter presents a literature survey on functional documentation as proposed by D. Parnas and J. Madey [34]. The discussion focuses on Limited-Domain relations and their tabular representations.

3.1 Functional Documentation

Functional is not used in its vernacular sense, but with its standard mathematical meaning. In mathematics, function means a mapping between two sets of elements (called domain and range, respectively) such that every element of the domain is mapped to exactly one element in the range. If the latter condition is not satisfied, the mapping is called a *relation* [34]. In this paper Parnas and Madey present their idea that all properties of computer systems and their components are seen as a set of mathematical *relations* instead of using vague, imprecise and intuitive language. Also several "functional documentations" are defined to describe the system and components systematically and precisely. However, their goal is to describe the contents of key computer systems - not their form. These documents include the following:

- System Requirements Document

- System Design Document
- Software Requirements Document
- Module Interface Specification
- Internal Design Document

This division into documents is intended to provide "separation of concerns". Different audiences are interested in different documents. For the aim of this research, we want to extract the high level Program Function Table from source code. This documentation is a sub-document of the Module Internal Design document.

3.2 Tabular Representation In Functional Documentation

The Program Function Table describes the effects of a program's execution precisely. To this aim we introduce Limited Domain Relations, and their application to program description and specification.

3.2.1 Limited Domain Relations

A digital computer can usefully be viewed as a finite state machine whose operation consists of a sequence of state-changes. If we are not concerned with the intermediate states of executions, then every deterministic program can be described as a program function whose domain is the set of initial states and whose range is the set of final states [28].

A function can not describe a nondeterministic program. For the simple reason that a nondeterministic program started in one start state may terminate in one of several final states. So relations are more appropriate to represent general programs. Furthermore, we need additional information to describe the set of starting states for which termination can be guaranteed. Here we give some formal structures from [33]:

- A binary relation R on a given set U is a set of ordered pairs with both elements from U, i.e. $R \subseteq U \times U$. The set U is called the Universe.
- The set of pairs R could also be defined by its characteristic predicate, $R'(p, q)$, i.e. $R = \{(p, q) : U \times U | R'(p, q)\}$.
- The domain and range of R can be expressed as follows:

$$Dom(R) = \{p | \exists q[R(p, q)]\}$$
$$Range(R) = \{q | \exists p[R(p, q)]\}$$

- Let U be a set. A limited-domain relation (LD-relation) on U is an ordered pair $L = (R_L, C_L)$, where:
  - R_L, the relational component of L, is a relation on U, $R_L \subseteq U \times U$,
  - C_L, the competence set of L, is a subset of the domain of R_L, $C_L \subseteq Dom(R_L)$.

In detail, LD-relations can be used to describe the effects of program execution if we see set U as the program state set. C_L can also be designed to identify the state set in which termination can be guaranteed.

In our thesis, we discuss only deterministic programs, so the relation can be described as a function. One and only one element in the range can be mapped from an element of the domain.

3.2.2 Tabular Representation

An LD-relation can be represented by conventional mathematical notations. However, the experience of several projects (A-7E, Darlington Shutdown Systems, Bell Labs) was that tabular expressions (function tables) enable us to describe LD-relations in a visual, easy to understand format [19].

There are several advantages to using tabular expressions. Firstly, functions implemented by digital computers exhibit discontinuities, which can occur at arbitrary points in the domain of the function [19]. Also the type of domain and range of

a function can be different in common cases. Those characteristics of conventional mathematical notations make the description of the behaviour of such functions too complex and hard to read. Tabular expressions are an ideal notation to give both precise and readable descriptions of these functions.

Tables can also help in thinking. Though questions of decidability and computational complexity are not affected by using tabular expression, this notation is of great help in practice. For discussion see [19]. When someone first determines the structure of the table, making sure that the headers cover all possible cases, we can then turn our attention to completing the individual entries in the table. The use of the tabular format helps to make sure that no cases are forgotten.

Tables can help in communication. One project might involve people from different backgrounds. People need one universal notation in all these documents, which means that the notation should be easy to learn and understand. Tabular notation is based on *predicate logic*, which is almost universally understood, and the visual aspect of the notation helps people communicate.

Tables help in inspection. For a very big project, by using tables inspection work can be divided and conquered by a systematic procedure. First, inspectors need to make sure that the set of rows and columns are complete with no overlaps. Then they can consider every entry in the table sequentially. Inspectors can therefore take breaks between inspection of cells.

For all these features of tabular notations, we see that extracting tabular expressions from source code can be a tremendous aid to inspectors and maintainers in understanding the behavior of programs. Our aim is to be able to build these tabular expressions automatically from the code.

3.2.3 Program Function Table

We have already discussed that a program can be described by LD-relations. If this program is deterministic these relations are functions. By using tabular expressions we can get a program function table that describes this program.

In [35], function tables are divided into in a variety of forms which include *normal function tables, inverted function tables, vector function tables, normal relation tables*

and so on. Of course, new forms can be invented for particular environments.

In this thesis, we choose *vector function tables* as our output tables. For a given source code, we intend to extract the functions between output variables and input variables.

In [35], a *vector function table*, T, is a table in which the elements of the main grid, G, are terms, the elements of H_1, H_3, ..., $H_{dimentiionality(T)}$ are predicate expressions, and the elements of H_2 are single variables.

Consider this example of C code that computes the sum of the absolute value of two parameters.

```
int sum_of_abs(int a,b)
{
    int sum;
    if (a>0)
    {
        if (b>0) sum = a+b;
        else     sum = a-b;
    }
    else
    {
        if (b>0) sum = b-a;
        else     sum = -a-b;
    }
    return sum;
}
```

The following *vector function table* represents the behavior of that C code.

	$a > 0$		$a \leq 0$	
	$b > 0$	$b \leq 0$	$b > 0$	$b \leq 0$
$sum =$	$a + b$	$a - b$	$b - a$	$-a - b$

$\wedge NC(a, b)$

Note: $NC(a, b)$ represents that the variables a and b are not changed after the program execution.

3.3 Conclusion

In an earlier section we discussed the functional documentation, LD-relations and Tabular representation. We noticed that the tabular expression is an ideal notation to describe effects of program executions. With the characteristic of discontinuity and type difference between domain and range, tabular notations are more applicable than conventional mathematical notations. Extracting these program function tables from source code could be a tremendous aid to inspectors and maintainers to understand the behavior of the program easily. In later chapters we will begin our analysis and methods in the implementation of this tool.

Chapter 4

Analysis

In this chapter, we present the general difficulties in generating function tables from source code in an automatic way, as well as the basis of the methods we used in the tool.

4.1 Automatic Table Generation Difficulties

Reverse engineering presents a different set of very challenging problems from forward engineering same as our attempt to build function tables. If it were easy to automate the generation of function tables from code, it would already be common practice. There are clearly difficulties in doing this, and this section describes the challenges specific to this task.

4.1.1 Main Difficuties

Unstructured Programs

The first challenge is the quality of the code itself. This depends on the programmers who developed the code, except in those cases in which the code was generated automatically or developed in compliance with rigorous coding guidelines. Sometimes the code is not just difficult for analysts to read, but some programmers even use GoTo - like instructions to implement their algorithms. See the example in Figure 4.1

```
    int i=0;
1:
    i++;
    if (i!=10)
        goto 3;
    else
        goto 2;
    goto 1;
2:
    printf("Program Completed.\n");
    exit;
3:
    printf("%d squared = %d\n",i,i*i);
    goto 1;
```

Figure 4.1: Unstructured code

For the example in Figure 4.1, we notice that the equivalent structured program in Figure 4.2 is more readable and more easily analyzed.

Unstructured programs make analysis more difficult, which is one reason structured programming constructs are so heavily recommended. For the purpose of this thesis we assume that the code is reasonably structured.

```
int i;
for(i=0;i<10;i++){
    printf("%d squared = %d",i,i*i);
}
printf("Program Completed.\n");
```

Figure 4.2: Equivalent structured code

Specific Language Difficulties

Obscure syntax and ill-defined semantics can confuse us when we try to understand specific language constructs. Programming languages like C are sometimes more complex than we think. Even if you are an experienced C programmer, there are still some statements that can frustrate you. For example

```
int i=0;
printf("\%d \%d \%d",i++,i++,i++);
```

Many programmers are frustrated and confused when they discover the result of this simple instruction. Even different compiler developers implement it in different ways. The result compiled by Visual C++ 6.0 is "0 0 0" and "2 1 0" is the result compiled by GCC v2.X.X.

Loops

Loops are often a big problem in reverse engineering. Partially because the definition of recursive loops used in operational semantics of imperative language is not compositional. Obviously loops are really convenient and necessary for searching, sorting and many other computations. The difficulty is how we can derive the functionality from the instructions in the loop body.

Another challenge is to determine the termination of loops. The result does not only depend on the algorithm of related loops. Termination may also be influenced by processor architecture. For example from numerical computation in Figure 4.3, we can see that this mathematically non-terminating loop will finally stop after underflow occurs.

Implementation Reality

Difficulties in implementation are easily overlooked when we discuss the difficulties in analysis. Many of these problems became apparent while developing our analysis tool, and we will discuss them later in section 4.3, 4.4, and 4.5.

```
float i=1;
while (i=0){
    i = i / 2;
}
```

Figure 4.3: Floating underflow

Gaps in human and machine

Manual extraction of function tables from code is time consuming, but analysts have been doing this successfully for more than 15 years. Automating this process has proved to be difficult, partially because humans are flexible in their approach and can tailor the basic process to fit the current problem in ways that are not understood well enough to automate.

4.1.2 Solutions

The first two difficulties really encourage us to do more preparation before we actually do the analysis. It therefore makes sense to conclude that a "clean-up" stage should be added into our process. In [2] Breuer and Lano [1] state that translating the source language into a more structured language is an essential preparation that results in the code being restructured to some extent to reveal its "essential structure". The aim is to structure the code so that each statement has clean semantics and corresponds to a meaningful fragment of a program specification.

We will discuss problems 3 through 5 later in section 4.3, 4.4, and 4.5.

4.2 Overview of Analysis

Above we discussed the difficulties we faced in developing the analysis built-in to our tool. In later sections we will discuss a more detailed analysis related to main difficulties. In section 4.4 we present the major algorithm used in our tool.

As we mentioned in Chapter 1, the high level imperative language C is our analysis

target. Based on the difficulties we discussed in the previous section and other reverse engineering experience documented in [2], including a clean-up stage is a really helpful preparation for static program analysis. As we said, a well-structured program with clean semantics makes our process more efficient. In the chapter dealing with tool implementation we discuss code lists and how this data structure helps us transform the original code into a well-structured equivalent.

In order to extract a vector function table from a specific program the main work of the tool is to derive the functions of output variables in terms of input variables under all conditions. We know that every terminating program can be described by a mathematical function. So given a program P we let $\mathcal{X}$ be the set of all input variables and $\mathcal{Y}$ be the set of all output variables. The function $\mathcal{F}$ corresponding to P can be expressed by

$\mathcal{Y} = \mathcal{F}(\mathcal{X})$ and $\mathcal{F} = (f_1, f_2, ..., f_n)$

So we can see that for every single output variable $Y_i \in \mathcal{Y}$, $Y_i = f_i(\mathcal{X})$. If we can derive all the functions f_i from the program, we can describe those functions by the appropriate tabular expression.

Imperative language programs comprise simple assignments and control statements which contain conditional statements and loops. At this stage of our work we have not considered exceptions and interrupts. Also in order to make our analysis easier we deal with only integer types, and do not handle other complex data structures. We also consider sub-programs as future work too in this thesis.

We use the following grammar to describe the target code:

Code	::=	$Blocks^+$;
Block	::=	Assignment | IF_stmt | LOOP_stmt;
Assignment	::=	VarID := Expr;
IF_stmt	::=	IF Predicate THEN Code ELSE Code;
LOOP_stmt	::=	WHILE Predicate DO Code;

From the grammar we can see that there are three kinds of blocks we should deal with. For each kind of block we are interested in the final $\mathcal{F}$ instead of how each

is implemented in the process. We will discuss the analysis of each kind of block separately.

Before that discussion, we present a picture to describe the process overview of our tool. In Figure 4.4, we see that we take C code as our input, which is then parsed and translated into an intermediate language. Then, by loop elimination and evaluation techniques we derive the functionality of the target code.

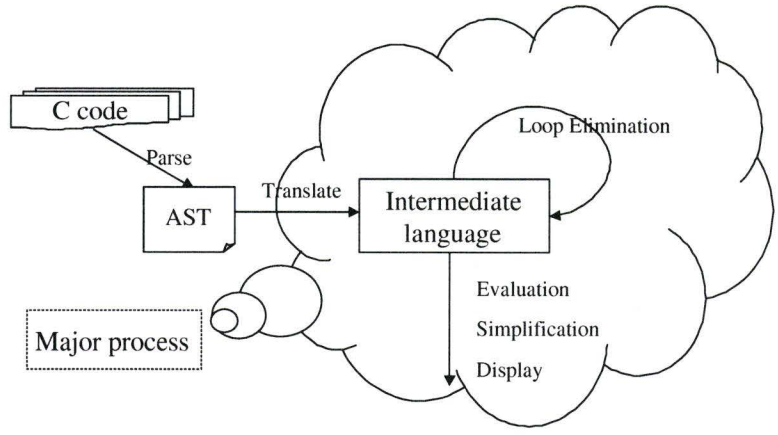

Figure 4.4: Analysis process

4.3 Simple Assignments

Assignments are a core component of any imperative language. They can be expressed by $< Variable >::=< Expression >$. Intuitively, for every single assignment we describe the variable' immediately prior to execution as value as 'V, which is known as "V before". From the operational semantics of the assignment we get the value V' after execution, which which is known as "V after". So what is V'? The answer is derived from the *Expression* on the right side of the assignment. In evaluating Expression, we use the "value before" of each variable. We also note that the semantics of sequential assignment are compositional.

From the intuitive analysis presented above we know that for every simple assignment we have that $< Variable >::=< Expression >$. Whatever the variable name, we represent the left variable by $Y_i \in \mathcal{Y}$. The value of the expression on the right will be the new value of Y_i, so $Y_i = f_i(\mathcal{X})$. For example, consider the very simple assignment in C shown below:

```
int x,y;
y=2*x+15;
```

Figure 4.5: A simple computing example

In Figure 4.5, we see that the value of $2\times'x+15$ will be the value of y'. This function can be expressed by $f_i = \lambda x : \mathcal{Z} \ . \ 2\times x+15$. So we can see that it is never difficult to derive the function f_i from a single assignment. However, we are more interested in code with sequential assignments. Here we give an example with sequential assignments.

```
int a,b,c; a=0;        (1)
b=a+100;               (2)
c=c+b;                 (3)
a=c;                   (4)
```

Figure 4.6: An example with multi-assignments

As Figure 4.6 shows, there are three variables which are all in both $\mathcal{X}$ and $\mathcal{Y}$. It is easy to ascertain that they are in both X and Y since they occur on the left and right hand sides in the assignments. We have not paid any attention to temporary variables which will affect the results. We assume that all these variables have their initial values as 'a, 'b and 'c. According to the four statements in Figure 4.6, we define four functions f_1, f_2, f_3, f_4. and we can see that

$f_1 = \lambda x : \mathcal{N} \ . \ 0$
$f_2 = \lambda x : \mathcal{N} \ . \ x+100$

$f_3 = \lambda x, y : \mathcal{N} \ . \ x+y$
$f_4 = \lambda x : \mathcal{N} \ . \ x$

In order to account for every step of the code we show the result for every statement:

after (1): $a' = f_1()$
after (2): $b' = f_2('a, 100) = f_2(f_1(), 100)$
after (3): $c' = f_3('c,'b) = f_2('c, f_2(f_1(), 100))$
after (4): $a' =' c = f_2('c, f_2(f_1(), 100))$

In the above expression, all the before and after signs are used appropriately in every statements. From the step results of every statement we find that the final functions for each output variable can be composed from the results in previous steps. Theoretically , the final result is correct because the operational semantics of simple assignments are compositional.

We have seen that in order to derive the correct value after execution of several pure assignment blocks we just need to record all changes in every assignment. In our algorithm, we build an assignment list in which every entry contains the current value of a variable and every variable can only have one entry in the list.

Another problem concerns expression simplification. Consider this example:

```
int a,b;
a = a + b;
b = a - b;
a = a - b;
```

Figure 4.7: An example of swap

The statements in Figure 4.7 result in swapping the values of variables a and b. Rather than the more common swap function which uses a temporary variable, here we prefer to use this method when the system does not have much memory available.

So the function which describes the above program is:

a' = 'b
b' = 'a

However, from the steps we used above, we can only get that

a' = ('a+'b)-(('a+'b)-'b)
b' = ('a+'b)-'b

Although the final result is equivalent to the result we wanted, the former result is preferable for readability. This simple example tells us that a simplification step would improve our final result. Since this is not focus of the research in this thesis, we just used a Computer Algebra System to help us implement this simplification.

The result of this analysis is embodied in the algorithm to extract the function that represents the simple assignment block is presented below:

1) extract input and output variables from the assignment into $\mathcal{X}, \mathcal{Y}$;
2) get the next code statement which should be dealt with;
3) put the left value variable name into the assignment list, if it exists then use the existing one;
4) put the right value expression into this assignment entry in the list and substitute all variables in the expression by all their values in the assignment list.
5) go to step (2) again until the end of code.

Figure 4.8: Algorithm of straight line code

4.4 Alternation

Alternations are frequently used in imperative languages. As we defined in overview they have the grammar:

*IF Predicate THEN Code*1 *ELSE Code*2.
Predicate here is a boolean expression which determines which branch the execution will take. *Code*1 and *Code*2 represent those two branches.

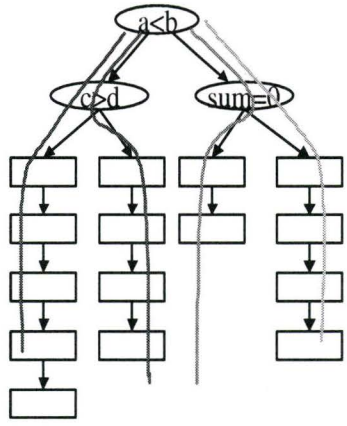

Figure 4.9: Execution branches under given conditions

In the previous section we presented an algorithm which deals with sequential assignment code blocks. For any execution branch it is not difficult to extract the final function for every output variable, modeled on the algorithm for sequential assignment blocks (see Figure 4.9). This provides us with the components of function $\mathcal{F}$. From this point on we can regard a code block with one condition statement as two sequential assignments blocks in which some parts overlap. From each of these blocks we can derive one component of the function $\mathcal{F}$. This is the case when there is one condition in the code block. For more conditions we can split the block into more blocks. If there are n conditions, then the number of blocks would be 2^n.

Here is another example:

The program in Figure 4.10 computes the value of sum to be the sum of positive values of a and b. Since there are two conditions in the code, we copy the code into four blocks as shown below:

So from those four code blocks we can extract four functions $\mathcal{F}_1$,$\mathcal{F}_2$,$\mathcal{F}_3$,$\mathcal{F}_4$ according to the four possible preconditions. We can see that the semantics of this

```
int sum,a,b;
sum = 0;

if (a>0) sum += a;
if (b>0) sum += b;
```

Figure 4.10: Sum of positive parameters

1.	2.	3.	4.
$\{a > 0 \wedge b > 0\}$	$\{a > 0 \wedge b \leq 0\}$	$\{a \leq 0 \wedge b > 0\}$	$\{a \leq 0 \wedge b \leq 0\}$
sum=0;	sum=0;	sum=0;	sum=0;
sum=sum+a;	sum=sum+a		
sum=sum+b;		sum=sum+b;	

function conform to the semantics of a tabular expression that represents function $\mathcal{F}$. This means that there will be no difficulty in expressing the function in a tabular representation.

Here we modify our algorithm to be suitable to code with conditions.

4.5 Loops Overview

Loops are the biggest challenge in our analysis. The operational semantics of loops may be defined by recurrence equations, which makes it hard to derive the explicit function from loop body. In this section we discuss the main problems we faced in our analysis of loops, and present our partial solution to the analysis of loop structures.

As in previous sections, we give two typical examples of loops in Figures 4.12 and 4.13.

The reason we give two examples here is that we want to begin from some specific loop examples and identify the associated recurrence relations and rules and use them to come up with general solutions.

We notice that for both of these examples the number of iterations is controlled

1) Extract input and out variables from assignments into $\mathcal{X}, \mathcal{Y}$;
2) Get next sentence which should be dealt with from code;
3) If this sentence is a condition statement, do 4 else do 5;
4) If the condition is decidable already, then get next sentence from the relative branch and do 2. Or record the condition and copy current assignment list and deal with every branch respectively.
5) Put the left value variable name into assignment list, if it exists then use the old one;
6) Put the right value expression into this assignment entity in list and substitute all variables in expression by all value in assignment list.
7) Go to step (2) again until the end of code.

Figure 4.11: Algorithm of alternations

by logical conditions of output variables involved in guards. Within the loop body there are still code blocks which may contain the control statements we mentioned before. However, if we take the inner most loop as our starting-point all the loops may not show up. So, if we can make our algorithm robust enough, nested loops should not be an exception. We also notice that semantics of the assignments in the loop body are totally different from the ones outside. So we will devote some sections in the thesis to discuss the problems thoroughly.

4.5.1 Main Problems

Problem Description

Our aim is to eliminate the loop structure in the source program. In other words, we aim to represent the behaviour embodied in a loop as straight line code. To do this we need to know the values of all variables changed by the loop, at the termination of the loop. This of course assumes that the loop does terminate. To help us understand the explicit functions implemented in the program, we choose to represent all functions using recurrence equations. To do this we assume we know the values of all variables after n-1 iterations of the loop body. So, if we can describe the value of each variable in the loop after n iterations in terms of values of all the variables after n-1 iterations,

```
/*sample sum*/
    int i,num,sum;

    sum=i=0;
    while (i <= num)
    {
        sum += i;
        i++;
    }
```

Figure 4.12: Sum of consecutive integers

we can define a recurrence relation for the value of each variable. Then we can use a pattern matching algorithm to get a more readable function definition. Before we go further with our algorithm, we need to deal with some important details. Note that analysis can be made much more complex if we have to take into account unpredictable programming habits and styles.

Multi-Assignment for Single Variable

First thing we should think about is multi-assignments in every execution branch. For a complex system which has complicated functionality there certainly can be multi-assignment to one variable during any execution branch. Take an example like this:

The above program in Figure 4.14 computes the sum of the maximum value in every row. Actually this example also demonstrates other interesting points like nested loops and conditions in the loop etc. However, in this section we just consentrate on multi-assignments.

So when we are dealing with new assignments in any branch we have to make sure there is a unique assignment entity for each variable in the assignment list data structure. For any assignment in the code, if there is no assignment entity according to the variable name of the left value we put a new entity into the assignment list. Otherwise we just use the old entity. For the right value of this assignment which

```
/*sample maximum*/

int max(int a[], unsigned int num)
{
    int c,i;
    i=0;
    while (i<num) {
        if (c<a[i])
            c=a[i];
        i++;
    }
    return c;
}
```

Figure 4.13: Maximum of an array

should be an expression, we should substitute all variables in the expression if there is an assignment entity which has the same name. Then all variables in this right value expression are just all the variables after n-1 iterations. After dealing with all assignments in the code in every execution branch, we will have the final assignment entity list in which every variable has the expression represented by variables after n-1 iterations and also there is one and only one entity corresponding to every output variable.

Mathematically, with this process we build an assignment list in which there are entities representing the function F_1 for every output variable. Furthermore, we notice that this process can use our simple assignment algorithm directly except that this function F_1 will not be the final function for this variable. It just shows the functionality of related variables after n iterations in terms of variables after n-1 iterations.

```
int max, sum, c[][], i, j;

max=0;
sum=0;
for (i=0;i<m;i++)
{
    for (j=0;j<n;j++)
    {
        if (max<c[i][j])
        max = c[i][j];
        sum += max;
        max = 0;
    }
}
```

Figure 4.14: Values in multi-assignments

Recurrence Relations

The next detail we consider is the recurrence equation for every output variable. As we said earlier, there is one and only one entity in the assignment list corresponding to every output variable and the right value expression is represented by all variables after n-1 iterations. In the example of computing the sum of consecutive integers in Figure 4.12, we notice

```
sum += i;
```

We define a function f here to represent just this statement. So
$sum' = f('sum,' i)$ *in which* $f = \lambda\ x, y : \mathcal{N}.$ x+y;
Unfortunately this is only correct when this statement is outside loops, which also means that the above function just shows the relationship of variables after n iterations in terms of values of variables after n-1 iterations. However, what we want is the relationship of values of variables after termination of the loop (assuming it does terminate), in terms of values of variables immediately prior to the loop.

In order to go further with our analysis of loops, here we explicitly state our assumption that the loop we are analyzing terminates after n iterations. We will discuss the termination problem in later sections. We use recurrence equations to derive the function that describes a loop's behaviour, as follows. We define $VarID_n$ to mean the value of VarID after n iterations.

For the example of computing a sum we can derive these recursive functions:

$$sum_n = \begin{cases} 'sum & ; \quad n = 0 \\ sum_{n-1} + c[i_{n-1}] & ; \quad n > 0 \end{cases}$$

and

$$i_n = \begin{cases} 'i & ; \quad n = 0 \\ i_{n-1} + 1 & ; \quad n > 0 \end{cases}$$

For these recurrence equations of output variables *sum* and i, when n equals 0 that means no instructions in the loop body have been executed even for one time. The loop is then equivalent to *skip*. Otherwise the result depends on the previous values.

Under the semantics of the loop in an imperative language, we can derive recurrence equations for each output variable. In later sections we will show more situations analyzed by this representation and discuss how we use this representation in our analysis.

Conditions in Loop

Another issue is IF statements in the loop body. Because we begin from the innermost loop from source code we want to deal with. There is no loop statement in current code. However, we still need to think about IF statements in a loop. Just as in the example of computing the maximum value in Figure 4.13. As we discussed in the section concerning conditions, the IF statement can split the execution path into different branches. For every branch we provide an assignment list corresponding to it.

1.	2.
$\{max \leq c[i]\}$	$\{max > c[i]\}$
max=c[i];	skip;

We can see that the value of max after n iterations could be either c[i] or the old max (after n-1 iteration). Fortunately we still can use recurrence equations to record output variables in this situation by giving more conditions. See this:

$$max_n = \begin{cases} 'max & ; \quad n = 0 \\ c[i_{n-1}] & ; \quad (n > 0) \wedge (max_{n-1} < c[i_{n-1}]) \\ max_{n-1} & ; \quad (n > 0) \wedge (max_{n-1}) \geq c[i_{n-1}] \end{cases}$$

We see that the result of output variable max is more complex than the variables we mentioned in the previous section for the reason that there is one more condition in the loop body. One thing we should pay attention to is that we use the previous value of a variable itself if there is no relative assignment corresponding to this output variable.

Let me recall an example with similar conditions outside loops. Think about this:

```
int max,a,b;

if (a>b)
    max = a;
else
    max = b;
```

Figure 4.15: An example with simple alternation

Using the algorithm we discussed ealier, we can represent the program in Figure 4.15 by the table in Figure 4.16.

However, we also can present this program by an expression of the form:

$max' =$	$'a >' b$	$'a \leq' b$
	$'a$	$'b$

Figure 4.16: Table representing Figure 4.13

$$max = \begin{cases} 'a & ; \quad 'a >' b \\ 'b & ; \quad 'a \leq' b \end{cases}$$

It may seem confusing that conditions in a loop body have been dealt with totally differently from those outside of loops. We know that conditions outside loops lead to more sections in tables. That is why we use tables to make our expressions more readable. However, what we did with conditions in loops is that we present all semantics of conditions in one recursive expression. So does this hurt the readability of the tabular expression? Of course not. Loops are complex structures in imperative languages. They can be used in very complicated functions. Without recurrence equations it is really hard to express those functions in other ways. So we describe all those conditions in one function expression and deal with it by pattern matching method. Then, we can perform further analysis by describing complex functions in this simple way.

Until now all we endeavored to do is to present variable functions in an explicit way. However, this not what we want at the end. We next consider whether we can extract some more common expressions from those recurrence equations. From all the examples we gave before, all those functions can be represented by more meaningful expressions like "sum" or "maximum".

For the very simple example of i, we give the i definition to:

$$i_n = \begin{cases} 'i & ; \quad n = 0 \\ i_{n-1} + 1 & ; \quad n > 0 \end{cases}$$

with no doubt, this function can be rewritten as a more meaningful expression, so that:
$i_n =' i + n$

So our question is how can we get this kind of result in general and how many similar kinds of functions can we recognize.

In the next section we discuss how to (partially) fulfil our expectations by using a method of pattern matching.

Pattern Matching

Pattern matching plays a significant role in our method. Once we derive the recurrence equation for every output variable, we already have a description of the functionality of the loop for every output variable. However, we need a more readable definition for such functions.

In this section we discuss how we classify the recurrence equation into more readable expressions.

Let us focus on some simple examples to begin our analysis. Consider the example of the index i again.

$$i_n = \begin{cases} i_0 & ; \quad n = 0 \\ i_{n-1} + 1 & ; \quad n > 0 \end{cases}$$

As we said before, we try to find a way to transform this expression into a more readable format $i_n = i + n$. From this example we can find some features which are essential for us to identify the explicit function. One is that the main relation of the right value expression is a binary function '+'. Also one argument is itself and the other is a constant value. Based on those features we can extract a recognizable form for this function. So, if we can organize those features into a pattern, we can make the transformation automatically. Thus, we come up with the pattern style shown in Figure 4.17.

The essence of this classification is that we record relevant features and ignore trivial details. Let us consider a more complex example to test our method. Consider the recurrence relation we derived for maximum, shown in Figure 4.18.

We begin with those features which can help us identify the explicit function. The first one is that there is one more condition besides (n > 0). We notice that for this condition the main relation is > and the first argument is itself and the second

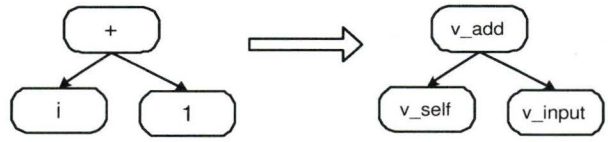

Figure 4.17: Pattern for addition

$$max_n = \begin{cases} 'max & ; \quad n = 0 \\ c[i_{n-1}] & ; \quad (n > 0) \wedge (max_{n-1} < c[i_{n-1}]) \\ max_{n-1} & ; \quad (n > 0) \wedge (max_{n-1}) \geq c[i_{n-1}] \end{cases}$$

Figure 4.18: Pattern for maximum

argument is related to the output variable (we call all expressions in which there are output variables, *output-related*). The second feature is that both assignments are related to the variables in the condition. To make it clear, we pack all those features together into the pattern shown in Figure 4.19.

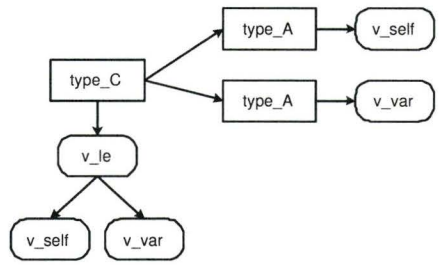

Figure 4.19: Pattern for maximum

In these patterns, the "type_A" and "type_B" connectors are used to indicate whether the connected entities are 'assignments' or 'conditions', respectively. **v_var** stands for the output variables which are assigned by new values in the execution of loop body. And **v_self** means the variables which are referred in the assignments of themselves.

From these two examples we need to identify all those features which are important in developing the final result. We start by defining all the specific entities by assigning all relevant types. Then we classify the targeted recurrence equation into a pattern by related features. Every pattern is a structure composed of particular types. There also should be a patterns database in which all patterns are stored. We use designated rules to classify those recurrence equation into a pattern structure and then compare them to determine whether it is can be recognized and be transformed into a more readable definition. The pattern matching process can be shown in Figure 4.20.

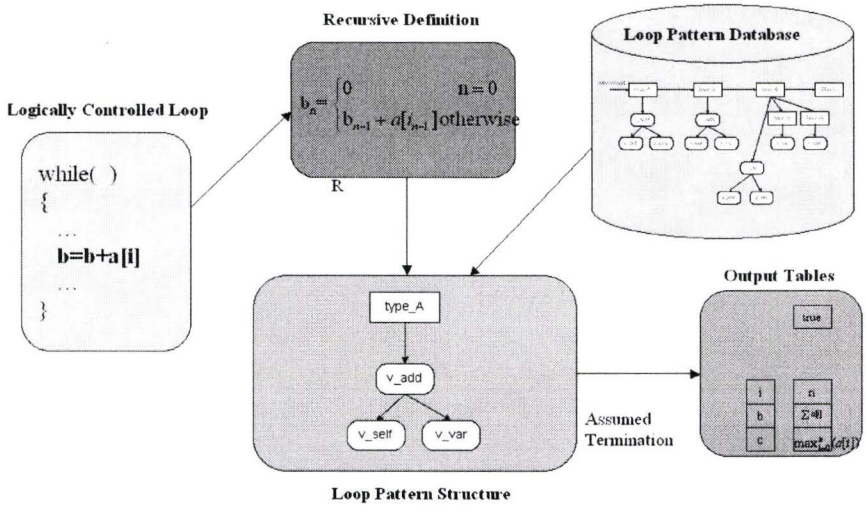

Figure 4.20: Procedure Pattern Matching

Iteration Count

We did not mention one thing which is really important to show the description of functionality. That is whether we can denote the final iteration count n, if the loop terminates. In our analysis until now we always assumed that our target programs will terminate. Like earlier sections we still begin from a simple example to see whether we can deduce a more general solution. Consider the example in Figure 4.21.

```
int i,begin,end;
...
i=begin;
while (i<end)
{
    ...
    i++;
}
...
```

Figure 4.21: Iteration count

For the above program, we ignored trivial detail which was not essential to this section. Our aim is to derive the value of n from the program. We know that the predicate (i<end) can determine whether the loop terminates. Variable end is an input variable here and from an earlier section we know that

$$i_n =' i + n$$

For this particular example $i_n = begin + n$, so if we substitute the variable i in predicate (i<end), we get that

$$begin + n < end \Rightarrow n < end - begin$$

However, in other cases it may not be as easy to derive the exact expression for n. Commonly we still assume that there exists a value of n such that after n iterations the guard is true and after n+1 iterations the guard becomes false.

4.5.2 Loop Elimination Algorithm

This section presents the detailed algorithm we use for loop elimination. An assignment list will record all the assignments in the loop body. The list is initially NULL.

1) Extract input and output variables from assignments into $\mathcal{X}, \mathcal{Y}$.
2) Walk through the whole code block and record all loop.
3) Identify a loop to "eliminate", starting with the inner-most nested loop.
4) For the identified loop, build an assignment list from the code blocks in the loop body using the algorithm that applies to straight-line code.
5) Build recurrence equations for every output variables in this loop.
6) Perform pattern matching for every recursive function.
7) Evaluate the variables in the loop guard and to try to find the iteration count if possible.
8) Go back to 3 until there is no loop.
9) From the beginning of the code blocks, perform the straight-line code algorithm again.

Figure 4.22: Algorithm of nested loops

4.5.3 Nested Loops

In fact, in the above algorithm in Figure 4.22, we already mentioned how to deal with nested loops. We notice that for the target code blocks, we first build some structure corresponding to every loop in the program. Then we deal with the first inner loop first. In the process, we try to use the pattern matching method to change the loop structure into simple assignments. Later we deal with outer loops using the same algorithm. Consider the nested example in Figure 4.23.

This program computes the maximum value of the sum of every row of matrix c. By our algorithm in Figure 4.22 we deal with the inner loop first. We can find two patterns defining sum and product. So we can derive the "program" in Figure 4.24, equivalent to the original code.

We can see that by using pattern matching the inner loop has been eliminated successfully. Then we can use our algorithm again to cope with the outer loop.

4.6 A more detailed example

After discussing these problems we conclude by walking through the final algorithm we use to analyze C source code. To understand the detailed steps we present a

```
int max, sum, c[][], i, j;
max=0;sum=0;
for (i=0;i<m;i++)
{
    for (j=0;j<n;j++)
        sum += c[i][j];
    if (max<sum)
        max=sum;
    sum=0;
}
```

Figure 4.23: An example with nested Loops

description of the whole process and how we deal with some specific examples.

Consider the example in Figure 4.13. The program calculates the maximum value of every element in an array. According to our algorithm we need to eliminate the loop structures, from the inner-most loop to outer-most loop. In this example there is just one loop in the source code, so we eliminate only this loop.

First, we analyze the assignments in the loop body, and then we can give the recurrence equations for variables i and c, as shown in Figure 4.25.

Second, we want to represent those functions in normal forms other than the recurrence equations in Figure 4.25. As we discussed in earlier sections we use a pattern matching technique to do this transformation. An abstraction step is needed to extract the essential functionality from this recurrence equations as shown in Figure 4.26.

After pattern matching, hopefully we can get an equivalent assignment (Figure 4.27) for every output variables in that loop, which is the major concept behind loop elimination.

At the end, we only have simple assignments and alternations in the source code without any iterations. Then, we perform the evaluation procedure to get the final functions for every output variable. This can then be represented by the tabular expression shown in Figure 4.28.

```
int max, sum, c[][], i, j;
max=0;sum=0;
for (i=0;i<m;i++) {
  {
```

sum += $\sum_{j=0}^{n-1}$c[i][j];j = n-1;

```
  }
  if (max<sum)
      max=sum;
  sum=0;
}
```

Figure 4.24: Inner loop eliminated

$$i(n) = \begin{cases} i(0) & : \quad n = 0 \\ i(n-1)+1 & : \quad n > 0 \end{cases}$$

$$c(n) = \begin{cases} c(0) & : \quad n = 0 \\ a[i(n-1)] & : \quad n > 0 \wedge c(n-1) < a[i(n-1)] \\ c(n-1) & : \quad n > 0 \wedge c(n-1) \geq a[i(n-1)] \end{cases}$$

Figure 4.25: Expression by recurrence equations

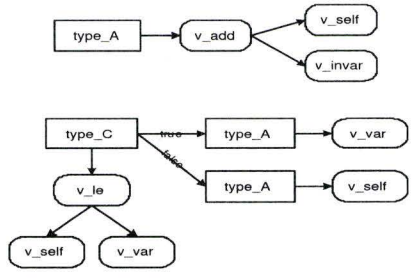

Figure 4.26: Pattern style expressions

$$\mathrm{i(n)} = \mathrm{i(0)} + \mathrm{n}$$
$$\mathrm{c(n)} = \mathrm{c(0)}\ ?>?\ max_{k=0}^{num-1}(a[k])$$

Figure 4.27: Equivalent assignments

	$true$
$i =$	num
$c =$	$max_{k=0}^{num-1}(a[k])$

$\wedge NC(a, num)$

Figure 4.28: Tabular expression representing code in Figure 4.13

Chapter 5

Requirements

This chapter identifies the requirements for our tool.

5.1 Assumptions

For any unexplored field, one is commonly advised to begin with assumptions which simplify the analysis domain. Our tool is no exception. The major object of our project is to construct automatic derivations of tabular expressions. Using simplifying assumptions will prevent us from being sidetracked from the main issues.

We know that the C programming language is a complex high level imperative language which is very popular and widely used in industry. Considering the complexity of C, and our time constraint, we make some simplifying assumptions instead of dealing with the whole C language. In our analysis domain, we take the data type **int** as the only type we handle. More data types can be postponed to future work.

Another assumption we make is to deal only with well-structured programs. The experience of the Shutdown System of the Ontario Power Generation shows that non-modular programs cause really frustrating problems for tabular expressions even when generated by hand [1]. So we restrict our target domain to well-structured programs. Intuitively, there should not be any GOTO statements in code blocks.

5.2 Input

In all static program analysis tools one essential thing is to identify the input domain. Usually developers prefer abstract syntax trees as their input. A number of tools have been developed for static analysis of abstract syntax trees. Our approach, by contrast, uses code list as input.

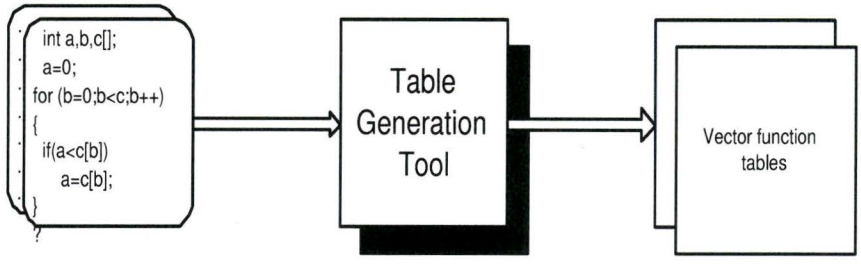

Figure 5.1: Table generation tool

5.2.1 Abstract Syntax Tree

As stated above, most research work on code static analysis uses abstract syntax trees as the input. For a programming language, syntax is concerned with the structure of programs. Concrete syntax is the representation of phrases as strings. It is concerned with the readability and ambiguity of a language which is formally defined by its words and its sentence structure. In contrast with concrete syntax, abstract syntax focuses on the basic structure of the language. An abstract syntax tree is commonly built from the syntax analysis by the parser. Researchers usually prefer to take the abstract syntax tree as the input instead of specific concrete representation strings.

As an example of an EBNF definition of program expressions, consider Figure 5.2.

For a simple arithmetic expression like "9+(8-6)", the tokens are 9, +, (, 8, -, 6,). The syntax tree will be as in Figure 5.3

```
expr   =    term
       |    expr '+'  term
       |    expr '-'  term ;
term   =    factor
       |    term '*' factor
       |    term '/' factor ;
factor =    number
       |    '(' expr ')' ;
number =    ['0'..'9']+;
```

Figure 5.2: EBNF definition of expressions

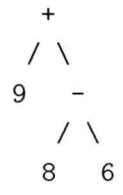

Figure 5.3: An example of an abstract syntax tree

5.2.2 Code List

In Chapter 4 we discussed the process overview of our analysis. We noticed that C source code is parsed and translated by LCC into an intermediate language, called code-list [25]. Our analysis and operation is then based on this intermediate language. In accordance with the aims of our research, LCC is an open source compiler for multiple architecture. There are several advantages in choosing this intermediate language rather than abstract syntax trees:

- cleaner semantics than with the C language
- same representation for alternations and iterations resulting from different code structures
- providing a base that makes it easier to implement evaluation rules

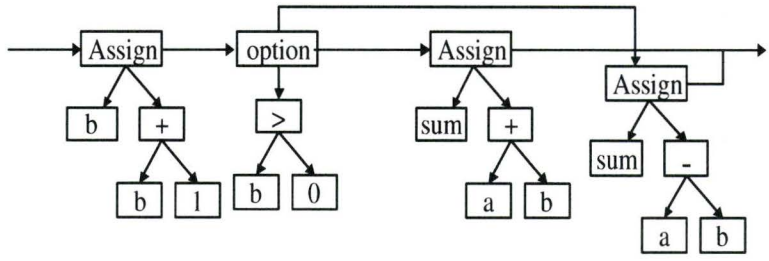

Figure 5.4: Code list representation of alternations

- the analysis is similar to abstract syntax trees analysis, so it is easy to apply it to other high level languages.

Consider, for example, an alternation statement of the form:

```
int a,b,sum;
if (++b>0) sum=a+b; else sum=a-b;
```

Notice that the structures represented by the code list of this code extract, as shown in Figure 5.4, have the same semantics as the original C program.

By means of such structures, code lists can provide programs coded in C with clear semantics. Notice also that equivalent programs in different formats can have the same code list representations.

For example, the two loops:

```
int sum,i,a[];
for (i=0;i<n;i++) sum += i;
```

and

```
int sum,i,a[];
i=0;
while (i<n) {sum = sum + i;i++;}
```

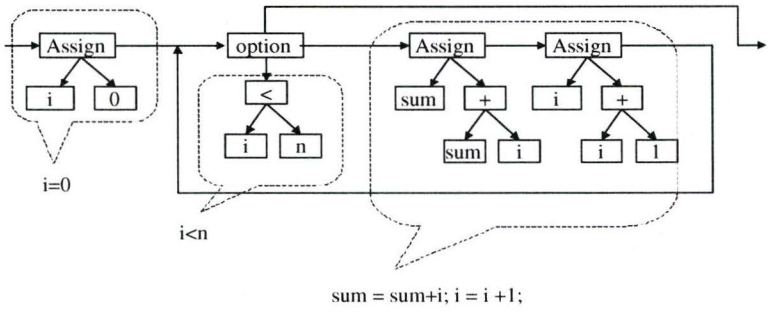

Figure 5.5: Code list representation of iterations

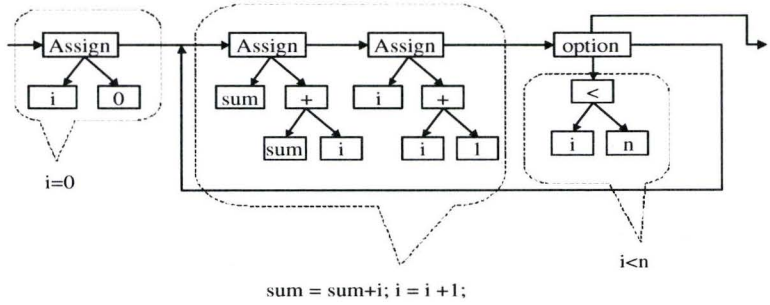

Figure 5.6: Code list representation of DO iteration

They can both be described by the code list in Figure 5.5, making it easier for us to focus on the structure rather than various kinds of loop forms.

Of course, for loops with different semantics, there is a difference in representation. Consider, for example:

```
int sum,i,a[];
i=0;
do {sum = sum + i; i++;} while (i<n) ;
```

We can see a difference in the sequence for this in Figure 5.6, compared to Figure 5.5, reflecting the semantic difference.

In summary, code lists provide a good intermediate language for the analysis of C code. In the implementation of our tool, we will extend the syntax of this intermediate language for the purpose of loop elimination. We will also use this language to express the final values of output variables.

5.3 Interfaces

The format of the table generation tool will be a simple executed file. The application input which records the C source code is designated by command line arguments. Some other options used for debugging code list structures will be considered too.

The user interface produced by our tool is as follows:

```
TableTool -target=table [-showlist] <C Code file>

<>: means this argument is required
[]: means this argument is optional
-target: identify the target result, we keep the old interface of
compiler.
-showlist: display the code list we want to analyze.
C Code file: designate the input C file
```

5.4 Output and display

The output of the function tables will be represented in text strings for the purpose of display. Graphic outputs are not the major effort in this research, though it could increase the readability of tabular expressions. A readable output with clear semantics will be acceptable for this version of our tool.

5.5 Other Requirements

Because of time constraints, we dealt only with a subset of C. Our tool will require extensions at a later date.

Chapter 6

Tool Implementation

In this chapter we discuss how our table generation tool is developed. Based on the algorithm we abstracted in Chapter 4, we now present more implementation details.

6.1 Data Structure

In the development of our tool, a number of data structures are used to make our algorithm efficient and extensible. In this section we will list all the major data structures and their functions. Because we use the open source compiler LCC to help us with the parsing, we have adopted some data structures from LCC [25].

In this section we will introduce all the major data structures and their supportive functions, if any. To make our description more understandable and systematic, we provide a data structure schema to display our design intentions. So for every structure we will have the following four sections: *definition, description, elements,* and *supports* .

Also we categorize all data structures in several groups, such that those in each group perform similar processes.

6.1.1 LCC Data Structures

A number of data structures were designed in the LCC project. We list all those that are involved in our tool. In the requirements documentation, we designate the

code lists as the major input. The structure of the code lists and their elements are discussed in this section.

Code

Definition:

```
typedef * struct code {
    enum { Blockbeg, Blockend, Local, Address, Defpoint,
           Label,    Start,    Gen,   Jump,    Switch
    } kind;
    Code prev, next;
    union {
        struct {
            int level;
            Symbol *locals;
            Table identifiers, types;
            Env x;
        } block;
...
        Node forest;
...
    } u;
} Code;
```

Description:

Code is constructed as a bidirectional list. There is just one code list corresponding to every input program. In our analysis we have to walk through the whole code list to inspect the input program.

Elements:

The element **kind** identifies the class of this code entity. The respective semantics of these kinds lead to different operation procedures.

The elements **prev** and **next** connect all the code into a bidirectional link. They are sequentially dependent from the beginning to the end.

The element **u** records the actual information about this code entity. Like the member **forest** all simple assignment descriptions are stored in this structure.

We will talk about the **Node** structure later.

Supports:

```
Code code(int kind);
```

The function **code** creates a new code data structure by accepting parameters of `kind` which designate the class of this code entity.

Node

Definition:

```
struct node {
    short op;
    short count;
    Symbol syms[3];
    Node kids[2];
    Node link;
    Xnode x;
};
```

Description:

Node is the essential structure for building the code list. Conceptually, it is an extensive abstract syntax tree. All simple assignments and control statement information are recorded in the code list.

Elements:

The element **op** identifies the operation code of this node.

The element **syms** records the symbol information related to this node.

The element **kids** points to the next level nodes if any exist. For example, if this node is designated as the operator ADD, then the element **kids** points to the two arguments of this operator.

The element **link** connects to the next node. Usually, if there are several simple assignments in a row, LCC links them together by the element **link**.

Supports:

```
extern Node newnode(int op, Node left, Node right, Symbol p);
```

The function **newnode** takes major elements of **Node** as arguments and returns a new node structure.

6.1.2 New Data Structures

Assignments Class

- **Condition**

  definition:

```
typedef struct condition{
    Node con;
    con_type type;
    struct condition * left;
    struct condition * right;
    Assign *a;
} Condition;
```

  Description:

  The structure **Condition** plays a very important role in the presentation of the assignment list. As we mentioned in our analysis, when we encounter IF statements, we will split code blocks into parts with overlapping blocks. In order to record the predicates corresponding to all the code split blocks, every IF statement has the **Condition** structure.

Element:

The element **con** identifies the boolean expression in an IF statement.

The element **type** shows whether this condition is connected by an assignment list or just a condition recorder.

The element **left** points to the assignment list corresponding to the split code block when the condition is true.

The element **right** points to the assignment list corresponding to the split code block when the condition is false.

The element **a** is connected to an assignment list corresponding to a split code block.

Supports:

```
Condition* newcondition();
int is_in_c(Condition *tree, Condition *leaf);
void printtable(Condition *tree);
```

The function **newcondition** builds a new condition structure containing default values.

The function **is_in_c**, a seeking function, returns true if the `leaf` is in the tree structure `tree`.

The function **printtable** can print the whole condition structure out in a text string representation.

- **Assign**

  Definition:

```
typedef struct assignment{
  char *name;
  int temp;
  enum {Const, Bool, Undo, LOOP, RES} type;
```

```
  union
  {
    int i;
    Node forest;
    struct func *fn;
    char *str;
  } value;
  struct assignment *next;
} Assign;
```

Description:

The structure **Assign** records the evaluated result for every variable in the relative code blocks. We notice that there is an element **next** which connects all assignments into a list. For every execution branch there is one and only one assignment list corresponding to it.

Elements:

The element **name** identifies the variable name.

The element **temp** shows whether this variable is temporary.

The element **type** tells the type of this evaluated result of this variable. It is used with the union structure **value**.

The element **value** stores the real evaluated result of every variable.

The element **next** points to the next **Assign** structure, which build an assignment list to describe the whole execution branch.

Supports:

```
Assign * newassign();
void printassign(Assign *a,int head);
Assign *assigntail(Assign *a);
int copyassign(Assign *src, Assign **dst);
int findvar(Assign *list, char *name, Assign **ret);
```

The function **newassign** builds a new **Assign** structure containing default values.

The function **printassign** can print out the whole **Assign** structure in text mode. Parameter `head` designates whether to print the assignment variable name out or the right hand expression.

The function **assigntail** puts the new **Assign** `a` at the end of the current assignments list.

The function **copyassign** copies the whole assignments list from `src` to `dst`.

The function **findvar**, a seeking function, searches the assignments list `list` to find the entity with name `name`.

Loops Class

- **Loop**

  Definition:

```
typedef struct loop{
    int done;
    Code begin;
    Code end;
    int sknum;
    Node Tn;//n termination, no guarantee to that
    Node node;
    Variant *variant;
    Assign *asg;
} Loop;
```

  Description:

  Loop is used to describe the loop structure in code blocks. In a program with nested loops, there is more than one loop in the code blocks. For our algorithm,

we need to deal with the inner-most loop first, and then the outer ones. So we need to record all loop structures for later analysis.

Elements:

The element **done** shows whether this loop has been dealt with.

The elements **begin** and **end** record the beginning and end code list entities for the loop. All the entities between **begin** and **end** constitute the whole code during execution of the loop body.

The element **sknum** stores the number of loops in the loop stack. Because we should deal with the inner-most loop first, we must push the outer loops into the stack for later use.

The element **Tn** shows the number of iterations if possible. We know that even for terminating loops, it is not always possible to compute the number of iterations.

The element **node** records information about the guard of every loop.

The element **variant** is a list containing all the variant variables.

The element **asg** records all assignments corresponding to the code blocks in the loop body.

Supports:

Not applicable.

- **Pattern**

  Definition:

```
typedef struct pattern{
    Definition *def;
    Node (*result)(PPara *);
    struct pattern *next;
} Pattern;
```

Description:

The structure **Pattern** is designed for the process of pattern matching so as to get a more common definition for functions with recurrence equations. All patterns are stored in the database, to be linked together.

Elements:

The element **def** present the major structure of **Pattern**. This can determine the explicit function by the information included in the **Definition** structure (see below).

The element **result** points to a function which will be executed when the pattern is matched.

The element **next** links all the patterns into one list.

Supports:

```
void getpatterns();
```

The function **getpatterns** builds the global pattern list from a particular store, currently a piece of description in code.

- **Definition**

Definition:

```
typedef struct definition{
    con_type type;
    Entity *en;
    struct definition *left;
    struct definition *right;
} Definition;
```

Description:

The structure **Definition** records the major features of a function. All recurrence equations will be classified into this pattern style structure before pattern matching.

Elements:

The element **type** denotes whether this function is a combination of two functions under a given condition.

The element **en** describes this condition if **type** is type_C. Otherwise **en** describes the function.

The elements **left** and **right** point to the left and night components of the combined function under the stated condition.

Supports:

```
int eq_pattern(Definition *src, Definition *dst);
```

The function **eq_pattern** compares two **Definition** structures and returns `true` if their contents are same.

- **Entity**

Entity:

```
typedef struct entity{
    pattern_type type;
    struct entity *left;
    struct entity *right;
} Entity;
```

Description:

The structure **Entity** describes the all the features related to a given pattern. It contains the function and the arguments.

Elements:

The element **type** denotes the function.

The element **left** and **right** point to the corresponding arguments.

Supports:

```
new_entity(Entity **ret,pattern_type type,pattern_type left,
pattern_type right);
int eq_entity(Entity *s,Entity *d);
```

The function **new_entity** builds a new **Entity** structure with values of `type`, `left` and `right`.

The function **eq_entity** compares two **Entity** structures `s` and `d` and returns `true` if their contents are same.

- **Variant**

Variant:

```
typedef struct loop_variant{
    char *name;
    struct loop_variant *next;
} Variant;
```

Description:

The structure **Variant** links all the output variables together. It helps the analyzer determine whether a variable is an output variable.

Elements:

Element **name** designates the variable name.

Element **next** points to the next entity.

Supports:

Not applicable.

Variables Class

- **Varname**

  definition:

```
typedef struct varname{
    char *name;
    enum {var_unknown,var_const,var_common,var_array1,var_array2} type;
    char *arg[2];
} Varname;
```

  Description:

  The structure **Varname** records all the common variables and arrays in the same structure. It facilitates variable comparison.

  Elements:

  The element **name** records the major name of this variable.

  The element **type** shows the type of this variable.

  The element **arg** tells the offset if this variable is an array.

  Supports:

```
char *showvar(Varname var);
char *getname(Node nd,Varname *var);
```

  The function **showvar** prints `var` out in text mode.

  The function **getname** builds a new **Varname** structure `var` from structure **Node**.

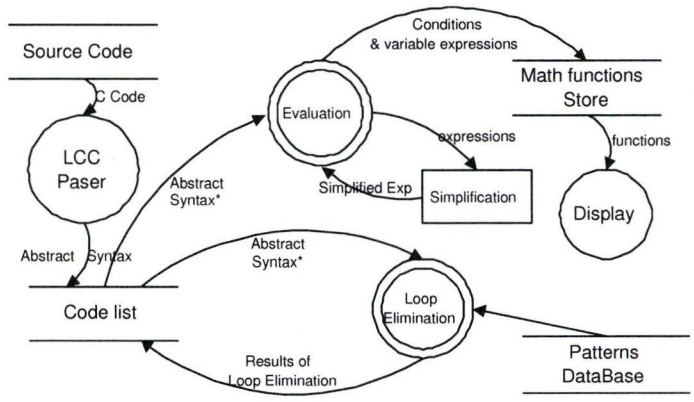

Figure 6.1: System architecture

6.2 System Implementation

In Chapter 4, we discussed the algorithm to transfer the input program to the final tabular expressions. In this section we will discuss all the major components and their control flow. To help the reader understand these, we give the big picture in Figure 6.1.

6.3 Procedure Implementation

From Figure 6.1 we can see that there are two major procedures in our system. In this section we will discuss details of these two procedures. In order to identify their functions and their interaction, we will describe all the data they have to deal with, the output they create, and all side effects. To help the reader understand all this, we give second level data flows in later sections.

6.3.1 Evaluation

Evaluation is performed by the procedures **Walker**, **Path Splitting** and **Evaluation** procedures. The relationship between these procedures as shown in Figure 6.2

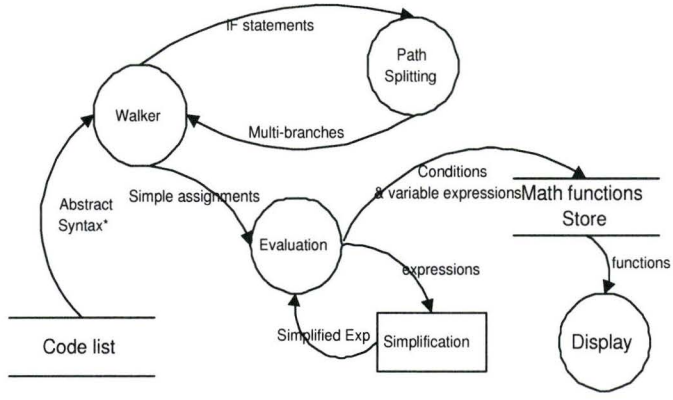

Figure 6.2: Big picture of Evaluation

The procedure Walker

This is the major function dealing with all the entities from the code list. It decides how our algorithm should cope with each code entity, representing the abstract syntax of our input programs. **Walker** should deliver statements such as simple assignments or IF statements into corresponding procedures to be analyzed. However loops are a special type of control structure that require more analysis, which will be done in the procedure **Loop Analyzer**.

Examining Figure 6.3, we notice that the main part of **Walker** is a loop, which deals with every entity from the code list which we are going to analyze.

- Input:

  Walker takes the code list as its input. This is a kind of abstract syntax graph structure.

- Output and Side Effects:

  Actually **Walker** does not do much work with the data structures related to our algorithm. It will hand all the work to procedures like **Evaluation** or **Path Splitting** after which the assignment lists are created.

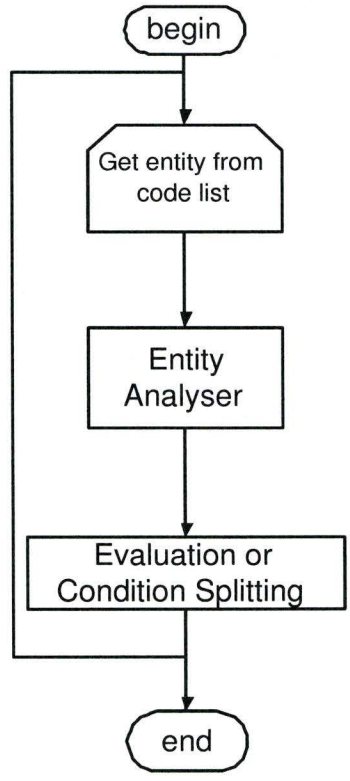

Figure 6.3: Procedure code Walker

The procedure Evaluation

This plays a significant role in our algorithm. It is involved in most of the major procedures. For the functionality of **Evaluation** a global variable **a_current** is used to record the current states for the input program.

In Figure 6.4 we give the control flow of procedure **evaluation**.

- Input:

  Evaluation takes an expression as input. Based on the current assignment list which actually plays the role of a state we can provide the value after applying the substitution.

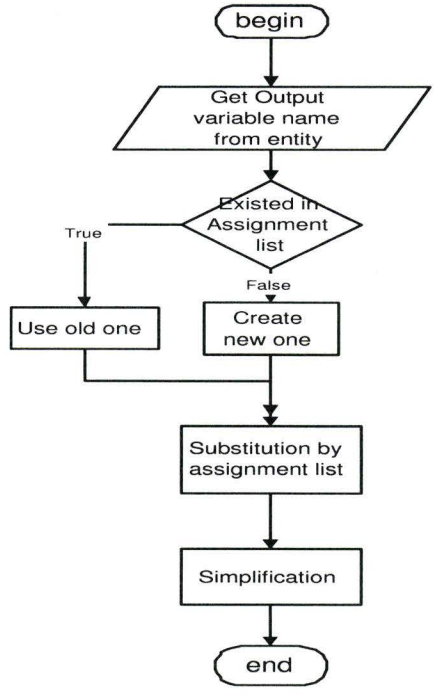

Figure 6.4: Procedure Evaluation

- Output and Side Effects:

  With **Evaluation**, we can get the new value of the variable, which is the left value of this assignment. Also we need to update the entity related to this output variable in the current assignment list. Things are more complicated if **Evaluation** is called by statements in the loop body. We need to form the result value in the expression of the recurrence equation. Fortunately this is not difficult for the data structure **Assign**.

The procedure Path Splitting

From the earlier part of this chapter we know that the data structure **Condition** has been used to record all conditions in IF statements. **Path Splitting** is the procedure which generates all the assignment lists according to the execution branches generated

by an IF statement.

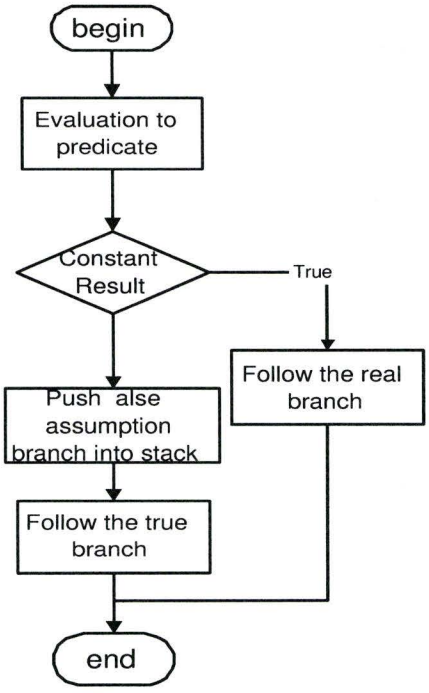

Figure 6.5: Procedure Path Splitting

- Input:

  One of the inputs is the code entity of the IF statement we want to analyze. At the same time, it is similar to the global variable **a_current** in the procedure **Evaluation**, since there is another global variable **c_current** that has been used to record information about the current execution branch.

- Output and Side Effects:

  When **Walker** encounters an IF statement in the code list, it distributes this entity to the procedure **Path Splitting**, which first evaluates the condition in the IF statement. If the result evaluates to either `true` or `false`, we walk

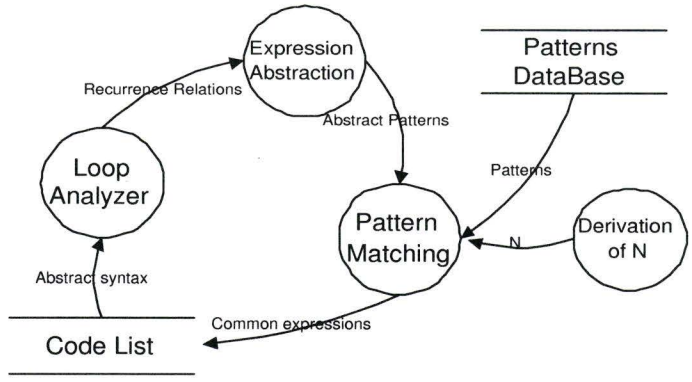

Figure 6.6: Loop Elimination

through the corresponding execution branch. However, in most cases we can not evaluate the condition. Then we make both assumptions about this condition, to let **Walker** proceed through both of these execution branches. So we push one execution branch onto the stack for later use, and walk the other one through.

6.3.2 Loop Elimination

We have seen that loop elimination includes several procedures which solve major problems such as recurrence equations, abstraction and pattern matching. We have already given these analysis in Chapter 4. In order to clarify the detailed procedure of loop elimination, we give a second level flowchart in Figure 6.6

Loop Analyzer

Loop Analyzer is responsible for dealing with loop relevant problems. In Figure 6.7 the main algorithm of this procedure is shown.

From this flowchart we find that there are three other procedures involved in this procedure. They co-operate to deal with loop structures in the input code. For the code blocks in every loop body, we found that there is a strong resemblance between assignments and IF statements, and the corresponding ones in the loop body. So we

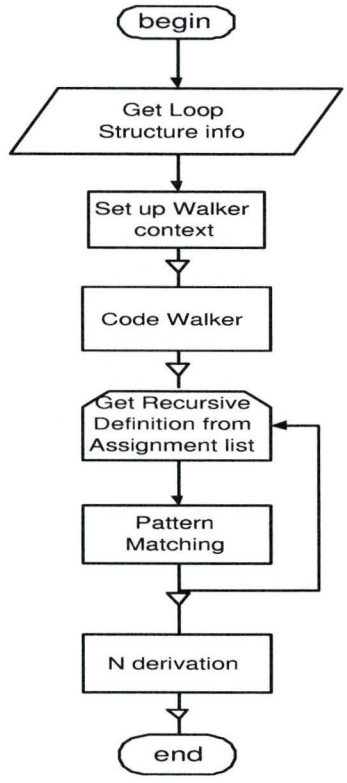

Figure 6.7: Procedure Loop Operator

use procedure **Walker** in both these analyse, which should make our method more efficient.

- Input:

  Loop Analyzer requires the loop data structure, which includes all the information on the loop. This structure should be built as part of the initialization step.

- Output and Side Effects:

  Throughout all loop related procedures, equivalent simple assignments are created to substitute for the loop statement. By executing this procedure repeat-

edly, we can also cope with nested loops.

Pattern Matching

After walking through the statements in inside one loop body, there is an assignment list which records all the recurrence equations of the output variables. **Pattern Matching** attempts to drive these recurrence equation into common expressions when possible.

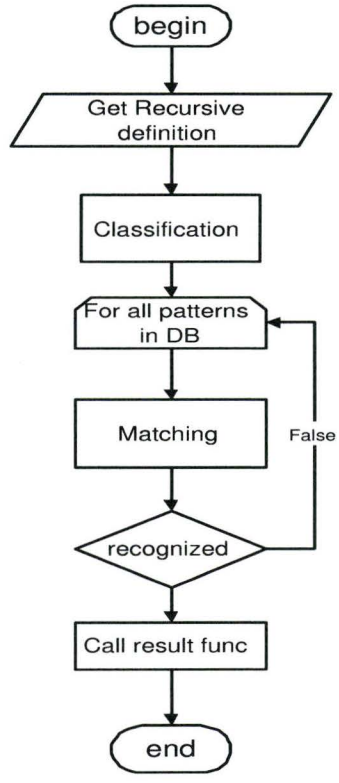

Figure 6.8: Procedure Pattern Matching

- Input:

For every output variable from a specific loop, we already have got its recurrence equation. We take these variables as this procedure's input. There is also a database for loop patterns. We use all the patterns in this database to try to recognize those recursive expressions.

- Output and Side Effects:

  If the recurrence equation of some output variable has been matched with any pattern in patterns database an explicit expression will be returned to the related variable. This process is repeated in the case of nested loops.

N derivation

The procedure **N Derivation** is designed to decide whether a loop terminates, and in how many iterations, based on some preconditions. However, it is extremely complex to derive this value in a systematic way. Here we just supply an intuitive solution for a very simple example. There is room for improvement here in future work.

Tabular Display

After we walk through the code list, we present all functions in the data structure **Condition** which states all the assignment lists. Tabular expressions are known for theirs readability, so we display these functions by means of tabular representations. As discussed in earlier chapters, we use vector function tables in our tool here.

6.4 A Real Example

To make the algorithm more understandable, we present a real example with debugging. Consider Example 4.23 in Chapter 4. This includes alternations and iterations, which can really take us through all the steps of our implementation.

From our analysis in Chapter 4, we know we must build the code list from source code at the beginning. By using all the data structures listed in the first section of this chapter, we construct the list shown in Figure 6.9.

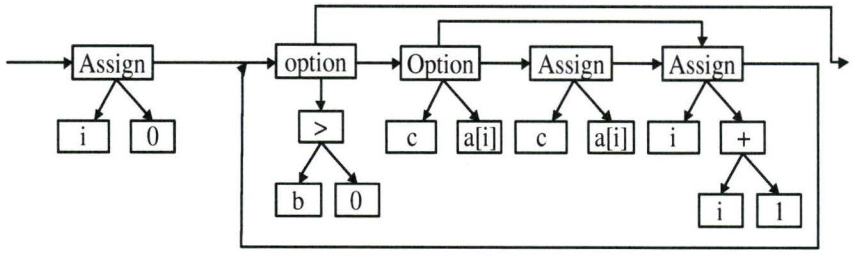

Figure 6.9: Code list of a real example

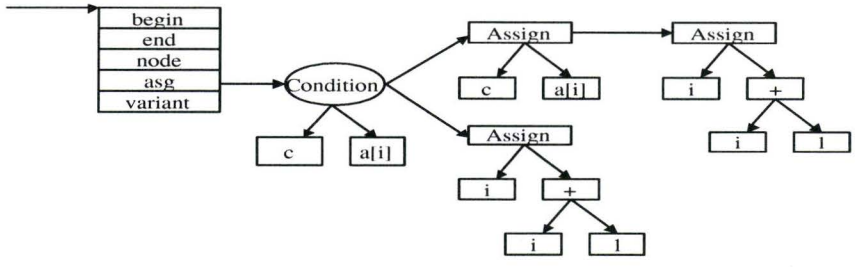

Figure 6.10: Loop structure of the example

In this example we perform loop elimination according to our algorithm. For every loop in the code we build a loop structure to contain all the information we need. Then we go through all the assignments in the loop body to create the assignments list, before representing these variables by recurrence equations. In Figure 6.10 we can see the structure of our implementation.

Then, from the assignment list in the loop structure, we produce the recurrence equation of every output variable. As discussed in Chapter 4, we can transform these recurrence equations to normal form. After this loop elimination procedure, we rebuild the code list as shown in Figure 6.11.

In the end, all loops have been eliminated, from the inner-most ones to the outer-most ones. Then we can begin to evaluate all the assignments from the code list. In Figure 6.12 we show the inside structure representing the final values of the output variables. We can see that it is easy to build tables from the information in this link.

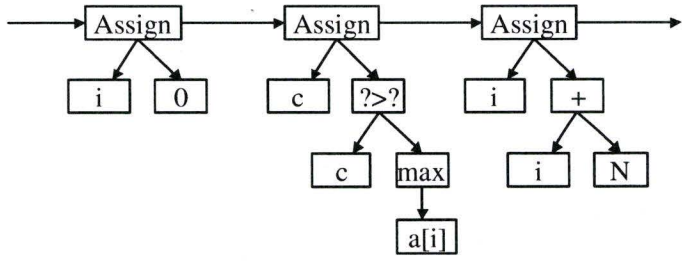

Figure 6.11: Code list after Loop Elimination

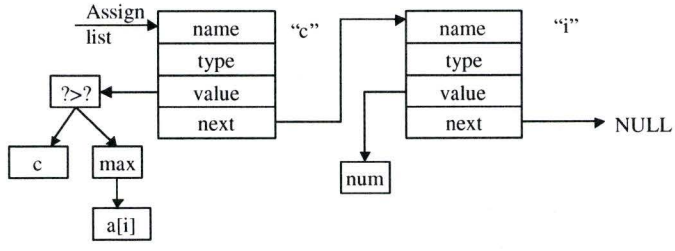

Figure 6.12: Final result of Evaluation

Chapter 7

Results

In this chapter we show some results of applying our methods and tool to typical examples. From these examples we see that our tool works well in these cases. Using pattern matching techniques several explicit functions can be extracted from the loop structures. Although currently we have only a few patterns in our database, it seems promising that we will be able to construct patterns that are successful within specific domains. Because of time constraints we display our results in text mode. However, we can see that those results can be represented by tabular expression without much difficulty. Also, these testing results can also provide direction for future work.

7.1 Straight line code

Simple assignments are the basic statements in imperative language. The solution for straight line code describes the functionality of the code for every output variables.

Figure 7.1 shows the results from our tool applied to the example in Figure 4.7. We see that our tool first prints out the input, output and update variables list. Then, the final value of every variable is displayed. For straight line code there is no other execution branch in the code, so we use "true" as the only applicable condition.

From the results shown in Figure 7.1, we can easily build the tabular expression of this example as shown in Figure 7.2. As we said that our tool represented in this thesis is just a prototype tool. The output as shown in Figure 7.2 will be the future

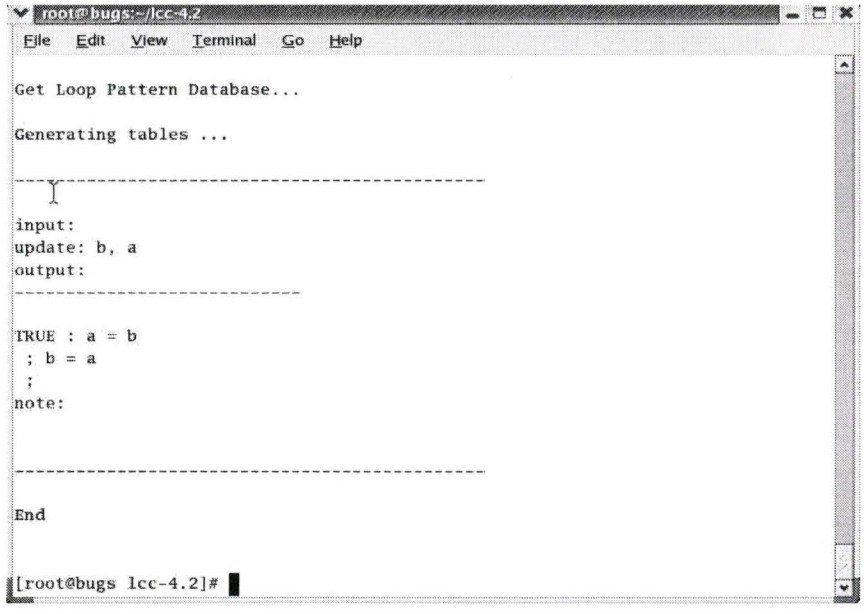

Figure 7.1: The result of the straight line code

work of this research.

7.2 Alternation

Conditions or alternative statements are the statements with alternative choice that use IF or SWITCH/CASE keywords, which are also primitive control statements in imperative languages. The readability of tabular expressions helps inspectors and

$TRUE$	
a'=	'b
b'=	'a

Figure 7.2: Table representing Figure 4.7

maintainers understand the descriptions of the behaviors of target programs with conditions. Consider an example from PID controller software as shown in Figure 7.3.

```
int derivative_term;

derivative_term = derivative_term *kd;
derivative_term = derivative_term >>5;
if (derivative_term > 120){
    derivative_term = 120;
}
```

Figure 7.3: An example of PID controller

We see that the result can be outputted as shown in Figure 7.4.

```
Generating tables ...

-----------------------------------------------

input: kd,
update: derivative_term
output:
----------------------------

(((derivative_term*kd)>>5) <= 120): derivative_term = (derivative _term*kd)>>5
 ;
(((derivative_term*kd)>>5) > 120): derivative_term = 120 ;
note:


-----------------------------------------------

End
```

Figure 7.4: The result of PID example

7.3 Iterations

So far we have seen that our tool can analyze programs consisting of straight line code and alternations. Although the result for any output variable could be a huge expression, it still accurately identifies the correct final value of that output variable. However, if there are loops in the code, the pattern matching technique probably can not recognize all the functions. Recurrence equations will be used as well for general examples. Next we present some result of iteration examples.

7.3.1 Single-level Iterations

First consider single-level loops as in Figure 4.12.

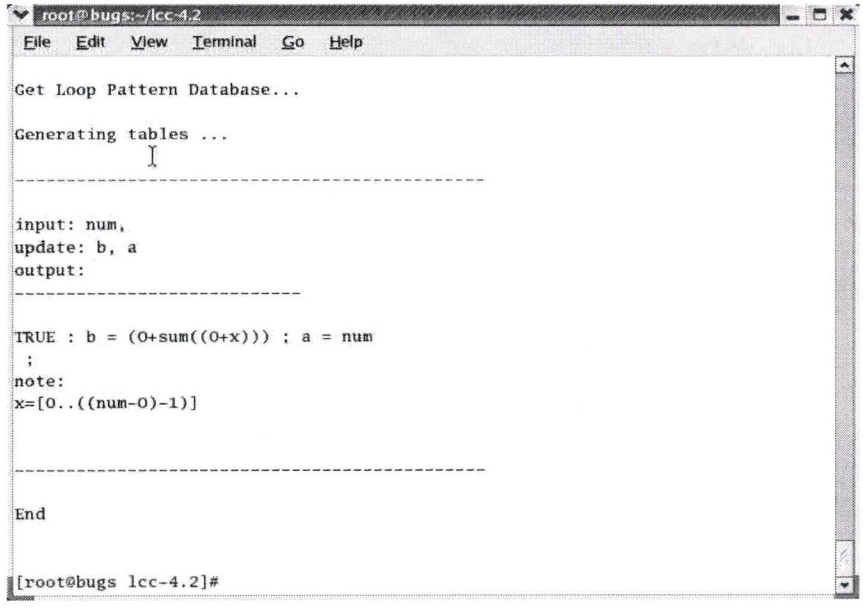

Figure 7.5: The result of the single-level loop

In this example the function returns the sum of consecutive integers from **0** to **num**. For our result in Figure 7.5, we notice that we use a temporary index variable

x in final value of variable **b**. We declare those temporary index variables in a section called **note**. In this example we see that **x** is in the range from **0** to **num**.

7.3.2 Nested Iterations

For programs with nested loops, we will show that the pattern matching results are used repeatedly. Therefore, a well-defined pattern can be used many times in code with loops and will recognize the pattern as many times as it occurs.

Consider an example with nested loops shown in Figure 7.6

```
int i,j,d,b,c[1024][1024];
int m,n;
d=0;
for (i=1;i<n;i++)
{
    for (j=1;j<m;j++)
    {
        if (b<c[i][j])
            b = c[i][j];
    }
    d += b;
    b=0;
}
```

Figure 7.6: An example with nested loops

In this example we see that we first get the maximum values of every row in an array **c**. Then we put the sum of those maximum values into variable **d**. In Figure 7.7 we show the result outputted by our tool.

We see that there are two temporary index variables in **note** section for every level of loops. The key word **max** in the result of output variable **d** means we get pattern matching successfully for the inner loop to get maximum. Then we can do pattern matching again for outer loop to get the final result for variable **d**. For detailed process of this loop elimination we describe it in Figure 4.22.

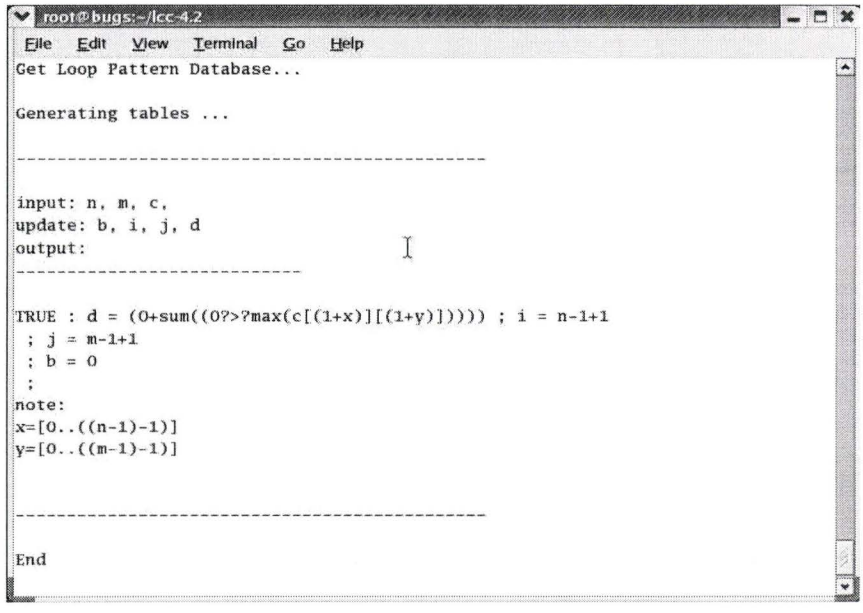

Figure 7.7: The result of the nested loop

Chapter 8

Conclusions and Future Work

In this thesis we have presented methods and a tool implementation for automatic recognition of expressions implemented in code, and this can lead to automatic table generation. We have also displayed results of particular examples, for which our application works well on a set of typical cases, and for which the techniques presented are applicable. This research work makes major contributions in the following ways:

- Proves the possibility of total automatic methods to generate function tables from high level imperative languages.
- Inspects all steps of automatic table generation process. We encountered and analyzed major difficulties in the automatic generation methods, and recorded our experience about specification recovery from code.
- Uses a Pattern Matching technique to abstract high level specifications from loops. This incomplete matching methods give a new way to extract the explicit functions out of target code with loops.
- Builds a practical tool to help software inspection. Every successful methodology always has comprehensive strong tool support [53]. This research work has started an attempt to develop the tools support this method.

There are still a number of problems we have to face to make our method and tool more complete.

- More data types should be considered in the future. Only **int** type has been dealt with in our tool. New involvement of those more complex data types will require a comprehensive data flow analysis and data abstraction process.
- Automatic method for iteration termination determination and iteration count derivation will be a focus of future research.
- Although we found that our tool is applicable in many cases, we still need to prove the soundness of pattern matching method in a more rigorous way.
- Procedures are also another problem we did not discuss in this thesis because of existent of time being. More research will be targeted on this issue in future.

Bibliography

[1] Archinoff, G.H., Hohendorf, R.J., Wassyng, A., Quigley, B., Borsch, M.R., Verification of the Shutdown System Software at the Darlington Nuclear Generating Station, Proceedings of the International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, May 1990.

[2] Breuer, P. and Lano, K., Creating Specifications from Code: Reverse-engineering Techniques, Journal Software Maintenance: Research and Pratice, vol. 3, 1991, pp. 145-162.

[3] Britton, K.H., Parker, R.A., and Parnas, D.L., A Procedure for Designing Abstract Interfaces for Device Interface Modules, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, ISBN 0-201-70369-6. pp. 295–313, 2001.

[4] E. Byrne. A Conceptual Foundation for Software Reengineering. In Proceedings for the Conference on Software Maintenance, pages 226C235. IEEE, 1992.

[5] Chikofsky, E. and Cross, J., Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1): 13–17, January 1990.

[6] Cousot, P., Cousot, R., Abstract Interpretation: a Unified Lattice Model for Static Analysis by Construction or Approximation of Fixpoints, Proceedings of POPL'77, ACM Press, Los Angeles, California, pp. 238-252.

[7] Dijkstra, E., Scholten, C., A Discipline of Programming, Page 18-19, 1976.

[8] Dijkstra, E., Scholten, C., Predicate Calculus and Program Semantics, Springer-Verlag, 1989.

[9] Ernst, M., Cockrell, W., Notkin, D., Quickly Detecting Relevant Program Invariants, Proceedings of the 22nd International Conference on Software Engineering(ICSE2000), Limerick, Ireland, 2000.

[10] Gannod, G., Cheng, B., Strongest Postcondition as the Formal Basis for Reverse Engineering. To appear in Journal of Automated Software Engineering. An earlier version of this paper appeared in the Proceedings for the SecondWorking Conference on Reverse Engineering, 1996.

[11] Gannod, G., Cheng, B., Using Informal and Formal Techniques for the Reverse Engineering of C programs, international conference on Software Maintaince, Nov. 1996, pp. 265-274.

[12] Gannod, G., Cheng, B., A suite of tools for facilitating reverse engineering using formal methods, Technical Report ASUCSE-TR99-02, Arizona State University, May, 1999

[13] Gannod, G., Cheng, B., A Formal Approach for Reverse Engineering: A Case Study, In Proceedings of the 6th Working Conference on Reverse Engineering, IEEE, October, 1999

[14] Flanagan, C., Qadeer, S., Predicate Abstraction for Software Verification, Proceedings of the 29th Annual ACM SIGLPAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, 2002.

[15] Heninger, K., Specifying Software Requirments for Complex Systems: New Techniques and Their Application, Collected Papers by David L.Parnas, Addison-Wesley, ISBN 0-201-70369-6. pp. 111–133. 2001.

[16] Heninger, K., Kallander,J., Parnas,D.L., and J.Shore, Software Requirements for the A-7E Aircraft, Naval Res. Lab., Memo Rep. 3876, Washington, DC, Nov. 27, 1978.

[17] Hirvisalo, V. Combining Static Analysis and Simulation to Speed up Cache Performance Evaluation of Programs. Nordic Workshop on Software Development Tools and Techniques, Copenhagen, IT University of Copenhagen, pp. 117–128, August 2002.

[18] Ireland, A., Stark, K., On the Automatic Discovery of Loop Invariants, 4th Nasa Langley Formal Methods Workshop, 1997.

[19] Janicki, R., Parnas, D.L., Zucker, J., Tabular Representations in Relational Documents, in Relational Methods in Computer Science, Brink, C., Kahl, W., Springer Verlag Vienna, pp. 184–196, 1997.

[20] Janicki, R., Khedri, R., On Formal Semantics of Tabular Expressions, Science of Computer Programming, 39(2001), 189–214, 2001.

[21] Janicki, R., Wassyng, A., On Tabular Expressions, In D.A. Stewart, ed. Proceedings of CASCON 2003, Markham, Ontario, Canada, 38–52, October 2003.

[22] Janicki, R., Wassyng, A., Tabular Expressions and Their Relational Semantics, Fundamenta Informaticae, Vol. 68, 1-28, 2005.

[23] Joannou, P., Wassyng, A., Modelling for Requirements Analysis and Design, in Software Important to Safety in Nuclear Power Plants, International Atomic Energy Agency, Vienna, Technical Reports Series No. 367, 1994.

[24] Kahl, W., Compositional Syntax and Semantics of Tables, SQRL Report No. 15, McMaster University, Hamilton, Ontario, Canada, 2003, to appear in Formal Methods in System Design.

[25] Fraser, Christopher W., Hanson, David R., A Retargetable C Compiler: Design and Implementation, Addison-Wesley, 1995

[26] Lee, L. and Hwang, S.H., Abstract Simulator: A DSP Software Timing Analysis Tool; web pdffile, URL: *http* : *//www.icspat.com/papers/97mfi.pdf* ;

[27] Li, Y.S., Malik, S., Performance Analysis of Embedded Software Using Implicit Path Enumeration in Proceeding of the 32nd Design Automation Conference, page 456–461, 1995.

[28] Mills, H.D., The new math of computer programming, Commun. ACM, 18, 1, pp. 43-48, Jan. 1975.

[29] Parnas, D.L., Precise Description and Specification of Software, Software Fundamentals, in Mathematics of Dependable System II, edited by V. Stavridou, Clarendon Press, pp. 1–14, 1997.

[30] Parnas, D.L., On the Criteria Be Used in Decoposing Systems, Communications of the ACM, 15, 12, pp. 1053-1058, Dec 1972.

[31] Parnas, D.L., Some Software Engineering Principles, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 664 pgs., ISBN 0-201-70369-6, 2001.

[32] Parnas, D.L., Predicate Logic for Software Engineering, IEEE Transaction on Software Engineering, Vol.19, No. 9, 1993, pp. 856–862, Sep. 1993.

[33] Parnas, D.L., Madey, J., Iglewski, M., Precise Documentation of Well-Structured Programs, IEEE Transactions on Software Engineering, pp 948–976, Vol. 20, No.12, Dec.1994.

[34] Parnas, D.L., Madey, J., Functional Documentation for Computer Systems Engineering, in Science and Computer Programming, (Elsevier) 25[1], pp. 41–61, Oct. 1995.

[35] Parnas, D.L., Tabular Representation of Relations, CRL Report 247, McMaster University, Communications Research Laboratory, TRIO(Telecommunications Research Institute of Ontario), 17 pages, Oct. 1992.

[36] Parnas, D.L., Lawford, M., The Role of Inspection in Software Quality Assurance, IEEE Transaction on Software Engineering, Vol.29, No.8, pp 674–676 Aug. 2003.

[37] Parnas, D.L., Less Restrictive Constructs for Structured Programs, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 31–48.

[38] Parnas, D.L., Some Software Engineering Principles, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 255–266.

[39] Parnas, D.L., Design Software for Ease of Extension and Contraction, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 269–290.

[40] Parnas, D.L., Inspection of Safety-Critical Software Using Program-Function Tables, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 371–382.

[41] Shen, H., Implementation of Table Inversion Algorithms, CRL Report 315, McMaster University, Communications Research Laboratory, TRIO, Dec. 1995.

[42] Shen, H., Zucker, J.I., Parnas, D.L., Tabular Transformation Tools: Why and How, proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS' 96), publised by IEEE and NIST, Gaithersburg, MD., pp. 3–11, June 1996.

[43] Susuki, N., Ishihata, K., Implementation of an Array Bound Checker, 4th ACM Symposium on Principles of Programming Languages, Los Angeles, CA, 1977

[44] Xu, J., On Inspection and Verification of Software with Timing Requirements, IEEE Transaction on Software Engineering, Vol.29, pp. 705–720, No.8, Aug. 2003.

[45] Ward, M.P., Proving Program Refinements and Transformations, Oxford University, DPhil Thesis, 1989

[46] Wang, Yali, Display Management System, (A tool to support the Display Method), CRL Report 297, McMaster University, Communications Research Laboratory, TRIO, Apr. 1995.

[47] Ward, M.P., The FermatT Assembler Re-engineering Workbench, International Conference on Software Maintenance 2001, 6th-9th, Florence, Italy IEEE Computer Society, pp. 659-662, November 2001.

[48] Ward, M.P., Reverse Engineering from Assembler to Formal Specification via Program Trans-formations, 7th Working Conference on Reverse Engineering, 23rd-25th Nov., Brisbane, Queensland, Australia, 2000.

[49] Ward, M.P., Bennett, K.H., A Pratical Program Transformation System For Reverse Engineering, Working Conference on Reverse Engineering, May 21-23, 1993

[50] Ward, M.P., Specification from Source Code - Alchemists' Dream or Practical Reality? 4th Reengineering Forum, Sep.19-21, 1994, Victoria, Canada, 1994.

[51] Wassyng, A., Lawford, M., Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project, Proc. of FME03 (Formal Methods Europe), Lecture Notes in Computer Science 2805, Springer 2003, pp. 133–153.

[52] Wassyng, A., Janicki, R., Using Tabular Expressions, In Proceedings of International Conference on Software and Systems Engineering and their Applications, Paris, Vol. 4, 1–17, December 2003.

[53] Wassyng, A., Lawford, M., Software Tools for Safety-Critical Software Development, International Journal of Software Tools for Technology Transfer, 2006.

Appendix:

A CD containing the source code for the proof-of-concept tool is included