1990

Automatic transformation of high-level logic specifications into high-performance target code.

Aiqin Pan
University of Alabama at Birmingham

Order Number 9114890

Automatic transformation of high-level logic specifications into high-performance target code

Pan, Aiqin, Ph.D.

University of Alabama at Birmingham, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

AUTOMATIC TRANSFORMATION OF HIGH-LEVEL LOGIC SPECIFICATIONS INTO HIGH-PERFORMANCE TARGET CODE

by

AIQIN PAN

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer and
Information Sciences in the Graduate School,
The University of Alabama at Birmingham

BIRMINGHAM, ALABAMA

1990

ABSTRACT OF DISSERTATION
GRADUATE SCHOOL, UNIVERSITY OF ALABAMA AT BIRMINGHAM

Degree Doctor of Philosophy Major Subject Computer and Information Sciences

Name of Candidate Aiqin Pan

Title AUTOMATIC TRANSFORMATION OF HIGH-LEVEL LOGIC SPECIFICATIONS INTO HIGH-PERFORMANCE TARGET CODE

Development of software systems typically proceeds from informal user requirements through development of formal specifications which form the basis for detailed designs and implementations. Of these, implementation is the most involved, costly, and source of errors which require continuous maintenance throughout the software life-time. An additional source of errors is miscommunication between users who develop the informal requirements and designers and implementors who effect those requirements.

A methodology is proposed for formal specification and automatic generation of software systems, based upon the theory of Two-Level Grammar (TLG). TLG is developed into a complete specification language under the paradigms of functional and logic programming. TLG specifications are unique in that they are a structured form of natural language which is executable, greatly increasing the reliability of the developed software system for the following reasons. Because it is a form of natural language, TLG may be used as an effective communication medium between users, designers, and implementors of the system, thereby reducing errors caused by miscommunication. Furthermore, because TLG is implementable, we may automatically generate the software system from the specification in a provably correct way. This is accomplished by transforming the TLG high-level specifications into efficient target programs expressed in C. Our target programs are then compiled using widely-available C compilers, making them highly portable, and they may interface with many existing C programs, facilitating software reuse. Finally, the main advantage is that developing TLG specifications is considerably easier and less error-prone than developing C programs, where considerably more implementation detail is needed.

The effectiveness of our method is demonstrated through the automatic generation of: 1) a lazy functional language graph reduction implementation, 2) a semantics-directed compiler for a Pascal-like language, and 3) a database/knowledge-base management system based upon SQL. For the DBMS, we have the additional advantage that the TLG specification results in an embedding of SQL into TLG

offering both deductive capabilities and expressive natural language queries. All the generated programs are comparable in time and space efficiency with corresponding C programs coded by hand and greatly improve upon interpretative and Prolog prototype versions of the TLG specifications.

Abstract Approved by: Committee Chairman Barrett R. Bryant

Program Director Warren D. Jones

Date 4/16/91 Dean of Graduate School Anthony Bruno

DEDICATION

This dissertation is dedicated to my parents.

ACKNOWLEDGEMENTS

First of all, I would like to take this opportunity to express my appreciation to all the people at the University of Alabama at Birmingham for leaving me a warm, delightful, and unforgettable memory in my life. During the three and a half years study at UAB, I gained a lot, both in professional development and personality growth. I thank everyone with all my heart, for their advice and encouragement as an instructor and faculty member, their cooperation as a colleague, and for help and understanding as a friend.

I especially thank my respected advisor Dr. Barrett R. Bryant for his continuous guidance and correction of my academic course of study, and for his research grant to help me financially and, more importantly, to offer me a chance to take part in his research directly. I am moved by his hard work, his high enthusiasm for his research, and his rigorous but kind style of training his students. He has always been a perfect example for me, not only in academic research, but also how to work and live as a human being.

My appreciation also goes to my committee members, Dr. Warren T. Jones, Dr. Edwin L. Battistella, Dr. Robert M. Hyatt, and Dr. Lee K. Mohler. They gave me many helpful suggestions and spent a lot of time discussing my research. I would like to thank Dr. Kenneth R. Sloan for his treasurable suggestion on improving my seminar presentation on this research, which eventually led me to my first professional position of employment with IBM Corporation in San Jose, California. During my Ph. D. study, the UAB Graduate School has supported me with a Graduate Research Assistantship, for which I am also extremely grateful.

Finally, my heartiest and deepest appreciation goes to my dear parents. They influenced me in my childhood with a strong academic atmosphere and have guided me and encouraged me throughout my life. I am also indebted to my younger sister, Aituan, for always giving me her most sincere cheer.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGEMENTS v

LIST OF FIGURES viii

CHAPTER

I. INTRODUCTION 1
- 1.1. Formal Specification and Transformation 2
- 1.2. Two-Level Grammar as a Specification Language 3
- 1.3. Implementation of Two-Level Grammar 5
- 1.4. Two-Level Grammar Specification of Database and Knowledge-Base Systems ... 7
- 1.5. Two-Level Grammar Specification of Language Implementation Systems 8
- 1.6. Summary 9

II. FORMAL SPECIFICATION AND TRANSFORMATION METHODOLOGIES 10
- 2.1. Properties of Formal Specifications 11
- 2.2. Specification Languages 12
- 2.3. Verification of Logic Specifications 14
- 2.4. Program Transformation and Synthesis 16
- 2.5. Overview of Transformation Strategies 17

III. TWO-LEVEL GRAMMAR SPECIFICATION LANGUAGE 20
- 3.1. Introduction to Two-Level Grammar Language 20
- 3.2. Type Structures 22
- 3.3. Built-In Functions 24
- 3.4. Determinism versus Nondeterminism 24
- 3.5. Mode Analysis 26
- 3.6. Sequential Interpretation 26
- 3.7. Data Flow Parallel Interpretation 27
- 3.8. Implementation of Rule Matching 31
- 3.8.1. Finite Automata Techniques 31
- 3.8.2. Context-Free Parsing Techniques 32
- 3.9. Two-Level Grammar as Specification Language 33

IV. IMPLEMENTATION TECHNIQUES FOR TWO-LEVEL GRAMMAR SPECIFICATIONS 36
- 4.1. Interpretation 37
- 4.2. Preprocessing 40
- 4.3. Transformation 42

TABLE OF CONTENTS (continued)

4.4. Automatic Type Inference 45
4.4.1. Two-Level Grammar Type Inference 45
4.4.2. Logical Type Inference 46
4.5. Generation of C Code 53
4.6. Evaluation 56

V. COMPLETE COMPILER SPECIFICATION USING TWO-LEVEL GRAMMAR 59
5.1. Denotational Semantics 60
5.2. Denotational Semantics Using Two-Level Grammar 62
5.2.1. Interpreter-Oriented Denotational Semantics 63
5.2.2. Compiler-Oriented Denotational Definition 63
5.3. A Two-Level Grammar Lambda Machine 65
5.4. Two-Level Grammar Implementation of Functional Programs 67
5.4.1. Strict and Lazy Evaluation 68
5.4.2. Fully Lazy Evaluation Using Graph Reduction 69
5.4.2.1. Association List 69
5.4.2.2. Unification 70
5.4.2.3. Realizing Higher-Order Functions in Two-Level Grammar 70
5.4.3. Compilation into Supercombinators 71
5.5. Polymorphic Type Inference 73
5.6. Summary 74

REFERENCES 75

LIST OF FIGURES

Figure	Page
2.1. Software Life-Cycle	10
2.2. Automatic Program Generation	11
2.3. Compilation versus Transformation	17
3.1. Two-Level Grammar for Quick Sort	25
3.2. Deterministic Two-Level Grammar for Quick Sort Split Function	26
3.3. Two-Level Grammar Derivation Tree for f(1,0)	30
3.4. Two-Level Grammar Derivation Tree for g(1,0)	31
3.5. Two-Level Grammar Derivation Tree Cuts	33
3.5. Comparison of Two-Level Grammar with Functional, Logic, and Natural Languages	33
3.7. ML and Prolog Specifications of Palindromes	34
3.8. Miranda and Prolog Specifications of Quick Sort	34
4.1. Implementation Techniques for Two-Level Grammar	36
4.2. Ordered Keyword Parsing Algorithm	39
4.3. Two-Level Grammar Interpretation	40
4.4. Implementation of Two-Level Grammar by Preprocessing	40
4.5. Implementation of Two-Level Grammar by Transformation	43

CHAPTER I

INTRODUCTION

The development of software systems typically proceeds from a set of informal requirements of the users of the system through development of a more formal specification which can be used as the basis of detailed design and implementation. Of these steps, implementation is usually the most involved, costly, and greatest source of errors which must be continuously corrected throughout the life of the software. An additional source of errors is a lack of communication between the users who develop the informal requirements and the designers and implementors who effect those requirements. The objective of this research is to eliminate these sources of errors by the development of a formal specification language through which novice users can communicate with professional designers and implementors, and which itself may be efficiently, correctly, and automatically implemented, thereby eliminating the need for much of the implementation process.

The embodiment of what we are attempting has been realized in only a small number of application areas. Most notable among these is the automatic generation of programming language syntax analyzers from Backus-Naur specifications (BNF) [Back60]. For example, YACC (Yet Another Compiler Compiler) [John75] has been widely used in producing compiler front-ends for fifteen years. The reason for its success is that the input specification language, BNF, is the standard method of expressing language syntax in almost every programming language reference manual and has been for three decades. BNF is easily understandable to programmers using the language, but at the same time provides the mathematical foundation and formality upon which an automatic parser generation system can be constructed. Furthermore, in the case of YACC, the parsers which are generated are not stand-alone programs with narrow utility. They are coded in the C programming language [Kern88], also an industry standard for software development, and hence can interface easily with many existing C programs, such as compiler back-ends.

Unfortunately, most other application areas lag far behind automatic parser generation. Even the remaining portions of compilers are still written by hand. The primary reason for this is that the existing specification methods have not been as versatile as BNF has been in its domain. The specifications have been very difficult to implement or have been unclear to users or both. In the compiler field, denotational semantics (e.g., see [Schm86]) remains the standard method of specifying programming

languages and is the principal subject for automatic compiler generation research. However, denotational semantics specifications are quite mathematical and so not well suited for user clarity. On the other hand, implementations, while provably correct, are also considerably slower than those produced the "old-fashioned way," by hand. This is also due in part to the mathematical models which are used as the semantic foundations. The result is that the semantic descriptions given in programming language reference manuals are still written in natural language and the major portions of compilers are produced manually.

We propose to accomplish our stated goal by using a formal specification language based upon Two-Level Grammar (TLG, also called W-grammar) [Wijn65]. TLG was first proposed as formal language and later used as a specification method for the syntax and semantics of ALGOL 68 [Wijn74]. The method as used in the ALGOL 68 definition was neither understandable to users of the language nor did it lend itself well to implementation. These two factors are in large part responsible for the ultimate demise of ALGOL 68. Edupuganty [Edup87] developed a new method of expressing Two-Level Grammar which overcame many of the difficulties of the ALGOL 68 definition. He developed readable and implementable specifications of programming languages using operational, axiomatic and denotational semantics approaches. This thesis extends his work in two major ways:

1) We develop TLG as a general-purpose specification language, not just for language specification. Additional constructions and improvements are added to the language which are needed for this more general use. The readability of TLG is also greatly improved by generalizing the usage of natural language in the specifications.

2) Edupuganty's work in the area of implementation was restricted to the development of prototypes. We give a method for the *efficient* implementation of TLG specifications so that these can be used in production-quality systems. Our primary method is to transform the specifications into target programs written in C. Because the distance between TLG and C is very wide, a number of new transformation and compilation techniques had to be developed to realize the desired efficiency.

In the sections which follow, the rationale and basis for our work will be further explored.

1.1. Formal Specification and Transformation

Ideally the users will state their requirements to the software system designer using some form of informal document. This document is then formalized into a specification. We desire a specification language which is robust enough to accommodate the original informal specification and then move to a more design-oriented, even implementation-oriented description. Such a specification language is said to be *wide spectrum* because it can cover a wide variety of potential usages. We also want our formal

specification to be expressed in a way that the user can still recognize his original intent. This goal is quite ambitious. It can only be accomplished if the specification is heavily natural language oriented. However, this is in direct confrontation with the goal of being formal. Natural language specifications tend to be quite informal and often ambiguous, incomplete, and imprecise. Therefore, we need to establish a framework for natural language specifications which avoids these pitfalls. To solve this, we will embed the natural language into the mathematically rigorous functional and logic programming paradigms, which have well-established advantages for formal specification languages (e.g., see [Kowa85] and [Turn85]).

In functional and logic programming, the line between programs and specifications is very blurred, potentially reducing the problem of writing a specification to that of writing a program. Both methods are declarative. Functional and logic programming languages have simple data types and uniform syntax; they are considered easier for a human to write a correct program in but usually are not as fast as more machine-oriented languages, e. g., imperative languages such as C. It is an objective of programming language researchers for functional and logic programs to run as fast as imperative programs. Current research into solving this problem follows two directions: 1) the construction of hardware or virtual machines to execute those languages (e. g., [Fuch87] and [Thak87]), or 2) the development of a correctness preserving method to automatically translate these languages into more machine-oriented languages (e. g., [Burs77] and [Mann79]). We choose the latter approach for the following reasons:

1) We desire a general purpose software specification method. This implies that the developed systems should run on conventional architectures, not special-purpose ones. Therefore, constructing a hardware machine is impractical. Virtual machines are equally unattractive because of the overhead involved in running them on a conventional machine.

2. Translation into more machine-oriented languages has the advantage that the software systems which are developed in this way can interface with existing software already developed by hand. This makes the methodology more generally useful. For example, if we already have part of a software system, we can use this technique to generate the rest, thereby reusing what has already been done.

1.2. Two-Level Grammar as a Specification Language

The Two-Level Grammar (TLG) specification language we propose follows the functional and logic programming paradigm espoused in the previous section. However, it has been shown in previous studies [Edup89] to have a number of advantages over conventional functional and logic programming languages. TLG most closely resembles Prolog in style but has been shown to be suitable as a functional

language ([Brya88a], [Edup89]) and as a data flow language [Brya86a], thus allowing for possible parallel execution. Besides the versatility of TLG in fitting various computation models, its natural language feature is perhaps the most important from a software development viewpoint [Brya88b].

Two-Level Grammar is a declarative specification language. Since the language specifies only what is desired, rather than how to compute it, it is much simpler to write and easier to prove the consistency with the user's requirements. TLG programs are self-documenting; a non-computer trained user can read the TLG specification by himself and easily understand what the system will do for him, so that he can more easily determine whether it matches his requirements. TLG specifications are also multi-layered, following the general principles of top-down design. The user can therefore view the system at whatever level of abstraction he wishes, thus facilitating the communication between the system designer and the user. This benefit can not usually be found in the other specification languages now in use. Because of the unique integration of natural language with functional and logic programming in TLG specification, a number of new implementation techniques must be developed in order to facilitate rapid prototyping of the system and automatic generation. We have the surprising result that the natural language part of the specification may actually be helpful in optimizing the logic of a TLG specification, for example, in resolving nondeterminism and directionality of variables.

The original Two-Level Grammar model may be thought of as a logic programming model where the name of the predicate is written in infix notation and distributed among the variables to read like a sentence in natural language. Queries must match the rules exactly in order to be satisfied. In order for systems developed from TLG specifications to be more convenient, we relax this requirement to allow approximate matching of rules. That is, if the natural language construction of the predicate name is not totally exact, it can still match with the correct rule. This is very advantageous in carrying out the following tasks:

1) Rapid prototyping. This is usually done in an interactive mode. If TLG queries must exactly match rules, then there is a high likelihood of errors. Particularly if a novice user is evaluating the system, he may not know the exact way in which rules are specified although he may have a general idea of the keywords and variables. Allowing approximate matching provides a form of natural language interface to the prototype.

2) Interfacing between modules. In a large software system, the specification will typically be developed by a team, rather than a single person. Each member of the team may develop specifications for modules which interface with other modules being specified by other team members. A common problem in this scenario is that the team members may not always specify

the interface with other modules correctly. Approximate matching allows these rough interfaces to be more smoothly integrated.

3) Natural language front end. Many applications, such as database and knowledge-base systems, require a natural language front end. The current difficulty in using natural language is the restrictiveness of the domains over which it may be applied. Two-Level Grammar provides a flexible shell in which such systems can be constructed for arbitrary domains. The system is expressed in TLG and then may be queried using the TLG form of natural language. The domains will be built in as part of the specification.

1.3. Implementation of Two-Level Grammar

There are three main approaches to implementing Two-Level Grammar: 1) interpretation as proposed by Edupuganty [Edup85], 2) preprocessing into Prolog, and 3) transformation into a machine-oriented language. The latter two approaches will be developed in this dissertation. Edupuganty's interpretation is an effective mathematical model of TLG execution. It is based upon finite automata and context-free parsing techniques used for formal language recognition, with an elementary unification mechanism for handling logical variable instantiations. The primary disadvantage of this technique is that the pattern matching procedure must be executed repeatedly for every query and sub-query generated during the interpretation. This is compounded by the fact that context-free parsing for unambiguous grammars is at best $O(n^2)$.

We first improved the existing TLG implementation technology by developing a method for translating the TLG specification into Prolog. This is an immediate improvement over the interpretation technique in that it can be regarded as a compilation. The structure of rules need only be parsed once instead of being processed each time the rule is invoked at execution time. Furthermore, the translation can take advantage of existing Prolog interpreters and compilers in providing additional optimizations. The main advantage of our technique is that we can determine directionality of variables as well as functionality of the predicates, facilitating the insertion of cuts into the generated Prolog programs. Such cuts greatly improve the efficiency of the generated Prolog programs. The results of the realization of this preprocessor are: 1) a method of interfacing TLG specifications with existing Prolog programs, and 2) immediate rapid prototyping of the specifications.

Our main objective in the implementation of TLG specifications is the production of high performance target code since this accomplishes automatic program generation. For this task, we require the use of transformation methods. The target language used by our transformation methods is C, which may be compiled into efficient machine code using existing compilers. C has become a

standard implementation language for large software systems and as a result is widely portable. Using C as a target language allows us to take advantage of this, thus enhancing the utility of our transformation system. Furthermore, our target programs may interface with many existing C programs, facilitating software reuse. Software maintenance is also enhanced in that modules that need to be redesigned can be specified in TLG and then transformed into the necessary C code. C is efficient because its structure and syntax are relatively machine-oriented. However, this means that if we use C to write a program directly, we must not only pay attention to its syntax and data structures, but also be concerned about whether we write the correct instructions on how the machine is going to solve the problem in detail. If we write a parallel C program, the work is even more. Our transformation methodology allows us to get the advantages that C offers as a programming language without having to spend the detailed effort required to write C programs.

To accomplish our research goals, we develop the theory of a transformation system to automatically generate efficient C programs from Two-Level Grammar formal specifications. The advantages of using transformation methodology are:

1) Ease of correctness proof. If the specification is correct and the transformation is correct, then every generated program is automatically correct. Since the transformation program can be reused, we translate the proof of correctness of the C program into the proof of correctness of the specification. The latter is usually much easier to do.

2) To maximize the reusability of the programming techniques. Every time a programmer writes a C program, he needs to go through the same rules for constructing the program. He has to reapply the same knowledge that he has about C repeatedly. The automatic transformation helps to save development time and make the resulting software more reliable.

3) To guarantee the quality and efficiency of the generated program. Different persons write different quality programs. Even the same person may write different quality programs at different times. If the transformer is of good quality, the quality of generated programs will have a minimum standard which is always met.

4) To make the specification language run more efficiently, if the specification is executable. It is very common that specification languages used in practice are executable, that is, they are themselves implementation languages. This is true of functional and logic programming languages, for example, and also Two-Level Grammar. By developing techniques for transforming the language into a more efficient target language, we have also increased the efficiency of the language.

1.4. Two-Level Grammar Specification of Database and Knowledge Base Systems

The Two-Level Grammar specification language has been made much more general than the notation used in Edupuganty's language definition studies, especially as a shell for database and knowledge-base systems. We pointed out earlier that the TLG shell offers a natural language interface within the framework of the TLG language. Besides the natural language interface, TLG offers many other advantages to the technology. When used as a query language, it offers the advantages of both functional and logic programming languages in expressivity of the queries. Logic programs have long been used in database systems, particularly for deductive and knowledge-base systems. The use of functional languages in this capacity is a much more recent research topic for which TLG is a timely solution.

Our main result is to show that when TLG is used as a specification language for such systems, it may also be used as an elegant query language for accessing information in the database. The query language is very close to natural language and can be made to fit the domain of the problem application very well. Using a form of natural language deducible from the system specification, a user can easily query the database and construct new rules. Furthermore, the robustness of the natural language which may be understood by the system increases as the domain becomes more specific. We view the relationship between TLG and databases in several levels. In the simplest example, epitomized by the use of the traditional TLG model, a TLG specification defines a database which may then be queried only according to the structure of the facts and rules in the database. This restrictiveness may be greatly loosened by our approximate matching methodology. Using approximate matching, the more restrictive the domain of the database, the more flexible the structure of the queries can be since the approximate matching has less ambiguity in a sparse collection of predicate/relation names. On the other hand, as the domain becomes wider, the rules will become correspondingly denser and queries must be more exact in order to avoid ambiguity. We call this technique *multi-level natural language understanding.*

Our first database application is the implementation of an SQL relational database management system with "intelligence." This system supports the basic SQL queries but also allows the user to formulate more general forms of SQL queries, including queries over universal relations, SQL queries embedded into natural language, and pure natural language queries. The result is an embedding of SQL into a sophistocated database management system with deductive capabilities and a general-purpose natural langauge interface. As with the other components of our research, which allow convenient interfaces with exisiting systems, it is also possible for existing SQL databases to be incorporated into our system allowing us to build extensions or make modifications to existing SQL systems.

The second application is to an object-oriented database system with learning capabilities. The system accepts natural language queries from users and attempts to match the queries with its TLG rule-base. If an appropriate match can not be found, then the information from the query is added to the rule base. Hence, the system learns new rules from user queries. During the acquisition of the new rules, the new information is added into the database using an object-oriented structure similar to a semantic network. We therefore show that TLG is general enough to represent these type of graphical data structures.

1.5. Two-Level Grammar Specification of Language Implementation Systems

There are many areas of programming language systems implementation that were not addressed by Edupuganty's original work. His specification of denotational semantics was an elegant way of representing operational semantics in a denotational way. This facilitated implementation by interpretation. However, there was no good way to use the specification for compiler development. We have developed a more practical way of expressing denotational semantics specifications and have developed a complete denotational semantics for a subset of the Ada programming language. The denotations include both static and dynamic semantics. The nature of the TLG specification is similar to an attribute grammar for producing the desired type checking and code generation in the form of lambda calculus expressions (e.g. see [Chur41]). The existing implementation methods for TLG can then be applied to produce a compiler for our subset of Ada which generates lambda code. This approach to automatic compiler generation works for any language which is specified using our method.

Lambda expressions produced by denotational semantics-directed compilers or compilers for functional languages are usually very inefficient to execute but the technique of graph reduction (e.g. see [Peyt87]) offers a time and space efficient method of both interpreting and compiling lambda expressions. Even though the specification of graph theoretic problems is somewhat alien to the logic programming style of TLG, as were the object-oriented data structures mentioned in the previous section, due to the lack of pointers for constructing such structures, we have completely specified this process and shown that our transformation techniques can translate logic specifications without pointers into C programs which take advantage of pointers to eliminate copies of shared data structures. Using TLG, we have specified a variety of virtual machines for the execution of lambda code, using the evaluation strategies of 1) strict evaluation, 2) lazy evaluation using tree reduction, 3) strict and lazy combinator reduction, 4) fully lazy evaluation using graph reduction, and 5) compilation into supercombinator programs. Any of these can be used as the back-end for compilers produced by our denotational semantics specifications. Therefore, TLG can be used as a specification method for all

aspects of compiling and the implementation methods developed in this dissertation advance the field of automatic compiler generation.

1.6. Summary

The rest of this thesis will develop the idea in more detail and more clearly. We begin by introducing desirable properties of specification languages and the theory of program transformation in Chapter 2. Chapter 3 discusses Two-Level Grammar in detail, TLG as a logic programming language, and show why TLG is a good specification language. On the other hand, TLG is difficult to implement. In Chapter 4 we discuss the three main ways in which TLG implementation can be realized: interpretation, preprocessing into Prolog, and automatic transformation into C. Chapter 5 illustrates significant software systems which have been developed using the TLG specification methodology.

CHAPTER II

FORMAL SPECIFICATION AND TRANSFORMATION METHODOLOGIES

The traditional software life-cycle may be characterized by the diagram shown in Figure 2.1.

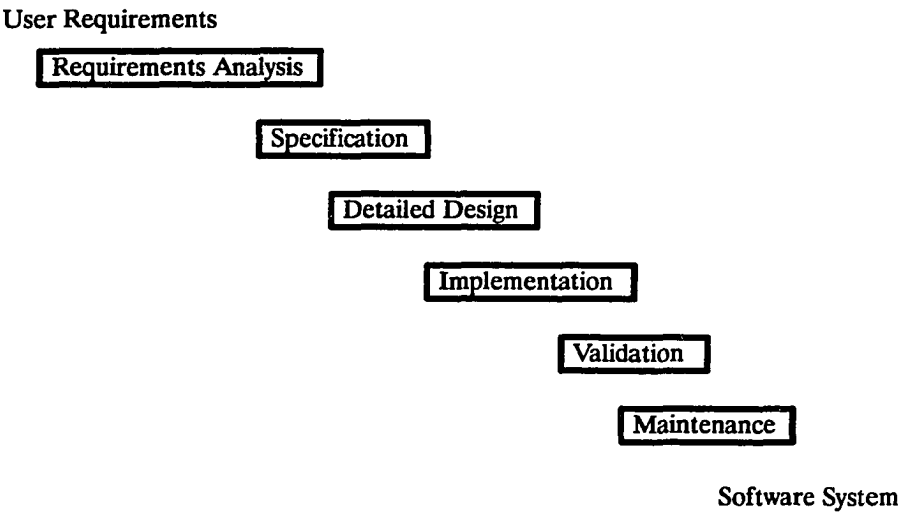

Figure 2.1. Software Life-Cycle

Currently, much of the life-time of a software system is spent in the maintenance phase, usually resulting from "bugs" in the original implementation which are detected over time with use of the software. By "bug" we usually mean a piece of code in the implementation which does not conform to the original specification, although there may also be flaws in the design as well, and it is even possible that the specification does not meet the user's requirements. Of these potential problems, validation only addresses the detection of "bugs" and it is well-known that it can not even detect all of these. Unfortunately, formal verification is not at present a practical reality. What is much more feasible is the automatic implementation of software from the specification. Such an "automatic programming" system would intend to avert the steps of detailed design and implementation, guaranteeing that the latter does follow its specification, thereby also eliminating the need for validation and a major portion of maintenance. The function of this type of system, also called an automatic program generator or program synthesizer, is shown in Figure 2.2.

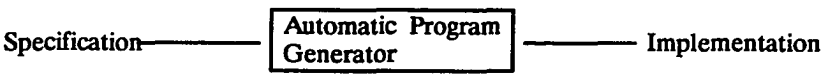

Figure 2.2. Automatic Program Generation

In order for this type of system to be realizable, it is necessary that the specification be formal. That is, it should be precisely defined in terms of a formal language with well-defined syntax and semantics.

Our objective in this thesis is to automate the software design and implementation process, at the same time providing a better tool for requirements analysis. We propose Two-Level Grammar toward reaching this end. In this chapter, we study the properties of formal specifications and survey major existing specification languages, leading up to why TLG is the desired specification language.

2.1. Properties of Formal Specifications

The goals of a formal specification are:

1. To be a reference for software designers, facilitating improved software design through detection of incompleteness, inconsistency, and ambiguity. The designer needs to be able to express the user requirements correctly, as he understands them.
2. To be a reference for users of the system. A concise, understandable specification clearly states capabilities and features. After the designer has written the specification, the user's ability to read and understand what has been written will greatly facilitate validation of the requirements analysis phase, detecting any inconsistencies between the requirements and specification as early as possible.
3. To be a reference for implementors, allowing implementation to rigorously follow the design specification. An implementor needs to understand the requirements of the user and plan of the designer. The specification serves as their primary medium of communication.
4. Correctness proofs of implementations are facilitated by formalization. It is impossible to reason about a specification, or anything else, unless it is well-defined. A formal specification provides the mathematical basis upon which to reason about the functionality of software systems.
5. Automatic implementation. The specification can be directly interpreted or automatically translated into executable code. Such an implementation can serve as a prototype of a production-quality system, or even be efficient enough to itself serve as the final product.
6. Facilitates modification of the system. Since the requirements of software systems may evolve over time, even an originally "perfect" software system may require maintenance. Such maintenance should be facilitated by the formal specification.

In order to meet these goals, we must develop a specification language to be sufficiently wide spectrum to allow a range of specification paradigms. However, there are a number of potentially conflicting objectives to be met in such a specification language:

1. Specifications based on natural language must be facilitated to be readable to designers, implementors, and users. However, natural language itself is too broad to be efficiently implemented for general types of problems, since it requires domain specific knowledge to be understood.

2. In a specification, abstract data types should be supported, facilitating object-oriented design. This suggests a functional and/or object-oriented specification. However, languages following these models are typically not close to natural language in readability, often requiring detailed knowledge of the underlying paradigm. This is in direct confrontation with our goal that the specifications be understandable to the novice user.

3. Specifications must be structured and mathematical in nature in order to serve as a basis for correctness proofs and automatic implementation. It is obvious that a natural language-like specification would be too general to meet these criteria. Of existing paradigms, functional and logic programming specifications appear to suit this goal the best.

4. We desire that specifications can be efficiently implemented on conventional architectures, at least for the purpose of prototyping. It is ideal if the specifications can be directly implemented. The most efficiently executable languages continue to be the imperative languages. However, these require such detail that they are not suitable for expressing a formal specification.

5. Parallel execution should be facilitated by the specification. Each paradigm we have mentioned offers a methodology for specifying parallelism, either implicitly or explicitly. However, how to express parallelism in high-level specifications is as yet a problem with no consensus solution.

2.2. Specification Languages

There are an endless number of specification languages, making an exhaustive comparison very difficult. We attempt to mention the major ones here. Formal specification languages tend to be *algebraic* or *operational*. Algebraic specifications are mathematical definitions of the properties that the software system should have, without specifying the implementation detail needed to achieve those properties. Major examples of algebraic specification languages include Obj [Gogu79], Clear [Burs81], VDM (the Vienna Development Method) [Bjor82], and CLU [Lisk86]. Operational specifications tend to be state-oriented, describing the system in terms of how it affects a notion of state. The main example of an operational specification language is Z [Abri79]. Outside of these main classifications, the paradigms

of functional [Turn85], logic [Kowa85], and object-oriented [Dahl87] programming have all been advocated as specification methodologies. These models may support either an algebraic or operational approach to specification. In general, one desires that the specification language which is used be executable in some manner. Executable specification languages have the following advantages: 1) incremental development of a specification is possible, 2) the specification is a prototype of the system it specifies, and 3) the behavior of the executable system can be checked for consistency with respect to the specification requirements.

Another major emerging theme in specification methodology is the notion of *transformational programming* [Pepp84], where not only a language but an entire theory of developing specifications into implementations is provided. The major embodiment of this idea is the CIP (Computer-aided, Intuition guided Programming) project at the Technical University of Munich [Baue85]. CIP-L, the specification language of the project, is wide-spectrum, allowing for the range of specifications between the formal problem description and efficient machine-oriented programs. An environment of transformation rules is provided to allow the software developer to initially develop his specifications at a high level and then move through stepwise refinement to the final production-quality code. The transformations are not completely automatic, instead relying on an experienced programmer's "intuition."

The goals of the above specification languages are facilitated by their reliance on formal logic. Logic is concise, unambiguous, and allows automated reasoning about the specifications. For the remainder of this discussion, we would like to concentrate on "pure" logic specifications. However, it should be mentioned that functional languages also have many of the same properties exhibited here, namely a declarative style, inductive (i.e. recursive) definition of computation rules, and constructive data structures, and hence can be considered as a paradigm for formal specification in their own right [Turn85].

The discussion, definitions, and examples which follow are adapted from [Hogg84]. A *logic specification* is intended to give a precise definition of every relation required in computing the desired solution. The *principal specified relation* is the relation computed by the program when given a *most general goal*, a call to the top level procedure in the program using only variables. An example of specifying the subset relation using logic is given below.

subset(X, Y) iff $(\forall U)(U \in Y$ if $U \in X)$
$U \in X$ iff $(\exists V)(\exists X')(X = V{:}X' \wedge (U = V \vee U \in X'))$
empty(X) iff $\neg(\exists V)(\exists X')(X = V{:}X')$

This specification is comprised of three *procedures*, subset, $\in$, and empty, with obvious semantic interpretations. Each of these rules consists of a *definiand*, the left side of the iff, and a *definiens*, the

right side. We regard subset as the principal specified relation. Therefore, the most general goal would be subset(X, Y). In contrast with Hogger's separation of formal logic specifications, which are considered unexecutable, from logic programs, which are, Kowalski [Kowa85] treats these as inseparable. His specification of the subset relation might be expressed by the logic program below, given using Prolog notation.

```
subset([], Y).
subset([V | X], Y) :- member(V, Y), subset(X, Y).
member(V, [V | Z]).
member(V, [U | Z]) :- member(V, Z).
```

Note that we have factored the empty predicate into the definition of subset. To contrast this approach with that taken in functional languages, consider the same specification coded in ML [Miln90].

```
fun subset [] Y = true
  | subset (V::X) Y = & (member V Y) (subset X Y);
fun member V [] = false
  | member V (U::Z) = if U = V then true else member V Z;
```

The main differences between the two approaches are to be seen in the definition of the member function. In Prolog, all cases not listed default to false. In ML, this must be explicitly specified. More significantly, Prolog allows unification of variables as a means to test for equality. ML and most other functional languages would require an explicit test for equality. Further discussion of the relationships between functional and logic specifications may be found in [Kowa85] and [Turn85]; [DeGr86] is an entire volume devoted to relevant language issues in combining the two approaches in programming.

2.3. Verification of Logic Specifications

Program verification is the process of formally proving that a program functions according to its specification. This means that a proof is provided that, for any input, the program will execute correctly. This is in contrast with *validation*, which uses a set of test cases to demonstrate that the program behaves correctly on those set of test cases, not necessarily all possible inputs. One of the principal advantages in using logic specifications is the theory of verification which has been developed for logic programs, which is essentially a theorem proving process. We summarize that theory from [Hogg84] here.

There are three conditions which must be satisfied for a program to be considered *correct*.

1. The program must be implied by the specification. That is, the program can not do more than the specification allows.
2. The program must be complete with respect to the specification. Everything that is specified must be included in the program.
3. The program must terminate when queried with the most general goal.

The process of logic program verification developed in [Hogg84] uses a form of program transformation called *definiens transformation* whereby a definiand may be replaced by a definiens in deriving new definitions of predicates that follow from the specification. The goal of the transformation is to construct a program where every predicate is defined by a rule of the form:

$r(\ )$ if B_i

which is then compatible with a Prolog rule definition. If the set of rules that are derived in this way are consistent with the program to be proven, then that program is judged to be verified.

Consider an example of the verification procedure for the Prolog subset program. The steps below outline the proof.

1. We start with the definiens of the subset specification: $(\forall U)(U \in Y$ if $U \in X)$.
2. The sub-equation $U \in X$ is a definiand of the membership rule so we can replace it with the definiens for that rule. This is sometimes referred to as *unfolding* in program transformation. The result is $(\forall U)(U \in Y$ if $(\exists V)(\exists X')(X = V{:}X' \wedge (U = V \vee U \in X')))$.
3. This expression may then be transformed according to the logical rule:

   A if $(\exists Z)(b(Z) \wedge c(Z)) \equiv \neg(\exists Z)b(Z) \vee (\exists Z)(b(Z) \wedge A$ if $c(Z))$

   Let A represent $U \in Y$, Z represent V and X', b(Z) represent $X = V{:}X'$, and c(Z) represent $(U = V \vee U \in X')$. Then we have

   $\neg(\exists V)(\exists X')(X = V{:}X') \vee (\exists V)(\exists X')(X = V{:}X' \wedge (\forall U)(U \in Y$ if $U = V \vee U \in X')$
4. We now can apply a procedure known as *folding* to replace the first part of disjunction with empty(X), since it is the definiens of the empty rule.

   $empty(X) \vee (\exists V)(\exists X')(X = V{:}X' \wedge (\forall U)(U \in Y$ if $U = V \vee U \in X')$
5. Next we decompose the inner disjunction as follows:

   $empty(X) \vee (\exists V)(\exists X')(X = V{:}X' \wedge (\forall U)(U \in Y$ if $U = V) \wedge (\forall U)(U \in Y$ if $U \in X'))$
6. Simplifying $(\forall U)(U \in Y$ if $U = V)$ to $V \in Y$ and folding $(\forall U)(U \in Y$ if $U \in X')$ to subset(X', Y) gives

   $empty(X) \vee (\exists V)(\exists X')(X = V{:}X' \wedge V \in Y \wedge subset(X', Y))$

   which corresponds exactly to the body of the Prolog subset rules, thereby proving their consistency with the specification.

To verify the rest of the program, we would have to perform similar procedures for the member rules. The result would be a proof of *partial correctness*. To prove *total correctness*, we also need to prove termination. This is the most difficult aspect of program verification as it is undecidable in the general case, being reducible from the halting problem for Turing machines (e. g., see [Mann74]). An informal argument may be used to explain that the above program terminates because every recursive call to the

subset and member further decomposes the list. As the list can not be infinite, we must ultimately reach the case where the list is empty, that is, the termination condition. More details about proofs of termination for inductive logical definitions may be found in [Mann74].

2.4. Program Transformation and Synthesis

The verification procedure of the last section used a form of program transformation to verify that Prolog rules could be derived from formal logic specifications. It should not be surprising that the same procedure can be used to *synthesize* those Prolog rules directly. The theory of logic program synthesis using definiens transformation is developed further in [Hogg84]. In general, program transformation ([Burs77], [Burs81], [Darl82], [Feat87], [Meer87]) is a means to mechanically develop efficient programs from formal specifications. It has been proposed as a methodology for automating the software development process. The art of program transformation combines the knowledge and techniques from programming languages, artificial intelligence, and software theory. The long range objective is to dramatically improve the construction, reliability, maintenance, and extensibility of software. An alternative to program transformation as a method of synthesis is to use natural deduction ([Mann79, [Mann80]). However, the general procedure is the same.

There are some similarities between compilation (e. g., see [Aho86]) and transformation. Just as compilation develops efficient machine code from source programs, transformation develops efficient programs from specifications. Generally saying, a transformation is the formal development of specification to implementation while a compilation is the formal development from implementation to machine execution. Their difference is decided by our viewpoint of machine language, as illustrated in the following text.

Traditional compilation is characterized by the following key restrictions:

1. Compilation is automatic. That is, once we begin, the process continues without any additional input from the programmer, and the product of the process is ready for execution.
2. The source language is limited to that which a compiler can "cover," that is, any program that is legal to express in the language should be compiled satisfactorily.
3. Compilation begins with the source program only, and needs no advice on how to proceed.

By relaxing combinations of the restrictions of compilation in certain ways, we get the major subcategories of transformation as below, from [Feat87].

1. *Extended compilation*, characterized by permitting advice from the user and partially relaxing the limits on the source language. The transformation system accepts not only the source program, but also guidance on how to do the transformation. The source language is extended, taking

advantage of the additional guidance to permit use of more expressive constructs. All guidance must be given at the very beginning, along with the program.

2. *Meta-programming*, characterized by relaxing the restrictions that the process be automatic and that it take no advice. The transformation process may involve significant interaction between system and programmer, meaning that advice can be provided both at the start of and during transformation. As a result, coverage of the specification language can be attained without significantly limiting the constructs of that language.

3. *Synthesis*, characterized by relaxing all language restrictions while attempting to retain the automatic and unadvised nature of compilation. A specification may be expressed in a style that gives no hint of a reasonably efficient implementation, hence the need to synthesize the program.

In summary, compilation develops efficient code from source programs. The result of the compilation is ready for execution (in particular, will not need to be further transformed or compiled) while transformation develops efficient programs from specifications. The result of transformation usually needs further compilation, or transformation in some cases. Figure 2.3 outlines the difference. Generally saying, a transformation is the formal development from specification to implementation, while a compilation is the formal development from implementation to machine execution. There is no significant difference between a specification and an implementation, except a specification is more expressive and less algorithmically detailed. Therefore, under some conditions, a compilation can be viewed as a transformation. A case study is illustrated in the following section.

2.5. Overview of Transformation Strategies

It is clear that the nature of the specification to be transformed, the specification language, the target programming language, and the efficiency requirements may all influence the development from specification to program. Unfortunately, little is known about how to deal with all these issues at once in anything other than an ad-hoc manner.

Research that has addressed, but not solved, the problems at this strategic level includes:

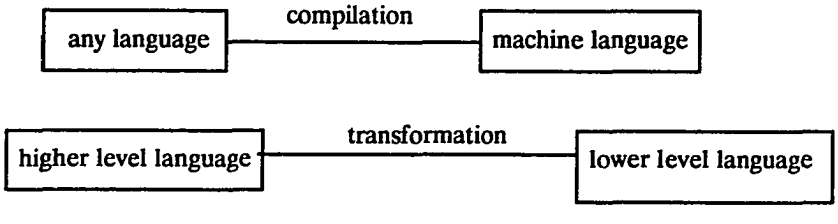

Figure 2.3. Compilation versus Transformation

1. The structure of an applicative language specification is used to suggest an overall strategy for improvement in efficiency, where the elements of the strategy are applications of several transformation tactics [Feat82].
2. In an efficient implementation of a given specification, the optimal control structures and data structures may be mutually dependent.

The following techniques are usually considered to help the transformation.

1. Techniques to introduce or alter the computation structure:
   a) *Fusion*: the merging of nested function calls (in the context of recursion equation programs) or consecutive loops (in the context of iterative programs), where the first function call / loop builds up a composite object which is used by the second function call / loop.
   b) *Tupling*: merges parallel function calls or loops so that their independent computations may be performed collectively, and so that their common computations need not be repeated.
   c) *Generalization* and *specialization*: *Generalization* is a technique to solve a problem by considering a more general one. *Specialization* is in some ways the complement of generalization. Its idea is to take advantage of the context in which some value is being computed to tailor that computation to the context, with the objective of realizing a more efficient computation of the same value.
   d) *Filter promotion* is potentially applicable to a specification or a portion of a specification in "generate and test" form. Its effect is to merge the filter testing into the generation process, and it is thus a special case of *fusion*.
   e) *Removal*: instances of forms of computation that are convenient in specification or intermediate stages in development, but can typically be replaced by lower-level more efficient code, are often used as goals for *removal*. The most common *removal* are removing *recursion* and *nondeterminism*.
   f) *Precomputation*: When some but not all of the data to be given to a program are known in advance, the program may be partially processed with that known data to give a "residual" program, which can be run later on the remaining data. This goal is often called *partial evaluation*.
2. Techniques to introduce or alter the maintenance and retrieval of data:
   a) *Memoizing* avoids recomputation of expressions by storing the results of evaluations the first time they are computed, and retrieving the stored values upon subsequent requests for the same computation.

b) *Tabulation* is a specialized form of memoizing where the context is a goal of computing a single result or table of results.

c) *Formal differentiation* aims only to maintain computed results incrementally, as the values upon which they depend gradually change.

3. Techniques to manipulate and implement abstract data types and transformations on abstract data types.

We may use many of these techniques in developing a transformation system from Two–Level Grammar specifications into C target programs.

CHAPTER III

TWO-LEVEL GRAMMAR SPECIFICATION LANGUAGE

Two-Level Grammar (TLG) was introduced by van Wijngaarden [Wijn65] as a formal grammar for the purpose of language specification. The name "two-level" comes from the fact that TLG consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing machine [Sint67]. TLG was first used in specification to successfully define the complete syntax (both context-free and context-sensitive aspects) of the programming language ALGOL 68 [Wijn74]. It has also been applied in other language definition studies ([Marc76], [Paga81]) on a smaller scale, especially in the book by Cleaveland and Uzgalis [Clea77] completely devoted to the subject of TLG for language specification. This book introduces the notion of using TLG in an interpretive specification of semantics.

The first attempt to develop practical implementations of TLG was made by Wegner [Wegn80], who was able to develop a parsing technique for a restricted class of two-level grammars. Wegner's parser allowed programs generated by TLG language specifications to be verified with respect to syntactic correctness, including for context-sensitive syntax. However, the semantics of programming languages did not become implementable until the TLG interpretation method of Edupuganty and Bryant [Edup85], which showed how operational semantics could be implemented. This was later generalized to other methods of specification, including axiomatic semantics [Brya86b] and denotational semantics [Edup87], which led to the ultimate use of TLG as a programming language in its own right.

Maluszynski [Malu84] proved the equivalence between a class of two-level grammars and logic programs. However, TLG did not become useful for practical programming until the functional and logic programming models of Bryant and Edupuganty ([Brya88a], [Brya88b], [Edup89]). They showed that the two-levels of the TLG notation could be formulated as a set of domain definitions and the set of function definitions operating on those domains.

3.1. Introduction to Two-Level Grammar Language

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. Domains may be structured as linear data structures such as lists, sets, or singleton data objects, or be configured as tree-structured data objects. The

standard structured data types of product domain and sum domain may be treated as special cases of these. Type declarations have the following form:

IDENTIFIER-1, IDENTIFIER-2, ..., IDENTIFIER-m :: data-object-1; data-object-2; ...; data-object-n.

where each data-object-i is a combination of domain identifiers, singleton data objects, and lists of data objects, which taken together form the type of **IDENTIFIER-1, IDENTIFIER-2, ..., IDENTIFIER-m**. Note that if $n = 1$, then the domain is a true singleton data object, whereas if $n > 1$, then the domain is a set of the n objects. Syntactically, domain identifiers are written entirely in upper case letters, with underscores for readability (e. g., **INTEGER_LIST, SYMBOL_TABLE**, etc.), and singleton data objects are finite lists of English words written entirely in lower case letters (e. g., sorted list). A list structure is denoted by a regular expression or by following a domain identifier with the suffix _LIST. For example, the following type declarations respectively define a list of integers and a compiler symbol table configured as a list of records, each with three fields: id, type and value.

INTEGER_LIST :: {INTEGER}*.
SYMBOL_TABLE : {id IDENTIFIER type TYPE value INTEGER}+.

Note that the first of these is redundant since **INTEGER_LIST** already has the assumed meaning. Following conventional regular set notation, * implies a list of zero or more elements while + denotes a list of one or more elements. As in any programming language, readability is promoted through the use of appropriate names for identifiers. Furthermore, there exists a predefined environment, defining such domains as **INTEGER, BOOLEAN, CHARACTER, STRING**, etc., in the obvious ways. The main difference between list structures and tree structured domains is whether the defining domain identifier declaration is recursive or not. Recursive domains are slightly more powerful in that they allow "context-free" data types to be defined, such as expression strings with balanced parentheses as in the following example:

EXPRESSION :: (EXPRESSION).

The context-free grammars defining such data types must be unambiguous.

The function definitions are the main part of a TLG program. Their syntax allows for the semantics of the function to be expressed using a structured form of natural language. Function definitions take the form:

function-predicate : sub-function-1, sub-function-2, ..., sub-function-n.

where $n \geq 0$. Function predicates are a combination of English words and domain identifiers, which correspond to variables in a conventional logic program (e. g., divide a list of numbers **INTEGER_LIST** into sub-lists **INTEGER_LIST1** and **INTEGER_LIST2**). The use of English in the function predicate may be regarded as a form of infix notation for functions, in contrast with the customary prefix forms of most

other programming languages. This greatly enhances readability of programs. Each sub–function on the right hand side of a function definition should correspond to a function predicate defined within the scope of the TLG program. The most important aspect of function definitions is that every domain identifier with the same name is instantiated to the same value, as in Prolog. This is called *consistent substitution*. If variables have the same root name but are subscripted, then the subscripts are used to distinguish between variables. A subscripted variable V1 will then be different from a variable V2 and the two can have different values. However, they will be of the same type, namely type V.

The following is an example of a TLG specification for determining if a character string is a palindrome.

Domain Declarations

LETTER :: CHARACTER.
LETTERS :: STRING.

Function Definitions

string EMPTY is a palindrome.
string LETTER is a palindrome.
string LETTER LETTERS LETTER is a palindrome : string LETTERS is a palindrome.

Following consistent substitution, the two occurrences of **LETTER** in the third rule must have the same value. Note that the domain declarations are somewhat redundant since the types **CHARACTER** and **STRING** may be used in the function definitions for **LETTER** and **LETTERS**, respectively. However, we prefer that the writer of the specification not have to concern himself excessively with type declarations. Therefore, we propose to omit the domain declarations and specify only the function definitions. The types must then be inferred from the function definition rules as in modern functional languages. Further discussion of TLG type inferencing will take place in Chapter 4.

3.2. Type Structures

The notations for domain declarations allow us to define the principal domains of type theory, namely product domains (tuples), sum domains (discriminated unions), and function domains. Our types are modeled after the ML functional programming language [Miln90], with domains D_1, D_2, ..., D_n structured as

1) A product domain $D = D_1 x D_2 x ... x D_n$ with selection operation $\downarrow m$ to select the m–th component of D. In TLG, this may be represented as a tagged tuple (e. g., **D : d1 D1 d2 D2 ... dn Dn**) with the selection operation being directly by name. This is more general than usual product domains which allow selection by number only. For example, a symbol table record was defined earlier as a product of identifier name, type, and value.

   SYMBOL_TABLE_RECORD : id IDENTIFIER type TYPE value INTEGER.

A selection operation to select a field from this record would be of the form:

select id from id IDENTIFIER type TYPE value INTEGER giving IDENTIFIER.
select type from id IDENTIFIER type TYPE value INTEGER giving TYPE.
select value from id IDENTIFIER type TYPE value INTEGER giving INTEGER.

This selection operation generalizes to any product, with either named or numbered components by the following rules:

select TAG from TAG NOTION FIELD_LIST giving NOTION.
select TAG1 from TAG2 NOTION2 FIELD_LIST giving NOTION1 :
select TAG1 from FIELD_LIST giving NOTION1.

2) A sum domain $D = D_1 + D_2 + ... + D_n$ with injection operation inD D_m ($d_m \in D_m$) and projection operation d | D_m ($d \in D$). This is represented as a union of TLG domains which may be tagged if necessary to distinguish between non–disjoint domains (e. g., **D : d1 D1; d2 D2; ...; dn Dn**) with the injection operation consisting of adjoining the tag to the domain element to make it an element of the domain D and the projection operation by pattern matching. For example, we can define the set of all finite lists of elements from domain D by a sum domain as in [Schm86] as follows:

FINITE_LIST_OF_D :: null list; NON_NULL_LIST_OF_D. {D^*}
NON_NULL_LIST_OF_D :: D; D NON_NULL_LIST_OF_D. {D^+}

Now the constructor function for lists (i.e. cons (d, l)) can be defined as the following sequence of rules which project the list l into the domain D^n and then inject the result of the function into domain D^{n+1}.

cons D and null list into D.
cons D and NON_NULL_LIST_OF_D into D NON_NULL_LIST_OF_D.

3) A function domain $D_1 \rightarrow D_2$ with abstraction operation fun x ⇒ E ($x \in D_1$, $E \in D_2$), defining a function with argument x and body E, and application operation f x ($f \in D_1 \rightarrow D_2$, $x \in D_1$). TLG can define any function or function application which is expressible by the λ–calculus [Chur41]. The abstract syntax of λ–calculus expressed in TLG is given below.

LAMBDA_CONSTANT :: {definition of constants}.
LAMBDA_VARIABLE :: {definition of variables}.
LAMBDA_APPLICATION :: LAMBDA_EXPRESSION LAMBDA_EXPRESSION.
LAMBDA_ABSTRACTION :: fun LAMBDA_VARIABLE ⇒ LAMBDA_EXPRESSION.
LAMBDA_EXPRESSION :: LAMBDA_CONSTANT; LAMBDA_VARIABLE; LAMBDA_APPLICATION; LAMBDA_ABSTRACTION.

Two–Level Grammar also supports polymorphic types. The predefined domain **NOTION** represents the universal domain of values. Any operation defined on domain **NOTION** is considered to be polymorphic; it may be applied to values of any type. In most cases, types may be inferred from their usage in function definitions. The actual usage of domain declarations is needed only to clarify the types of those functions which are defined in such a way that the types of variable can not be deduced from their usage in the function.

3.3. Built-In Functions

There are also several predefined functions in the TLG programming language. Primitive operations, such as arithmetic and logical operations, are defined by predicates of the form **where OPERAND1 operator OPERAND2 is RESULT.** The list operations head and tail may be implicit and called through pattern-matching or be explicit; the append predicate is referenced explicitly. Control operations such as **if-then-else** and **endstmt** to provide function sequencing are also provided as predefined functions. Perhaps the most powerful feature provided as a TLG primitive is the Zermelo Frankel (ZF) set expression, which allows the implementation of abstract sets such as $\{x \mid x \in D\}$. We illustrate this with a TLG example of the quick sort algorithm using generators developed by Turner [Turn90].

Domain Declarations

```
PIVOT :: INTEGER.
INTEGERS_LESS, INTEGERS_GREATER, SORT_LIST :: {INTEGER}*.
```

Function Definitions

```
quick sort EMPTY_LIST into EMPTY_LIST.
quick sort PIVOT INTEGER_LIST into SORT_LIST1 PIVOT SORT_LIST2 :
      generate all INTEGER1 from INTEGER_LIST
            condition INTEGER2 < = PIVOT giving NUMBERS_LESS,
      quick sort NUMBERS_LESS into SORT_LIST1,
      generate all INTEGER2 from INTEGER_LIST
            condition INTEGER3 > PIVOT giving NUMBERS_GREATER,
      quick sort NUMBERS_GREATER into SORT_LIST2.
```

The TLG primitive function generate-all is used to produce the list of integers less than the pivot (NUMBERS_LESS) and the list of integers greater than the pivot (NUMBERS_GREATER). Further discussion of ZF expressions and their implementation in TLG is given in [Brya88b].

The evaluation rules for TLG lambda expressions described in the previous section are also primitives of the language. The following function evaluates a lambda expression.

```
reduce LAMBDA_EXPRESSION1 to LAMBDA_EXPRESSION2.
```

These rules can be completely defined using TLG functions, as will be shown later when we give a complete set of lambda reduction rules expressed in TLG. The efficient implementation of lambda reduction rules will be described in Chapter 5.

3.4. Determinism versus Nondeterminism

Function predicate definitions, hereafter called *rules*, may be either deterministic or nondeterministic. Nondeterminism can arise in two ways: 1) an assignment of values to variables is not unique, or 2) more than one rule may be applied. The rule heads below, adapted from [Edup87], illustrate both types of nondeterminism:

partition INTEGER_LIST INTEGER_LIST into INTEGER_LIST INTEGER_LIST.
partition INTEGER_LIST1 INTEGER_LIST2 into INTEGER_LIST1 INTEGER_LIST2.

The first rule accepts a list whose first half is the same as its second half and returns the content of each half in **INTEGER_LIST** and the second rule uses an arbitrary partition to return part of the list in **INTEGER_LIST1** and the rest in **INTEGER_LIST2**. If a function call of the form **partition 1 2 3 1 2 3 into INTEGER_LIST3 INTEGER_LIST4** is given, then both rules may be matched. In the case of the first rule, the variable **INTEGER_LIST** is instantiated to **1 2 3**. In the second rule, the assignment of values to **INTEGER_LIST1** and **INTEGER_LIST2** is not unique. There are a total of eight different assignments possible, considering that the lists may also be empty.

Nondeterminism can often be eliminated from TLG specifications. Besides the partition function just given, all of our previous TLG specifications have been deterministic. In such examples as palindrome and quick-sort, the variables of the different rules are disjoint and hence can never be instantiated to the same values. In logic programming, this is one desirable way of eliminating nondeterminism. The other method is through the use of an if-then-else conditional statement. Figure 3.1 shows a traditional quick-sort algorithm, adapted from [Brya88a]. The TLG is nondeterministic in the **split** function since the last two rule heads can not be distinguished. However, since each of these rules differs only in their initial guard predicate, the two rules can be combined into one using the guard as the condition of an if-then-else statement. The resulting function is shown in Figure 3.2.

Domain Declarations

```
PIVOT :: INTEGER.
INTEGERS_LESS, INTEGERS_GREATER, SORT_LIST1, SORT_LIST2 :: {INTEGER}*.
```

Function Definitions

```
quick sort EMPTY_LIST into EMPTY_LIST.
quick sort PIVOT INTEGER_LIST into SORT_LIST1 PIVOT SORT_LIST2 :
        split INTEGER_LIST with PIVOT into lists INTEGERS_LESS and INTEGERS_GREATER,
        quick sort INTEGERS_LESS into SORT_LIST1,
        quick sort INTEGERS_GREATER into SORT_LIST2.

split EMPTY_LIST with PIVOT into EMPTY_LIST and EMPTY_LIST.
split INTEGER INTEGER_LIST with PIVOT
                into INTEGER INTEGERS_LESS and INTEGERS_GREATER :
        where INTEGER < = PIVOT,
        split INTEGER_LIST with PIVOT into INTEGERS_LESS and INTEGERS_GREATER.
split INTEGER INTEGER_LIST with PIVOT
                into INTEGERS_LESS and INTEGER INTEGERS_GREATER :
        where INTEGER > PIVOT,
        split INTEGER_LIST with PIVOT into INTEGERS_LESS and INTEGERS_GREATER.
```

Figure 3.1. Two-Level Grammar for Quick Sort

```
split EMPTY_LIST with PIVOT into EMPTY_LIST and EMPTY_LIST.
split INTEGER INTEGER_LIST with PIVOT into lists INTEGERS_LESS and INTEGERS_GREATER :
    if INTEGER < = PIVOT then
        split INTEGER_LIST with PIVOT
            into INTEGERS_LESS2 and INTEGERS_GREATER2 endstmt
        append INTEGER with INTEGERS_LESS2 giving INTEGERS_LESS1
    else
        split INTEGER_LIST with PIVOT
            into INTEGERS_LESS2 and INTEGERS_GREATER2 endstmt
        append INTEGER with INTEGERS_GREATER2 giving INTEGERS_GREATER1
    endif.
```

Figure 3.2. Deterministic Two-Level Grammar for Quick Sort Split Function

3.5. Mode Analysis

TLG functions may also return values in one or more variables as in Prolog. However, in contrast to Prolog, where the syntax of input and output variables is uniform, TLG variable modes may be determined by an analysis of the natural language framework of the function definition. For example, in a TLG rule of the form:

divide a list of numbers INTEGER_LIST into sub-lists INTEGER_LIST1 and INTEGER_LIST2

it is clear that **INTEGER_LIST** should be input to the divide function and **INTEGER_LIST1** and **INTEGER_LIST2** are output variables. Using the TLG natural language vocabulary for mode inferencing method was first proposed in [Brya88b].

3.6. Sequential Interpretation

The sequential interpretation of a Two-Level Grammar program was first detailed in [Edup85] and has been refined in several subsequent papers. Our discussion is adapted from [Brya86a]. Sequential interpretation may be thought of as the top-down construction of a context-free grammar derivation tree. Each function is expanded left to right recursively (a depth-first, in-order traversal) by applying its subfunctions, treated as its children in the derivation tree, until all the leaves of the tree are calls to functions with no subfunctions. This signifies the completion of the interpretation. The derivation tree of the Two-Level Grammar represents the control flow of the program execution. Execution consists of matching each function call in a "sentential form," represented by a node in the tree, with a function definition and spawning new branches from the sequence of subfunctions in the function body. In a deterministic TLG, there will be at most one function definition corresponding to any function call. If there are more than one, the interpretation must be nondeterministic. During the execution of the TLG program there may be some variables whose instantiations are unknown during the expansion. These variables are instantiated using a process which is semantically equivalent to unification and may

be realized by passing a pointer to the variable down the tree. In TLG terminology, this instantiation is called *synthesis*.

The interpretation algorithm is formalized below. For simplicity we assume that the TLG being executed is deterministic.

procedure call (function)

1. Find the applicable function definition. This rule will be of the form

   function : subfunction-1, subfunction-2, ..., subfunction-n.

2. If any variables are synthesized by the rule match, create appropriate structures to represent the synthesized values.
3. Expand the derivation tree with **function** as the root and the branches being the n subfunctions.
4. For i = 1, 2, ..., n, *call* (**subfunction-i**)

3.7. Data Flow Parallel Interpretation

The notion that TLG could be executed using the dataflow model of computation has been detailed in [Brya86a]. Data flow interpretation of function application is a move from sequential evaluation of functions to massively parallel evaluation wherever possible. In data flow models of computation, a function may be evaluated as soon as all of its arguments are available. Furthermore, only essential data dependencies are specified. Functional composition will be sequential but independent function arguments may be evaluated in parallel. It is also true that in a Two-Level Grammar, only essential functional dependencies are specified. Dependencies occur only in the following cases:

1) All subfunctions in a function body are always dependent on their parent function.
2) Synthesized variables in a function definition are dependent on the subfunctions which synthesize them.
3) Two subfunctions, subfunction-i and subfunction-j, in the same function body have a dependency if either synthesizes a variable which is also referenced in the other. If subfunction-i synthesizes the variable X, for example, and X is used in subfunction-j, then subfunction-j is dependent upon subfunction-i. Note that the same variable can only be synthesized by one subfunction or consistent substitution would be violated.

The sequential interpretation algorithm does not exploit the lack of dependencies for parallel evaluation. An interpretation algorithm based on dataflow parallelism is given below:

procedure call (function)

1. Find the applicable function definition. This rule will be of the form

   function : subfunction-1, subfunction-2, ..., subfunction-n.

2. If any variables are synthesized by the rule match, create appropriate structures to represent the synthesized values.
3. Expand the derivation tree with **function** as the root and the branches being the n subfunctions.
4. For all i ($1 \leq i \leq n$) such that subfunction-i is not dependent on any subfunction-j ($1 \leq j \leq n$), *call (subfunction-i)* concurrently. This step is repeated until all subfunctions have been expanded.

Note that because of step 4, this procedure is concurrent.

As an example of the application of this procedure, consider the TLG representation of the function $f(x)=x^2-x*2+3$.

Domain Declarations

X, RESULT :: INTEGER.

Function Definitions

```
compute f of X giving RESULT :
      where RESULT1 = X * X,
      where RESULT2 = X * 2,
      where RESULT3 = RESULT1 - RESULT2,
      where RESULT = RESULT3 + 3.
```

When this function is called with argument X, all instances of X in the body will be given the appropriate value simultaneously, according to the consistent substitution principle. It is also clearly seen that the first two subfunctions are independent, and hence would be evaluated by our interpretation algorithm in parallel, but the third subfunction depends on both of them and the fourth depends on the third.

Given the TLG definition of a function, we may either interpret it sequentially or by the data flow method. In general, the results will be the same. However, there are some functions for which the call by-value and call-by-name computation rules used in sequential (i.e., left-to-right order) interpretation are not considered to be *safe* computation rules [Vuil73]. These functions may only be correctly evaluated using *fix-point* computation rules [Mann74]. However, using the data flow concept, a TLG may be interpreted using only fix-point, i.e. safe, computation rules. We will therefore be guaranteed to always compute the correct result.

In order for TLG evaluation to be safe, we need to slightly redefine our concept of function "matching." In particular, we need to allow *partial parameterization* [Paga79], which allows functions to be evaluated if only some of their input arguments are present. The result of this application will be another function. Partial parameterization may proceed until the resulting function is a constant. In

some functions having many arguments, the result of the function may depend upon only one argument making the other arguments useless. The idea of partial parameterization is not present in the data flow concept which does not allow a function to be evaluated until *all* its arguments are present. To extend TLG to encompass partial parameterization, we will allow a function body expansion to occur even if not all variables are instantiated, provided that this match can be done uniquely, without ambiguity. This will allow a function value to be computed if the *minimum* number of arguments are present to ensure correct computation. Partial parameterization of TLG combined with data flow interpretation effectively accomplishes delayed evaluation (i.e., call-by-name) and is an embodiment of the *full substitution* computation rule (e. g., see [Mann74]) which is a safe computation rule always giving the correct result.

The concept of TLG interpretation by partial parameterization will be illustrated in the following examples. Consider the following recursive function:

$f(0,y)=0$
$f(x+1,y)=f(x,f(x+1,y))$

By inspection we can see that the function calls itself repeatedly in the second case, continually decreasing the value of the first argument. Ultimately the first argument will become 0 and the result should be 1. By sequential interpretation, however, using call-by-value, this result can never be computed because the interpreter will attempt to evaluate the infinite recursion before applying the first function rule.

The above function is expressed in TLG notation as follows. For convenience, we will use the unary alphabet {i} for numeric computations. Note that **UNARY_INTEGER** and **UNARY_ZERO** are predefined domains.

Domain Declarations

```
X, Y, RESULT :: UNARY_INTEGER.
```

Function Definitions

```
compute f of UNARY_ZERO and Y giving UNARY_ZERO.
compute f of X i and Y giving RESULT1 :
        compute f of X i and Y giving RESULT2,
        compute f of X and RESULT2 giving RESULT1.
```

It can be seen that the value of Y is irrelevant in the first function definition. If X is i^0, then the rule will fire, regardless of the value of Y. If $X>0$, then the second function rule will fire, again regardless of the value of Y. If we allow these rules to be applied even when Y is unknown, we can compute the result of the function correctly. This is illustrated by the application of f(1,0) shown in Figure 3.3. This interpretation tree shows the infinite recursion which would occur if the tree were traversed depth-first

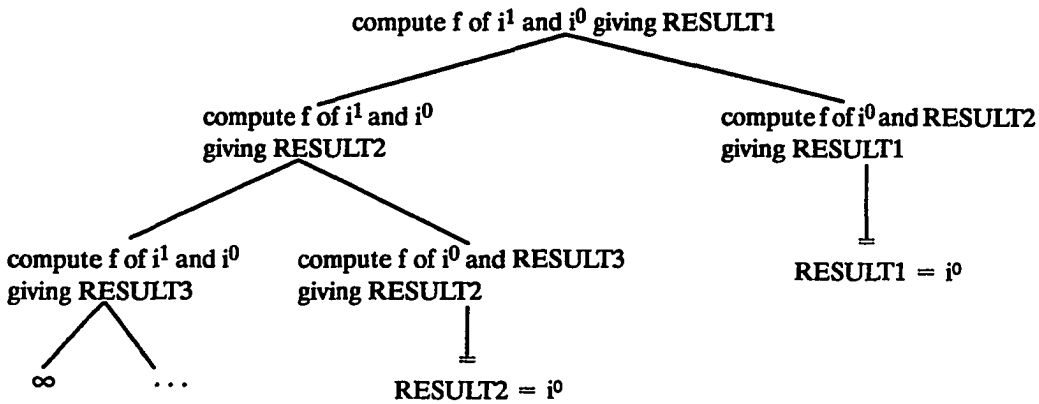

Figure 3.3. Two-Level Grammar Derivation Tree for f(1,0)

(i.e. sequentially, left-to-right). However, using the data flow concept with partial parameterization, we see that the right branch of the tree at level 1 can compute the result of the entire function. The dependency of **RESULT2** on the left branch is ignored because it is not essential - because the first argument is 0, the first function rule will fire regardless of the value of **RESULT2**. No matter what the arguments, the TLG will compute the value 1 for the function. This is the *least fix-point* of the function - f(x,y)=1.

It should be pointed out that the above function is evaluatable by the sequential call-by-name computation rule, but this rule is still not considered safe. The following example from [Mann74] shows its fault.

g(0,y)=0
g(x+1,y)=g(x+2,g(x+1,y))*g(x,g(x+1,y))

We observe that the left operand of the multiplication operand results in an infinite sequence of recursive calls. The right operand, however, decreases to the fix-point of 0, as defined by the first rule. Therefore, the least fix-point of g will be 0. This is clearly indicated in the TLG representation of g shown below.

Domain Declarations

X, Y, RESULT :: UNARY_INTEGER.

Hyperrules

compute g of UNARY_ZERO and Y giving UNARY_ZERO.
compute g of X i and Y giving RESULT1 :
compute g of X i and Y giving RESULT2,
compute g of X ii and RESULT2 giving RESULT3,
compute g of X i and Y giving RESULT4,
compute g of X and RESULT4 giving RESULT5,
compute RESULT3 times RESULT5 giving RESULT1.

An example of computing g(1,0) is shown in Figure 3.4. The infinite sequences are clearly seen but the fourth and fifth subfunctions allow termination and computation of the value 0 for RESULT1.

Two-Level Grammar may therefore be used as a data flow language with safe computation rules. This will later allow us to use TLG as a meta-language for denotational semantics and to transform TLG into parallel machine code.

3.8. Implementation of Rule Matching

An initial TLG interpreter has been prototyped by [Sund87]. The theory underlying this interpreter was developed by [Edup87] and will be described here. The process of expanding a function call (steps 1 and 2 in the above algorithms) involves pattern matching. Because TLG is based on the theory of formal grammars and is composed of a collection of strings which may be either regular sets or context-free languages, the most straightforward way of implementing the pattern matching is through finite automata and context-free parsing techniques. We shall illustrate the techniques through the example of a function to divide a list of integers:

divide INTEGER_LIST1 INTEGER1 INTEGER2 into INTEGER_LIST2 INTEGER2
and INTEGER_LIST3 INTEGER2 : ....

and associated function call:

divide 1 2 3 4 into INTEGER_LIST1 and INTEGER_LIST2

3.8.1. Finite Automata Techniques

Let us assume that all domain identifiers used in the example TLG program are defined by regular expressions as follows:

INTEGER_LIST :: {INTEGER}*.

To recognize the constants which comprise the function definition (e. g., **divide-into-and** in the above example) and the strings defined by regular domain identifiers, we generate finite automata (FA). A single FA would be sufficient for indexed variables (e. g., **INTEGER_LIST1** and **INTEGER_LIST2**) having the same domain type. For the function matching problem, the FA's corresponding to the different

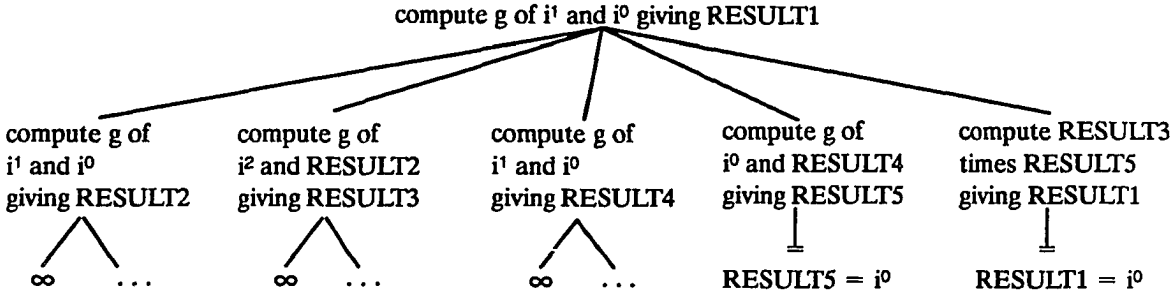

Figure 3.4. Two-Level Grammar Derivation Tree of g(1,0)

variables and constants involved can be concatenated to form another FA. Thus, for the example given, to recognize the string **1 2 3 4**, the FA's corresponding to the variables **INTEGER_LIST** and **INTEGER** can be combined to form a single nondeterministic FA (NFA) and then converted into a deterministic FA (e. g., see [Hopc79]). All of these transformations can take place statically during compilation of a TLG and hence do not affect run-time efficiency.

In the example, it is seen that the initial constant **divide** is common to both the function call and the function definition, and the matching is straightforward. It now remains to match the string **1 2 3 4** against the variables **INTEGER_LIST1 INTEGER1 INTEGER2**. Since the FA's for variables **INTEGER_LIST1** and **INTEGER1** can be combined to form an NFA, this too poses no difficulties. If the matching is successful, which it is in this case, the other instances of **INTEGER1** and **INTEGER2** in the function definition are instantiated to actual values due to consistent substitution. Continuing, the next matching problem is whether the variable **INTEGER_LIST1** in the function call matches **INTEGER_LIST2 INTEGER1** in the function definition. Here the variables **INTEGER_LIST1** and **INTEGER_LIST2** are synthesized variables (i.e. uninstantiated), whereas **INTEGER1** is instantiated to **3** (from the previous pattern matching). The pattern matching thus involves matching the synthesized variable **INTEGER_LIST1** with **INTEGER_LIST2 3** (note that **INTEGER_LIST2** is uninstantiated). This problem reduces to the question of whether the regular expressions represented by **INTEGER_LIST1** and **INTEGER_LIST2 3** are equivalent, which is known to be a decidable problem [Hopc79].

3.8.2. Context-Free Parsing Techniques

For explaining this method in the context of the example, we give the following context-free domain declaration:

INTEGER_LIST :: INTEGER_LIST INTEGER; EMPTY.

Now consider the problem of matching **INTEGER_LIST1 INTEGER1 INTEGER2** against **1 2 3 4**. To accomplish this parsing in a straightforward manner, a "dummy" start rule of the form **S** $\rightarrow$ **INTEGER_LIST INTEGER INTEGER** is created. Then the parsing reduces to the question of whether **S** $\Rightarrow$ **1 2 3 4**, which is straightforward by any traditional parsing technique for context-free grammars (e.g., see [Aho86]).

Let us now consider the case where the function variables are not instantiated. Consider the matching of **INTEGER_LIST1** against **INTEGER_LIST2 INTEGER1**. The matching proceeds by generating a parse tree for the variable **INTEGER_LIST**, and matching the frontier of a cut of the tree with the string being parsed (**INTEGER_LIST2 INTEGER1**). The situation can be visualized as shown in Figure 3.5(a). The matching of the tree cut with the input string has to be done breadth-first. If the match is not

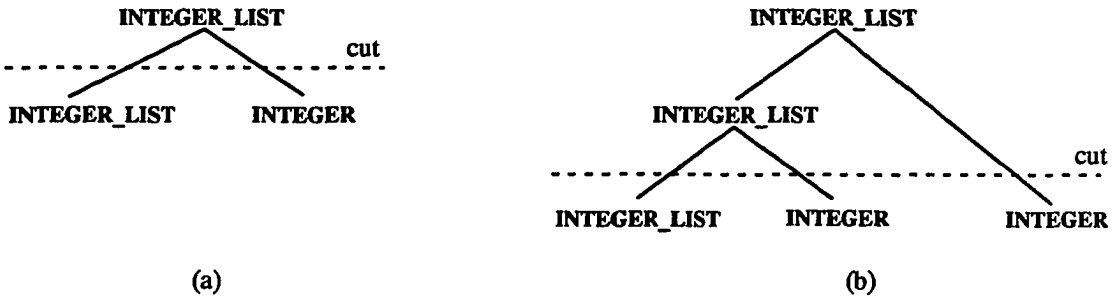

Figure 3.5. Two–Level Grammar Derivation Tree Cuts

successful, the parse tree is expanded another level (shown in Figure 3.5(b)). We are interested in matching only the structure of domain variable declarations and hence do not need to consider the subscripted variables as special cases.

3.9. Two–Level Grammar as Specification Language

Two–Level Grammar is an embedding of natural language into a functional programming language, generically called FPL, and Prolog, as illustrated by Figure 3.6. To show this more clearly, let us contrast TLG specifications with functional and logic specifications. We have seen the TLG description of palindromes. Consider the definitions of this problem using ML and Prolog given in Figure 3.7, neither of which includes the type checking mandated by the TLG specification. Both of these assume the existence of additional functions (e. g., reverse) in order to accomplish the designated task. In our opinion, the TLG specification is more readable and considerably more concise. For another example, consider the quick sort algorithm specified in TLG given earlier showing the inclusion of generators and list comprehensions in TLG. Corresponding examples in Miranda [Turn90] and Prolog

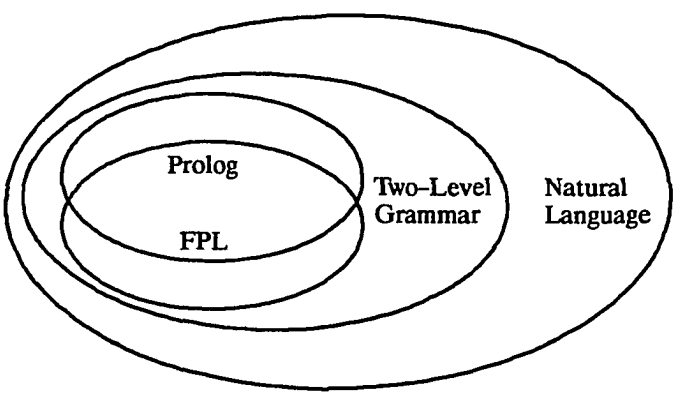

Figure 3.6. Comparison of Two–Level Grammar with Functional, Logic, and Natural Languages

ML

```
fun   palindrome [] = true
  |   palindrome (LETTER :: LETTERS_LETTER) =
        case LETTERS_LETTER of
          [] = > true
        | HEAD_OF_LETTERS_LETTER :: TAIL_OF_LETTERS_LETTER = >
            let val LETTER_LETTERS = reverse LETTERS_LETTER
            in (& ((hd LETTER_LETTERS) = LETTER)
                  (palindrome (tl LETTER_LETTERS)))
            end;
```

Prolog

```
palindrome([]).
palindrome([LETTER]).
palindrome([LETTER | LETTERS_LETTER]) :-
      reverse(LETTERS_LETTER, [LETTER | LETTERS]), palindrome(LETTERS).
```

Figure 3.7. ML and Prolog Specifications of Palindromes

[Cloc87] are given in Figure 3.8. The conciseness of the TLG version is comparable to that of Miranda because of the use of list comprehensions. However, we claim that the TLG version is considerably more readable because of the use of logical variables and natural language.

Miranda

```
quick_sort [] = []
quick_sort (PIVOT : LIST) = quick_sort [X | X <- LIST; X < = PIVOT] + + [PIVOT]
      + + quick_sort [Y | Y <- LIST; Y > PIVOT]
```

Prolog

```
quick_sort([], []).
quick_sort([PIVOT | LIST], SORTED_LIST) :-
      split(PIVOT, LIST, NUMBERS_LESS, NUMBERS_GREATER),
      quick_sort(NUMBERS_LESS, SORTED_LIST1),
      quick_sort(NUMBERS_GREATER, SORTED_LIST2),
      append(SORTED_LIST1, [PIVOT | SORTED_LIST2], SORTED_LIST).

split(PIVOT, [X | LIST1], [X | LIST2], LIST3) :-
      X < = PIVOT, split(PIVOT, LIST1, LIST2, LIST3).
split(PIVOT, [Y | LIST1], LIST2, [Y | LIST3]) :-
      X > PIVOT, split(PIVOT, LIST1, LIST2, LIST3).
split(_,[],[],[]).
```

Figure 3.8. Miranda and Prolog Specifications of Quick Sort

Two-Level Grammar has the combined advantages of functional programming specifications, logic programming specifications, and natural language specifications, while eliminating many of their disadvantages. Particularly notable advantages are:

1. Two-Level Grammar provides exceptional clarity of specification. A novice computer user with an elementary knowledge of mathematics can understand the semantics of the algorithm without difficulty because what the specification does is written clearly in natural language.
2. TLG is strongly typed but free of type declarations. The specification writer needn't write many type declarations as the TLG compiler will statically determine the types.
3. TLG is based upon logic but is purely functional. In logic programs, variables are bi-directional and hence have different modes in different situations, creating some confusion. The TLG compiler uses the vocabulary of function symbols to determine directionality. On the other hand, the functional and logic programming basis facilitates correctness proofs of the specification.
4. TLG specifications can be made completely deterministic and so can be executed using conventional programming techniques. TLG programs can be translated into efficient C equivalents, thereby accomplishing automatic implementation of the software from the specification.
5. The TLG specifications contain several levels of implicit parallelism which can be detected and exploited at compile-time.

CHAPTER IV

IMPLEMENTATION TECHNIQUES FOR TWO-LEVEL GRAMMAR SPECIFICATIONS

The principal techniques for Two-Level Grammar implementation that we have developed are interpretation, preprocessing into Prolog, and transformation. The main difference among all these techniques is: the interpretation technique interprets the Two-Level Grammar program directly into the result, given a TLG query; the preprocessor technique first translates TLG programs into Prolog-like intermediate code, and then uses a conventional Prolog interpreter or compiler to execute it. The transformation technique is to automatically transform TLG programs into C programs and let the C program run using a conventional C compiler. Figure 4.1 illustrates the difference among the various techniques.

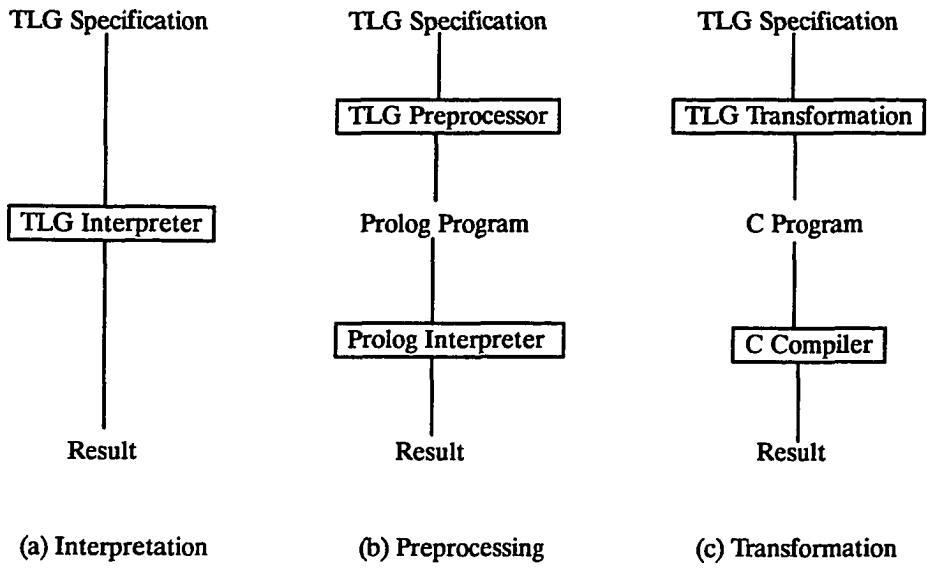

Figure 4.1. Implementation Techniques for Two-Level Grammar

However, before proceeding with the details of the different implementation techniques, we first discuss three significant extensions to the TLG language that we have developed.

1. Type declarations are allowed to be optional. This means that a TLG specification is reduced to a set of function definitions and the implementation must automatically infer the types of variables

from their usage in the function definition. We have developed such a type inference system which is unique to Two-Level Grammar.

2. In order to make Two-Level Grammar even more natural language-like, we would like to ease some restrictions in rule matching of queries to enhance robustness and fault tolerance, including the capability to recognize simple grammatical mistakes, slight variations in word order, noise words, and punctuation, as well as omission of keywords or context. This requires that we develop some techniques for natural language processing. However, because Two-Level Grammar imparts a structure to natural language, we needn't attempt a complete natural language syntax and semantics implementation. Such a system would be sufficiently complex as to negate the objectives we are trying to achieve, and may also not be within the bounds of current technology. Our approach relies on a natural language pattern matching technique we have developed called ordered keyword parsing.

3. Additional functions are added to Two-Level Grammar which allow reasoning. For example, in our query, we may ask "how" or "why" questions or request the steps of the proof procedure using any style of natural language.

In Section 4.1, it is discussed how these extensions to the TLG model may be implemented using the interpretation techniques developed by Bryant and Edupuganty ([Edup85], [Brya86a], [Brya88b], [Edup89]). Section 4.2 describes the preprocessor which translates TLG specifications into Prolog, and section 4.3 describes the transformation system which is the principal implementation technique we have developed.

4.1. Interpretation

To illustrate the capabilities which we wish to achieve, consider again the palindrome problem given in the previous chapter.

string EMPTY is a palindrome.
string LETTER is a palindrome.
string LETTER LETTERS LETTER is a palindrome : string LETTERS is a palindrome.

The first modification to the traditional TLG model is the omission of the type declarations. To implement the above program requires that the types for LETTER and LETTERS be determined. In our interpretive prototype, we perform this test at run-time, matching the rules with the respective queries dynamically, assuming that any successful match is correctly typed.

A more dramatic improvement in the Two-Level Grammar specification language is the relaxation of the way in which queries match rules. The traditional view of TLG was for each query to exactly match a rule in the TLG rule base. In Edupuganty's interpreter, this was enforced by finite

automata to match keywords and context-free parsing to match variables. Because one of the goals of our research is to make TLG a form of natural language interface between software users, designers, and implementors, as well as to facilitate rapid prototyping, we do not require that the respective parties know all the details of the interface syntax. Approximate matches will be accepted as long as each query will match a specific rule. Only the use of variables must be exact – the person writing the query must know what he is querying for. As examples of the type of flexibility we are seeking, the following queries should match the definition of palindrome given above.

1) **is string abcba a palindrome?**
2) **is eve also a palindrome?**
3) **would toot be a palindrome too?**
4) **ww palindrome?**
5) **how about level?**

In the first case, we allow the query to be stated as a natural language question which is a permutation of the original rules. In the second case, the keyword **string** is omitted. Query 3 shows a complete difference in wording from the original rule base. The fourth query is the minimum query which can still be matched; all other noise words are omitted. All of these queries may be successfully matched by our interpreter because of the distinguishing keyword, **palindrome**. Finally, we also allow queries to be based upon context. In the fifth query, even a reference to palindrome is omitted. The interpreter can detect the context of the query as pertaining to **palindrome** because all the previous queries do and there are no alternative rules in the rule base to match.

Allowing the flexibility of approximate rule matching provides the users of the system with a convenient mechanism for accessing the rules without knowing the exact syntactic details in which the rules are specified. However, it complicates the implementation considerably. A completely different method of rule matching must be employed in order for these types of queries to be answered. On the other hand, the system must be computationally efficient as well. Our solution is to use a technique for pattern matching we have developed called ordered keyword parsing. In this approach we build several levels of lexicons, ordered by the importance of the words. For example, words such as a, the, etc., would be at a lower level than verbs such as is, would, and so forth, which would be at a lower level than nouns. The structure of our ordered keyword parsing algorithm is illustrated in Figure 4.2.

Our system processes the rule base prior to interpretation and constructs the keyword table. Obviously the larger and more diversified the rule base, the larger the lexicons and the narrower the margin for approximating the keyword matching. For this reason, we say that the our system "understands" natural language at multiple levels. If the semantic domain expressed by the rule base is smaller, then the domain of syntax for queries is larger and more flexible because there are fewer

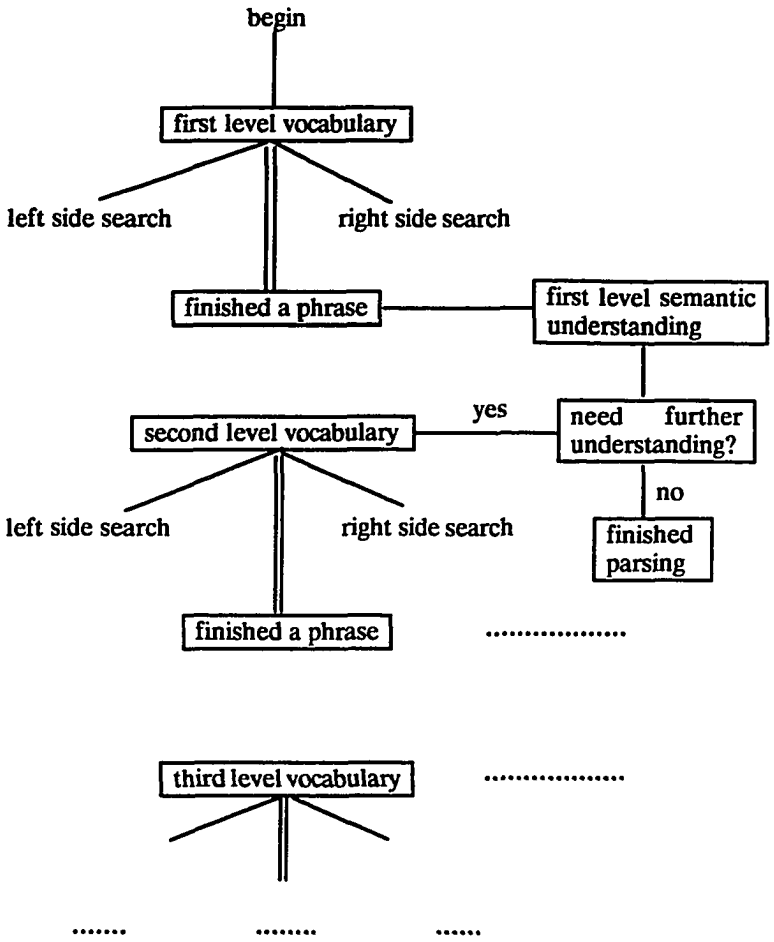

Figure 4.2. Ordered Keyword Parsing Algorithm

objects to match. On the other hand, as the semantic domain grows, the syntax domain correspondingly shrinks. The utility of this technique for processing queries to database and knowledge-base systems has also been researched.

The interpretation scheme for TLG which we have developed is shown in Figure 4.3. The input of this interpreter is a TLG program (rule base) and queries to be answered using that rule base. The TLG rules are interpreted using finite automata, context free parsing, and natural language processing techniques to match rules, applying the rules using backward chaining and unification as in Prolog. The multi-level key word parsing technique is the main technique that we use to understand the natural language part of the TLG. These three modules will be used in all the different implementation techniques of TLG.

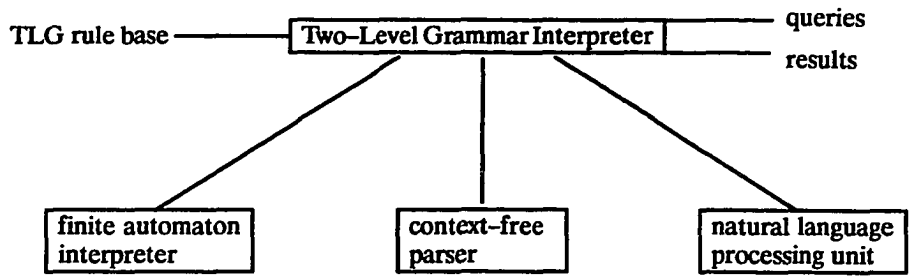

Figure 4.3. Two-Level Grammar Interpretation

4.2. Preprocessing

Another method for implementing TLG is the preprocessor technique. We translate TLG into a form of Prolog, made more efficient by automatically generating cuts where appropriate, as determined by the determinism of TLG rules. It is also possible to tag variables with their mode (input or output) and type (integer, boolean, etc.) using the mode and type inference procedures developed in the next section. Because of the nature of Prolog, these procedures take place at run-time. However, this method can be used as the front-end of the transformation system and is exactly the way in which we begin the transformation process. For this reason and the fact that other aspects of these procedures are extended to be applicable to C, more details about these components will be described in the next section. Figure 4.4 shows the basic mechanism of the preprocessor technique.

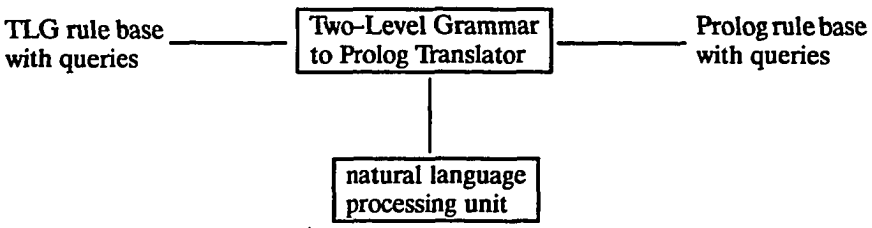

Figure 4.4. Implementation of Two-Level Grammar by Preprocessing

The Two-Level Grammar to Prolog translator includes a natural language processing unit, to translate the natural language of TLG into Prolog-like predicates. The finite automata and context-free parser of the previous section are not shown here as they play a lesser role. They are needed only for the syntactic parsing of TLG rules in the preprocessor. The semantics are now handled by the Prolog interpreter. The main approach of this method is to determine the collections of rules, identifying which rules belong to the same function. Once the keywords comprising a rule are determined, the remaining text is assumed to be the arguments. If the arguments make up regular domains, then they are grouped

into a list in the Prolog representation. If the arguments are part of a context-free domain, they must be represented by Prolog structures. The type inference algorithm we have developed determines this information.

As an illustration of this preprocessor method, consider the traditional example of the palindrome. The corresponding Prolog program is shown below:

```
palindrome(input, nil, output, true) :- !.
palindrome(input, LETTER, output, true) :- character(LETTER), !.
palindrome(input, string(LETTER, LETTERS, LETTER), output, BOOLEAN) :-
      !, character(LETTER), string(LETTERS),
      palindrome(input, LETTERS, output, BOOLEAN).
palindrome(input, ANYTHING_ELSE, output, false).
```

This program can run in Prolog directly, assuming the existence of the character and string type checking predicates. Note that this program is semantically more equivalent to the TLG palindrome function than the Prolog example given in Chapter 3. However, it is perhaps even worse as a specification because it requires considerable implementation detail to be coded in the predicates. Fortunately our preprocessor can generate this automatically.

It is also the case that different TLG's might be translated into the same Prolog target programs. As an example of this, consider two different TLG's to describe paternal relationships, adapted from [Chan73]. Program P1 is:

```
john is mary's father.
Z is Y's grandfather : X is Y's father, Z is X's father.
```

The same program can also written in a different way, as shown in program P2 below:

```
the father of mary is john.
the grandfather of Y is Z : the father of Y is X, the father of X is Z.
```

However, both TLG's may be translated into the Prolog program:

```
father_of(input, mary, output, john).
grandfather_of(input, Y, output, Z) :-
      father_of(input, Y, output, X), father_of(input, X, output, Z).
```

The reason these two TLG's are treated the same is that they have the same keywords, father and grandfather. All the other words are noise words. On the other hand, our natural language processing system can also detect the direction of possession (e. g., pertaining to the of and 's constructions) in order to know which variables are input variables and which are output variables. Therefore, both TLG's are translated into the same Prolog program.

Let us consider how the queries are handled by the preprocessor. For the above program, both yes/no and "who is" type questions may be answered. These are respectively referred to as Class A and Class B type questions in [Chan73]. An example of a Class A question is:

is john mary's father?

This is translated into the Prolog query:

father_of(input, mary, output, john).

which can be seen to match the first rule. The formal answer given by our interpretive system would be:

yes, john is mary's father.

which could be embedded in the Prolog prototype as well. Class B questions include "who is," "where is," or "under what condition" as an answer. The first of these is appropriate in this case. As an example, consider the query:

who is mary's father?

This is translated into the Prolog query:

father_of(input, mary, output, Who).

The TLG answer is:

john is mary's father.

For a more complicated query, consider:

who is the grandfather of X?

This has two variables and no constants in the argument list. It may be translated into Prolog as follows:

grandfather_of(input, X, output, Who).

This query can not be answered by Prolog unless we include the rule:

grandfather_of(input, X, output, father_of(input, father_of(X)).

and introduce a cut at the end of the previous grandfather rule. That is, this rule will only be matched if the other rule can not succeed. The rationale for adding this rule is the implicit relationship established by the use of the verb is in the original TLG rule. If **the father of X is Y** and **the father of Y is Z**, then **the father of the father of X is Z**, substituting for **Y**. This type of reasoning is not possible in Prolog without considering the natural language semantics of the rule. The query would then be answered in TLG by:

the father of the father of x is the grandfather of x.

4.3. Transformation

The transformation technique that we mention here is the main topic of this dissertation. According to our result, this is the most efficient technique for implementing Two-Level Grammar, compared to the other two techniques that we mentioned before. The transformation is totally automatic. The main job of the system is to transform high level TLG specifications into high performance C code and let the generated C code run under the C environment (e. g., compiled into

machine code by a C compiler). Since the transformation only takes place one time no matter how many times the resulting system is executed, even if transformation is a relatively complicated procedure, considerable work is saved at run-time, thereby producing overall performance improvement. Figure 4.5 shows the main structure of the transformation system.

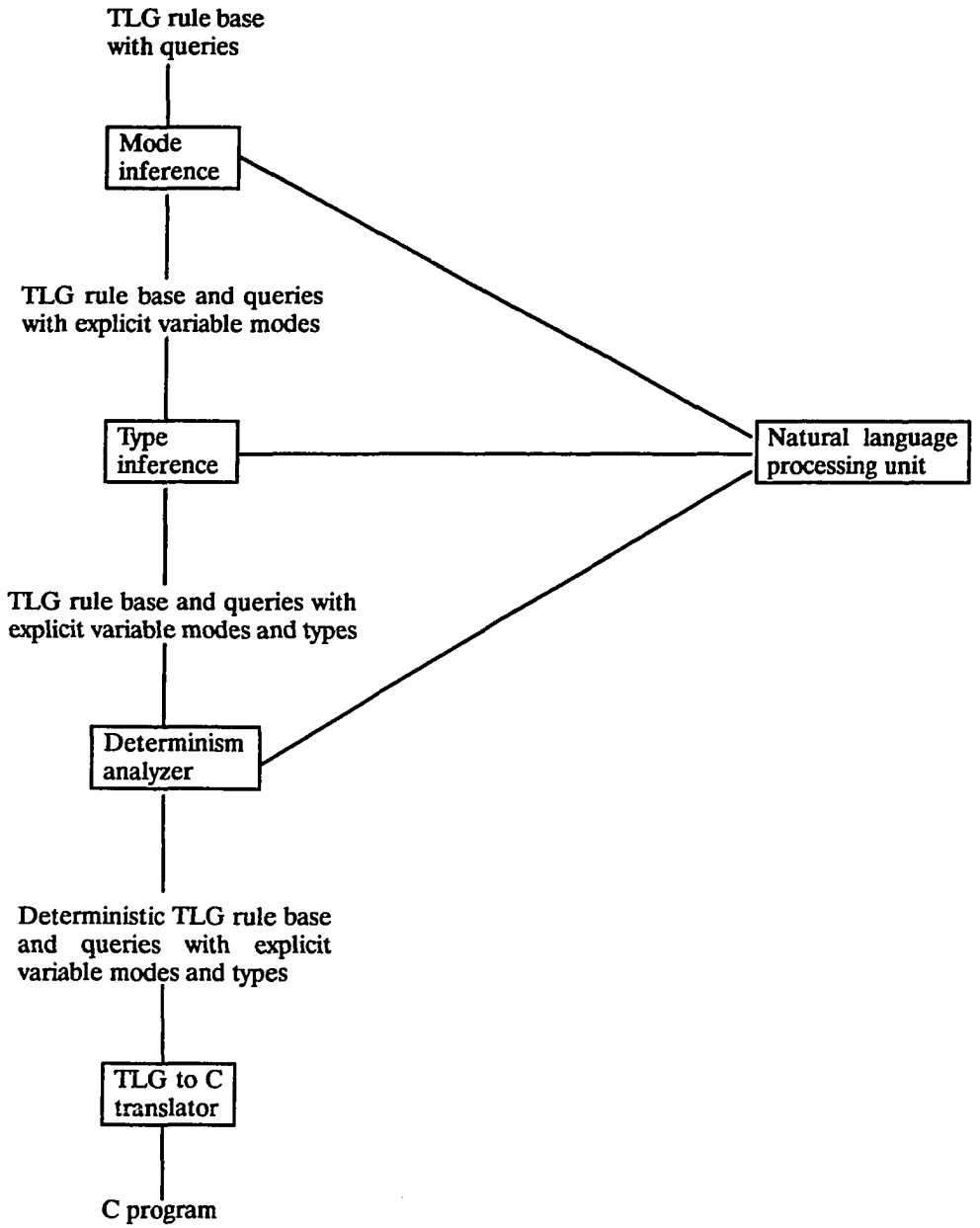

Figure 4.5. Implementation of Two-Level Grammar by Transformation

We will discuss the details of the transformation system in the next section. The main idea of this procedure is : we input the TLG rule base program (TLG specification), then we have several procedures to transform this high level logic specification into C code.

1. Mode inference. We use the natural language embedded in TLG rules to establish the directionality of the logical variables. For example, we define a set of keywords which may be reasonably associated with the notions of "input" and "output" variables, as originally suggested in [Brya88a]. We expect that this type of heuristic information can greatly improve upon traditional mode analysis techniques (e. g., see [Debr87]). In order to translate the TLG into C, we need to know which variables are to be passed as "in" parameters to C functions and which are to be passed as "out" parameters (i.e. using pointers).

2. Type inference. In order to effectively translate TLG specifications into C programs, the system of types must be well-known. As part of our enhancements in TLG, we have eliminated the explicit type declarations for programming convenience. This means that types of logical variables must be inferred from their usage. Because TLG "types" are defined by regular sets and context-free languages, we may use parsing to perform type inference. Parsing techniques offer improvements in efficiency and capability over existing type inference methods (e. g., see [Bans88]) since language recognition is computationally "easier" than unification used in other methods. However, we have also developed a method of type inference which is independent of parsing and may be applied directly to the Prolog intermediate form. Either of these techniques are sufficient for our purposes. The result of this step is a symbol table indicating the types of the various variables involved in the TLG program.

3. Determinism analysis. In the usual case, because of their functional nature, TLG specifications are deterministic. However, it is sometimes convenient to express problems nondeterministically. Given a slightly restricted form of nondeterminism, we are able to simulate it using multi-branch if-then-else statements and recursion. It should also be noted that the determinism or lack of it in TLG specifications can be determined partly by the type inference procedure which will establish the domains of the rule variables.

4. Translation into C. Once we know the modes and types of variables and choice points of nondeterminism, we generate C programs to effect the TLG specification. The primary goal of this translation will be to preserve the semantics of the TLG specification and our transformation rules are expressed in a provably correct manner. Second, we wish to take advantage of the

efficiency with which C programs can be written, including structure sharing, use of iteration instead of recursion, and replacement of dynamic data structures with static ones.

The result is a transformation system which can translate Two-Level Grammar specifications into efficient C programs.

4.4. Automatic Type Inference

We give two methods of performing type inference. One uses the context-free parsing techniques that may also be used to interpret TLG specifications. The other uses λ-ϕ-reduction to compute the types from the formal function definitions.

4.4.1. Two-Level Grammar Type Inference

Two-Level Grammar programs are normally interpreted using a combination of finite automata for matching regular set patterns and parsing for context-free patterns. These recognizers are set into motion by querying the TLG program with a function call. The main task is then to match the symbols of the function definition with the symbols of the function call to determine which rule is appropriate. For example, in the append function, if we give the query:

append 1 2 3 with 4 5 6 giving LIST

then the interpreter will fail to match **1 2 3** with **EMPTY** and instead will match it with **INTEGER INTEGER_LIST1 (INTEGER = 1, INTEGER_LIST1 = 2 3)**. Continuing the interpretation, **4 5 6** will match **INTEGER_LIST2** and **LIST** will match **INTEGER INTEGER_LIST3**. The remainder of the execution is essentially like that of Prolog, except the unification of terms is done using string pattern matching instead of unification.

In contrast with many existing type inference systems which use the query to determine the types of variables (e. g., see [Bans88]), we would like to perform a complete static analysis of the program, independent of any queries. Our method derives a grammar from the different alternatives of the function arguments. Since all invocations of our example function use the identifiers **append**, **and**, and **giving**, we can determine that the function has pattern **append APPEND1 and APPEND2 giving APPEND3.** That is, there are three variables. The way the variables are used in the function definitions then gives rise to the following initial production rules:

APPEND1 :: EMPTY; INTEGER INTEGER_LIST.
APPEND2 :: INTEGER_LIST.
APPEND3 :: INTEGER_LIST; INTEGER INTEGER_LIST.

We then proceed to unify the different function arguments. Beginning with the first function rule, we have that **APPEND3** $\equiv$ **APPEND2** $\equiv$ **INTEGER_LIST**, implying that **INTEGER_LIST** $\equiv$ **INTEGER_LIST** and **INTEGER_LIST** $\equiv$ **INTEGER INTEGER_LIST.** Treating these as regular expression equations and

solving for **INTEGER_LIST** gives **{INTEGER}***, i. e. a list of zero or more integers, as the type of both **APPEND2** and **APPEND3**. Now, **APPEND1** is equivalent to the regular expression **EMPTY + INTEGER INTEGER_LIST**, which simplifies to **{INTEGER}***. Therefore, the type of this function is **{INTEGER}*** × **{INTEGER}*** → **{INTEGER}***.

Note that the determination of types in the previous example was simplified by the facts that 1) it was known that the element type of the lists was **INTEGER**, and 2) the patterns of the variables were right-linear, meaning that we only needed to solve for regular expressions. If case (1) had not been true and we did not know the type of the list elements, then it would be necessary to either treat the function as polymorphic (e. g., treat the list elements as being of type **NOTION**) or request the user to specify the type in an explicit domain declaration. If case (2) held, we would have a context-free pattern and so would be more limited in the types of equations we could solve, since the equivalence of two context-free languages is undecidable. Consider briefly the function definitions for the palindrome problem given earlier. The single argument has patterns of the form:

PALINDROME :: EMPTY; LETTER; LETTER LETTERS LETTER.

The only function call determines that **LETTERS ≡ PALINDROME** from which we can deduce the following context-free production rule defining **LETTERS**.

LETTERS :: EMPTY; LETTER; LETTER LETTERS LETTER.

Without domain declarations to tell us that **LETTER ≡ CHARACTER**, we can determine nothing further. It is worth noting that the type defined by **LETTERS** is different from the type **STRING**, even though both are comprised of a list of **CHARACTER**'s. **LETTERS** is treated as a tree-structured domain while **STRING** is a linear domain. This assures that we never need to question the equivalence of regular sets and context-free languages.

It should be mentioned that the only previous work in this area was done by Maluszynski and Nilsson [Malu82] who showed that TLG rule-matching could be reduced to standard unification. However, their goal was to show that two-level grammars defined a class of logic programs. We have extended this result considerably by defining a specific method by which the pattern matching can take place, using both regular set patter matching and context-free parsing, and used this technique to infer the types of logic programs expressed in TLG.

4.4.2. Logical Type Inference

As mentioned earlier, our Prolog preprocessor is a front-end to the transformation system in the sense that we use Prolog as an intermediate language. Consequently, we will use Prolog syntax for data structures (e. g., [] for EMPTY, type(X) for X ∈ TYPE, where TYPE is a defined or basic type, upper

case for variables and lower case for data structures, etc.). For context-free patterns, we use a Prolog functor corresponding to the parse tree. For example, LETTER STRING LETTER is represented by string(LETTER, STRING, LETTER). Here, the functor string has no particular type connotation but serves to identify the argument to palindrome as being of that type. An example of this notation was shown earlier for palindrome. We call this form for our intermediate code TFM (Type Free Metalanguage). Because we begin with a restricted form of Prolog (although the original TLG specification has no restrictions), we are able to perform much better type inferencing than result from type inference techniques presented by [Bans88], [Zobe87], and others using abstract interpretation of standard Prolog. For example, Bansal's system is top down - the initial basic type information is given in the user's query. Our system extracts the type information from the source code by recognizing the data structure pattern and this analysis will be performed independently of user queries. Zobel's work is bottom up, getting type information by rewriting rules, which will affect the efficiency of the system.

Our type system has the following characteristics:

1. The system is one pass and highly efficient. The source code doesn't need to be rewritten, and the type inference procedure doesn't have side effects to the source code. The algorithm is completely syntax directed, and uses a uniform way to handle all data types including recursive data types, and also handles all source code structures including recursive rules. We do not need to check whether a rule is recursive or not, in contrast with the other methods which must perform this time consuming test.

2. The type inference algorithm also detects type errors, and can report the type error at exactly the position at which it occurs and the type of error. We claim that our algorithm can instantiate the type to the maximum extent possible.

3. For most source code, all the types can be inferred without human interference by referring to some built-in type declarations, e. g., X<Y has type Int < Int. However, for some programs, it is impossible to figure out all the types. This will be detected and the user is then requested to specify the undefined types.

4. We use a new concept of unification, which we call inclusion unification. This is based on the standard unification techniques but has the added characteristic of being able to unify two terms without the concept of logical variables.

5. Our algorithm only needs one scan of the source code and hence is generally O(n), where n is the length of the source code. The worst case of our algorithm is O(nm) where m is the environment length which is equivalent to the number of different functors in the source code. Although m

is a function of n, it will generally be much smaller, which is why we argue that the complexity is approximately O(n).

To perform the type inferencing, we use a new concept based on standard unification called inclusion unification. Our type inference algorithm is based on this inclusion unification. For efficiency considerations, all unifications are between a set of two elements. For clearer illustration, we give our inclusion unification algorithm by comparing with the standard unification algorithm (e. g., see [Chan73]):

Standard Unification Algorithm

Input: A pair of terms, denoted by W

Output: Unified term, M, and most general unifier σ

Step 1: Set $k = 0$, $W_k = W$, and $\sigma_k = \epsilon$

Step 2: If W_k is a singleton, $M = W_k$, halt; $\sigma = \sigma_k$ is a most general unifier for W. Otherwise, find the disagreement set D_k of W_k.

Step 3: If there exist elements v_k and t_k in D_k such that v_k is a **variable** that does not occur in t_k, go to step 4. Otherwise, halt; W is not unifiable.

Step 4: Let $\sigma_{k+1} = \sigma_k\{t_k/v_k\}$ and $W_{k+1} = W_k\{t_k/v_k\}$. (Note that $W_{k+1} = W_k\{\sigma_{k+1}\}$.)

Step 5: Set $k = k + 1$ and go to step 2.

Inclusion Unification Algorithm

Input: A pair of terms, denoted by W and a definition of the inclusion relationship on the domain D

Output: Unified term, M, and most general unifier σ

Step 1: Set $k = 0$, $W_k = W$, and $\sigma_k = \epsilon$

Step 2: If W_k is a singleton, $M = W_k$, halt; $\sigma = \sigma_k$ is a most general unifier for W. Otherwise, find the disagreement set D_k of W_k.

Step 3: If there exist elements v_k and t_k in D_k such that v_k **includes** t_k and v_k does not occur in t_k, go to step 4. Otherwise, halt; W is not unifiable (type error).

Step 4: Let $\sigma_{k+1} = \sigma_k\{t_k/v_k\}$ and $W_{k+1} = W_k\{t_k/v_k\}$. (Note that $W_{k+1} = W_k\{\sigma_{k+1}\}$.)

Step 5: Set $k = k + 1$ and go to step 2.

The difference between the standard unification and the inclusion unification is that, first an inclusion relation must be defined, and second the relationship between variable v_k and the unification result t_k is determined by an inclusion relationship, no matter whether v_k is a variable or not. This allows us to instantiate the type step by step. Finally, after scanning the whole source program, all the types should

be ground. Otherwise, the user is asked to define the non-ground types. For convenience, we denote the inclusion unification of terms v_k and t_k resulting in t_k by $(v_k,t_k)\rightarrow t_k$.

Before proceeding with the details of inclusion unification, let us give a high-level view of the difference between this and standard unification. Consider the problem of unifying a pair of terms W = {p(a, X, f(g(Y))), p(Z, f(Z), f(U))}. Using standard unification, the unifier is {a/Z, f(a)/X, g(Y)/U} and the resulting term is p(a, f(a), f(g(Y))). For inclusion unification, first set the inclusion relation as *function* $\subset$ a $\subset$ X $\subset$ Y $\subset$ Z $\subset$ U. In this case, inclusion unification gives the same result as standard unification. However, if we set the relation as U $\subset$ Z $\subset$ Y $\subset$ X $\subset$ a $\subset$ *function*, then the result is p(Z, X, f(U)), with unifier {Z/a, X/f(Z), U/g(Y)}. Therefore, it can be seen that the inclusion unification result is based on the definition of the inclusion relation.

The terms that participate in unification are type expressions. A grammar for type expressions is given below. Since our type inference program is written in Lisp, the type expression is in Lisp form. For a list of multiple types, the manipulation is similar to structures.

Type ::= Basic-type | Compound-type | $\perp$
Basic-type ::= int | string | unknown
Compound_type ::=
    [list, Type] | [struct, Type] | [$Type_1$, $Type_2$, ..., $Type_n$] | [function, Type] | [unknown, Id]

Note that we use nil as the base value for all structures. For example, when nil is met by the system, the type that is inferred is [struct, unknown], it means the parameter is of type structure, with the element of unknown type. If [] is met, then the type is [list, unknown]. We define the inclusion relationship as

$\perp \subset$ Type $\subset$ [unknown, Id] $\subset$ unknown

where Type is the rest of the type list above corresponding to a discrete partial order, although compound types have their own partial orders. For example, [list, int] $\subset$ [list, [unknown, Id]] $\subset$ [list, unknown]. The inclusion is transitive. Note that it is necessary to separate the types of unknown and [unknown, Id], and this is the key to making our implementation successful and highly efficient. [unknown, Id] denotes that while the specific type is unknown, we do know of a set of objects of that type which are unifiable and hence may be tagged with Id.

In type inference, we use an environment to record the data type. At the conclusion of the algorithm, the contents of the environment is the type specification of each argument in every predicate in the source code. The type inference algorithm is given below, divided into four cases for the program, individual rules, subgoals, and predicate arguments.

Type Inference Algorithm

1. Program type inference

   Input : TFM program Q as collection of rules R_1, R_2, ..., R_n

   Output: Type environment E for Q

   Step 1: Set E to be empty.

   Step 2: If Q is empty, halt and output E.

   Step 3: Set R to be the first rule in Q. Delete R from Q.

   Step 4: Rule type inference ($R \rightarrow E_r$)

   Step 5: Global inclusion unification ($E, E_r \rightarrow E_n$). Perform type reduction.

   Step 6: Set $E = E_n$ and go to step 2.

2. Rule type inference

   Input : Rule R of the form H :- B

   Output: Type environment E_r for R

   Step 1: Head type inference ($H \rightarrow E_h$).

   Step 2: Body type inference ($B \rightarrow E_b$).

   Step 3: Set E_r = append (E_h,E_b).

3. Subgoal type inference

   Input: Subgoal $p(A_1,A_2,...,A_n)$

   Output: Environment E_p for subgoal

   Step 1: Set P = $\{A_1,A_2,... A_n\}$.

   Step 2: Set P_1 to be the first parameter in P. Delete P_1 from P.

   Step 3: Parameter type inference (E_p, $P_1 \rightarrow E$).

   Step 4. Set $E_p = E$ and go to step 2.

4. Parameter type inference

   Input: Parameter A, current rule environment Er, current program environment Ep

   Output: Environment E_a for parameter

   Step 1: Generate type expression E_a" according to the syntax of the parameter A

   Step 2: Local inclusion unification (E_a", $E_r \rightarrow E_a$')

   Step 3: Global inclusion unification (E_a', $E_p \rightarrow E_a$)

Local inclusion unification is to unify the current type expression with the other type expressions that have the same variable in the current rule. For example, the variable X underlined in append([X|Y],Z,[X|R]), X should have local inclusion unification with the X in append.1. Global

inclusion unification unifies the current type expression with the type expression appearing in the last rule, with the same argument position in the same predicate. A central idea is that after we finish processing a rule (head and body), we need to do type reduction. The function of the type reduction is figuring out the optimal type expression and eliminating the redundant variable names. This step is very important because some recursive data types can be linearized in this step, which prevents the recursive data structures from unfolding too much during the inclusion unification procedure. The main algorithm is : if the parameter is of the same type, e. g., list, check the variable; if the variable is also the same name, e. g., X, then the new type is the tuple of the type expression and the variable's name, e. g., [list,X]. If the variables names are different, then eliminate the variable's name, and what remains is the desired type expression.

In summary, there are two situations under which we should perform inclusion unification:
1) When we have the same variable within the same rule (local inclusion unification).
2) When we have a parameter in the same position of the same functor (global inclusion unification).
Only under one condition should we perform type-reduction: whenever we finish scanning a rule.

An example of this procedure is given below. Note that to simplify the discussion, we limit list elements to be of one type only. Lists with multiple types can be handled similarly using structured or union types.

Source Code

R1: append(input, [], X, output, X).
R2: append(input, [X | Y], Z, output, [X | R]) :- append(input, Y, Z, output, R).

In the discussion which follows we use R1-append to denote the time when the type inferencing system scans the append functor of R1, R2-h-append to denote the time when the type inferencing system scans the head of R2, and R2-b-append to denote the time when the type inferencing system scans the body of R2. We use append.1 to denote the first argument to the predicate, append.2 to denote the second argument, etc. We have an environment denote by append.1 , append.2 and append.3 to keep the most updated type expression of each parameter. We use parenthesis (...) to denote the tuple of a type expression and its variable, and use square brackets [...] to denote the type expression only. The tuple (...) also can take part in the inclusion unification. Usually, after type reduction, all tuples will be reduced into type expressions only. We use *u* as short for unknown. After processing R1, we have **R1-append.1**: [list, *u*]. This means that the first parameter of append is a list, with the type unknown, and there is no variable belonging to this type in this parameter;

R1-append.2: (*u*, X). The second parameter of append is an unknown type and there is a variable named X belonging to this type;

R1-append.3: (u, X). The third parameter is an unknown type and X belongs to this type.
At this point, we observe the same variable X within the same rule, and this is one of the conditions to perform local inclusion unification. The result of the local inclusion unification of (u, u) is still u, so the type remains unchanged.

Now we have finished processing R1 and should do type reduction. The functions of type reduction are to optimize the type expression, e. g., straighten the recursive data structure from unfolded form into folded form, and to eliminate the variable because the same variable name doesn't mean anything to the next rule. If the type is unknown, with variable name X in several parameters, then these variables should all have the same type. The type reduction makes the X part of the type expression, so the new type expression is [u, X], to specify this as a special type, and denoting type equivalence among unknown types. Note that this notation of type expression is better then the system mentioned in [Bans88] and [Zobe87], where they use α, β, etc. to denote specific types (type variables). This system allows more flexibility and has more readability. After type reduction, we have:
environment:

append.1: [list, u]
append.2: [u, X] . Now [u, X] is a type, with no elements.
append.3: [u, X]

While the notation appears to remain the same, the meaning is actually quite different. At this point in the procedure each entity is a type expression and the variables have been eliminated. Before, (u, X) meant a list with the type unknown but a member of X; now [u, X] is reduced as a type expression. Next we scan a new rule R2.

R2-h-append.1: ([list, (u, X)], Y). The new type of append.1 is [list, (u, X)] because X is an element of the list and X is unknown type. Y is a variable of this type. Now it is time to perform global inclusion unification because condition (2) is encountered. After global inclusion unification, ([list,u], [list, (u, X)]) → [list, (u, X)]. so the type expression in the environment append.1 should be changed to [list, (u, X)].

R2-h-append.2: (u, Z). The second argument is an unknown type and Z is a variable of this type. After global inclusion unification, ([u, X], u) → [u, X]. The environment append.2 remained the same.

R2-h-append.3: ([list, (u, X)] , R). This situation is similar to R2-h-append.1. After global inclusion unification, the environment append.3 will be updated to [list, (u, X)] , At the same time append.2 will also be updated to [list, (u, X)] . The system now begins processing the body of R2:

R3-b-append.1: (u ,Y). ([list, (u, X)], u)→ [list, (u, X)]. append.3 remains unchanged.

R3–b–append.2: (u, Z). ([list, (u, X)], u)→ [list, (u, X)].

R3–b–append.3: (u, R). ([list, (u, X)], u)→ [list, (u, X)].

After outer level type reduction we, get rid of the variable X,Y, Z, we have:

append.1: [list, (u, X)]; append.2: [list, (u, X)]; append.3 [list, (u, X)]

After the inner level type reduction, we have

environment:

append.1: [list,[u,X]]; append.2:[list,[u,X]] append.3:[list,[u,X]]

This is the final form of the environment, but the type of the list is still unknown as type [u, X], so the system will request user assistance in clarifying the type. If the user specifies type integer, for example, then the complete environment is:

environment:

append.1: [list, int]; append.2: [list, int]; append.3: [list, int]

This can be directly translated into C type declarations using recursive data structures.

We have used this algorithm to test several more programs in this form, including a Prolog lambda calculus reduction machine [Pan89], a semantics–directed compiler for Ada [Pan90a], and a relational database system [Pan90b], and the result of the type inferencing is very accurate. It is infrequent that types have to be explicitly stated.

4.5. Generation of C Code

TFM differs from Prolog in two ways: 1) the input–output mode of a function is fixed, and 2) there is only limited backtracking, at most one level. These differences allow us to translate TFM into a static imperative language such as C.

Generally, if the nondeterminism is restricted to one level, that is, rules differ only in a set of guards that appear at the beginning of each rule body, then the nondeterministic conditional action can be translated into if–then–else statements. Our goal on code transformation is to work out an elegant, uniform, mechanical and efficient way to produce the target code. During this procedure, details of C functions and features must be considered for the mechanical transformation. The general scheme used to translate this form into C is described below:

1. Recursive data structures in TFM are translated into structures with the recursion replaced by pointers.

2. Functions in TFM are translated into C procedures with the same name and the same order of arguments, and the procedure also uses recursion as in the TFM program.

3. For each argument in output mode of the TFM program, we define a pointer for it. Any time this procedure is called, we also allocate memory to hold the value of the output parameter.
4. For data structures used in more than two procedures, we create a global static declaration.
5. Several automatic routines must also be included. Because the recursive structure in TFM is unlimited, we must have procedures to generate the input (initialize the data structure) and output (print out the data structure). The specific C routines will be based on the recursive data structure definitions.
6. Garbage collection routines also must be inserted in the appropriate place.

In the above translation scheme, we have several functions which generate the C code, each using the prefix "generate." At the top level of the program, the procedure generate_global_definitions uses the environment to produce a set of global structure definitions. If the same type is required by more than two different rules, it is defined globally. As an example, consider the append function. The structure list is used by both main and append so should be globally declared as follows:

```
struct list {int data; struct list *link;};
```

The body of the main procedure is produced by a procedure called generate_main and has the following structure:

```
generate_main
{   generate_input_variable_declarations;
    generate_output_variable_declarations;
    generate_memory_allocations;
    generate_input;
    generate_query_call;
    generate_output }
```

The procedures generate_input_variable_declarations and generate_output_variable_declarations are to create the local variable declarations for the main procedure. All the output variables should be declared as pointers, in order to allow their contents to be changed by the procedures to which they are passed. Furthermore, each output variable must have storage allocated to contain the result and each different type of structure pointed to must have its own storage allocation procedure, hence the generate_memory_allocations procedure. The input variables are then retrieved, followed by a call to the top-level query. Finally the result variables are output.

As an example of C code generation, consider the append function given earlier, which results in the following:

```
main()
{   struct list *append1, *append2;          /* generate_input_variable_declarations */
    struct list *append3;                    /* generate_output_variable_declarations */
```

```
    struct list *listalloc();                    /* generate_memory_allocations */
    inputlist (append1); inputlist (append2);  /* generate_input */
    append (append1, append2, append3 = listalloc());  /* generate_query_call */
    outputlist (append3);  }                     /* generate_output */
struct list *listalloc() {...}
inputlist (input_variable) struct list *input_variable; {...}
outputlist (output_variable) struct list *output_variable; {...}
```

Note that the listalloc, inputlist and outputlist procedures are generated at the same time as their calls in the main procedure.

Each of the predicate procedures is produced by sequencing through the statically created environment. The generate_procedure function maps each predicate onto the corresponding C procedure. The logical rules comprising the procedure will be partitioned into cases of an if-then-else conditional structure. The outline of generate_procedure is as follows:

```
generate_procedure (Proc)
{   generate_procedure_head (Head);
    generate_local_variable_declarations (Proc);
    if (generate_condition (Rule))
    {   generate_initial_assignment (Head);
        generate_body (Body);
        generate_return_assignments (Head); }
    else
    {   generate_procedure (Rule); }}
```

The parameter to generate procedure, Proc, a procedure comprised of a list of rules, is of the form rule(Head, Body, Rule). First, the head of the C procedure is produced along with declarations of the formal parameters. There will be one formal parameter for each argument to the corresponding predicate. The body of the procedure begins with a declaration of the local variables. During the generation procedure, the intermediate local variables are added dynamically. Any time we need an intermediate variable to denote a pattern, this variable must be declared at the head of the procedure. Often there will not be any additional local variables to declare after the parameters. The generate_condition procedure produces the constraint checks under which a rule will be executed. These will be derived from either instantiations in the rule head (e. g., if the input variables are ground values) or from guards in the first position of the rule body. In initial assignment, local variables which are components of a structured data type are initialized to the appropriate values. The body of the rule is then expanded and a procedure call is generated for each subgoal. Finally, the output variables of the predicate currently being processed are assigned the appropriate values, if this has not already been done through calls to the subgoal procedures. The entire process is repeated for each successive rule belonging to the same predicate, generating an if-then-else structure to process all the rules. The final

else clause is only executed in the case of failure to match any rule of the predicate in which case an error will be indicated. This will not be generated if the rules cover the entire domain of input values.

As an example of procedure generation, consider again the append function.

```
append (append1, append2, append3)     /* generate_procedure_head */
struct list *append1, *append2, *append3;
{     int X; struct list *listalloc(), *Y, *R;      /* generate_local_variable_declarations */
      if (append1 == NULL)              /* generate_condition */
      {                                 /* generate_initial_assignment (none - simple data type) */
                                        /* generate_body (none) */
            *append3 = *append2; }      /* generate_return_assignments */
      else
      {     X = append1 -> data;        /* generate_initial assignment (append([X|Y],Z,[X|R])) */
            Y = append1 -> link;
            append (Y, append2, R=listalloc());     /* generate_body */
            append3 -> data = X;                    /* generate_return_assignments */
            append3 -> link = R }}}
```

In the append predicate, the rules cover the entire domain of lists, so no final failure checking is needed after processing the second rule. For the second rule, we then need only assign the local variables their appropriate values (from components of the list). Note that the copy statements can be eliminated through further optimization. Through this example, we can see that the code generation is very mechanical and straightforward.

4.6. Evaluation

The generation of C programs from TFM specifications is a transformation of logic programs into imperative programs. In Section 4.5, we studied the transformation from TFM without backtracking into C. Since TFM is a subset of Prolog, the non-backtracking TFM can be simulated in Prolog by adding cuts using the following rules:

1) If the rule is a goal only, then a clause containing only the cut operator is added.

2) If the rule is a goal with some clauses on the right hand side, then add the cut operator after every clause.

This is not the same as a Prolog program with the maximum number of cuts. If that were the case, the Prolog program would not have any backtracking at all, and the if-then-else constructions will not work properly. In this sense, we have already implemented some backtracking facility in Prolog by using if-then-else. It is not difficult to implement this backtracking in C using "multi-level recursion," where the branch points are embedded in a series of nested if-then-else constructions.

Although full backtracking can be realized in the transformation system, we prefer not to do so. Our transformation system is intended to improve the execution efficiency of high-level specifications.

Backtracking would be a detriment to this task. Furthermore, most practical programs never need to use nondeterministic backtracking. Even if we are programming a nondeterministic algorithm, the deterministic implementation of that algorithm is adequate for most purposes. Otherwise, the parallel implementation of nondeterminism, supported by the Two-Level Grammar interpreter, can substitute for backtracking.

The C code that is generated using the syntax-directed method of Section 4.5 can sometimes be optimized. The generate_procedure function in particular can be optimized for both generation of the initial assignments and generation of output assignments. Sometimes assignments will be duplicated in different rules of the same procedure. This will happen when the head of the rule uses the same parameters. For example, in the binary tree sort function:

```
insert_2tree(X, 2tree(Left1, Data, Right), 2tree(Left2, Data, Right)) :-
        X < Data, insert_2tree(X, Left1, Left2).
insert_2tree(X, 2tree(Left, Data, Right1), 2tree(Left, Data, Right2)) :-
        X > = Data, insert_2tree(X, Right1, Right2).
```

the assignment statements

```
L = insert_2tree2 -> left;
D = insert_2tree2 -> data;
R = insert_2tree2 -> right;
```

will be duplicated for these two rules. This can be optimized by making a comparison between them. The intermediate variable assignments and memory allocation can also be optimized.

The final aspect that we discuss is parallel code generation. This requires that we perform 1) data dependency analysis to determine the producer/consumer relationships and 2) annotation of the program for parallel control. We illustrate this through an example. Consider the TLG below for computing the Fibonacci function.

```
fibonacci of 0 is 1.
fibonacci of 1 is 1.
fibonacci of N is R :
        N1 is N - 1,
        N2 is N - 2,
        fibonacci of N1 is R1,
        fibonacci of N2 is R2,
        R is R1 + R2.
```

From this TLG, we can generate C code executable on a dynamic parallel system such as the Sequent Balance 21000 [Oste86]. A pseudo-code parallel C program is shown below:

```
fib(N,R)
int N, *R;
{int N1,N2,*R1,*R2
fork: if(N==0) {*R=1; #}
fork: else if (N==1) {*R=1; #}
```

```
fork: else {N1=N-1; N2=N-2;
      fork 1: fib(N1,R1=intalloc()); fork 2: fib(N2,R2=intalloc());
      *R=*R1+*R2; #}
```

This is an extension of the ideas for parallel execution of Two-Level Grammar given in the previous chapter.

CHAPTER V

COMPLETE COMPILER SPECIFICATION USING TWO-LEVEL GRAMMAR

Edupuganty [Edup87] showed that Two-Level Grammar could be used as a meta-language for the specification of operational semantics, axiomatic semantics, and denotational semantics. From his operational semantics definition, interpretation of the TLG with some input source program will give the result of executing that program, thereby accomplishing automatic interpreter generation. Interpreting the axiomatic definition with a source program, a precondition and a postcondition for that program, describing the characteristics of its input and output, respectively, gives a correctness proof that the program does give the desired output for any valid input. On the other hand, Edupuganty's denotational semantics are quite operational in nature. Instead of defining denotational semantics of language constructs to map into a mathematic denotation, his TLG can only apply an internal denotation to an input program to derive the program's output.

Using a Two-Level Grammar representation of lambda calculus in denotational semantics, we can more define the denotational semantics of programming languages more completely using TLG. We may then compile input programs directly into denotational machine code (lambda calculus) and interpret this code by a lambda reduction machine, also defined using Two-Level Grammar. To demonstrate the effectiveness of our methods, we have developed a TLG definition of a simple imperative subject language using a compiler-oriented approach, which translates programs into lambda calculus, essentially embedding a functional style into TLG predicates. It is found that there is a very natural representation of the denotational semantics domains and evaluation functions in Two-Level Grammar. Furthermore, the implicit parallelism present in a denotational semantics definition can be realized in a parallel implementation of the TLG specifications. In particular, the compilation of source programs can be executed in parallel. Our result is that TLG is a very suitable metalanguage for denotational semantics.

In this chapter, we discuss our method for defining denotational semantics using TLG and how the resulting TLG lambda code is reduced, thereby establishing the complete implementation of a semantics-directed compiler. Much of this work has been described in [Brya89], [Pan89] and [Pan90a].

5.1. Denotational Semantics

Formal specification of programming languages addresses both syntax and semantics. Syntactic specification in Backus-Naur form (BNF) [Back60] are well known and widely used by both language users and language implementors. It is even possible to automatically generate a syntactic parser from a BNF specification. However, semantic specification lags somewhat behind. Most language manuals still use natural language to describe both static and dynamic semantics. The three most common formal semantics specification approaches are 1) operation semantics, where a translation is given from the high-level language into abstract machine code which may be directly interpreted, 2) axiomatic semantics, which define the effect of executing statements as transformation on logical predicates describing the current state of computation, and 3) denotational semantics, which define a formal mapping of programs into mathematical functions whose semantics are well understood. Of these, denotational semantics [Schm86] receive the most attention in the research community because of their mathematical properties which enable implementations to be symbolically derived and the fact that both static and dynamic semantics can be completely specified within a single definition.

Denotational semantics have historically been specified using a functional programming style. However, the combined functional and logic programming paradigm of Two-Level Grammar offers a number of advantages, namely:

1) there is a very natural representation of the denotational semantics domains and the domain constructors using Two-Level Grammar functors,

2) TLG definitions are very readable and elegant,

3) TLG can be used to define both syntax and semantics in a single unified definition, including interactions with other semantic specification styles, such as axiomatic and operational specifications [Edup87],

4) Two-Level Grammar definitions are directly implementable, as we have shown, making automatic compiler generation and semantics-directed interpretation feasible, and

5) the implementation can be made to run in parallel using many well-known techniques for AND-parallelism and OR-parallelism ([Cone87], [Wise86]), allowing both parallel compilation and interpretation.

A denotational semantics specification consists of three main components:

1) *abstract syntax*, which defines the construction of *syntax domains*,

2) *semantic domains* and their associated operations, and

3) *semantic equations* which map syntax domains into operations on semantic domains, effectively translating the syntactic representation of programs into their semantic representation.

The function notation used in denotational semantics is lambda calculus. Denotational semantics functions are strongly typed, in the sense of modern functional languages such as ML [Miln90]. Types may be scalar (e. g., integer), or structured as *product domains, sum domains,* or *function domains.* In Chapter 3, we have seen how these types are represented in Two–Level Grammar.

The denotational semantics of a program are typically defined to map program constructs onto functions which operate on the machine state, or configuration. For example, at a high–level, a program may be viewed as a function which accepts an input file and produces an output file. A statement operates on the input file, output file, and memory store, usually modifying one or more of these components as directed by its semantics. An expression uses the store to look up the values of variables and returns a single value resulting from the evaluation of that expression. If we consider static semantics as well, then an environment will be added to the semantic rules for these program constructs. As a simplified example of denotational semantics, consider the rules defining assignment, statement composition, and the while loop shown below, adapted from [Paga81]. In each case, γ represents a machine configuration consisting of a store σ, input file ϕ_1, and output file ϕ_2.

1) $S[\![\mathbf{V:=E}]\!] = \lambda\gamma.\mathit{let}\ \sigma = \gamma\downarrow 1\ \mathit{in}\ <\mathit{update\_store}\ \mathbf{V}\ E[\![\mathbf{E}]\!]\sigma\ \sigma, \gamma\downarrow 2, \gamma\downarrow 3>$. An assignment statement will cause the expression E to be evaluated and stored into the associated memory location for variable V. Formally, we extract the store component of the machine configuration and return a new configuration whose input and output components remain unchanged, but whose store is updated to reflect the assignment to variable V. Note that the expression E is evaluated by function *E* and this evaluation also requires the store σ.

2) $S[\![\mathbf{S_1;S_2}]\!] = \lambda\gamma.\mathit{let}\ \gamma' = S[\![\mathbf{S_1}]\!]\gamma\ \mathit{in}\ S[\![\mathbf{S_2}]\!]\gamma'$. This function begins with some machine configuration γ and produces a machine configuration which reflects the results of executing statements S_1 and S_2. This requires that we first execute S_1 on the input configuration γ to produce a new configuration γ'. This will then be the configuration used to execute S_2, which in turn produces the new configuration, not explicitly shown in the rule, which is the return value of the function.

3) $S[\![\mathbf{while\ C\ loop\ S\ end}]\!] = \mathit{fix}(\lambda f.\lambda\gamma.(C[\![\mathbf{C}]\!](\gamma\downarrow 1) \rightarrow f(S[\![\mathbf{S}]\!]\gamma), \gamma))$. The semantics of the while loop require that we first evaluate the condition C using the function *C* and the store (the first component of configuration γ). If this is true, we execute statement S and recursively invoke the while loop again. If the condition is false, we return the input configuration unchanged. The

equation form *fix*($\lambda f....f...$) indicates that the function is to be called recursively. More details about this will be given later.

These definitions are typical of denotational semantics for these language constructs.

Current research is focussed on automatically generating compilers and interpreters given a denotational specification of the programming language. The approaches can be classified as the *compile-evaluate* and *interpret-evaluate* methods. The denotational semantics of a programming language may be viewed as an abstract interpreter for that language. For example, if the function **P**: Program → Input-file → Output-file, typical of most denotational definitions, is applied simultaneously to a program and an input file, the result will be the output file which results from interpreting the program on the input file. This is interpret evaluation. In compile-evaluation, the function **P** is applied only to the program resulting in a new function **IO**: Input-file → Output-file. In this sense, **P** can be viewed as an abstract compiler of the program into "denotational machine code." The compile-evaluate method is more popular since the techniques involved in compiling functional languages are applicable and the implementation is more efficient. The abstract machine is usually a stack machine. Such implementations use a metalanguage of their own (which is not a desirable situation) since the denotational definitions input to compile-evaluation have to be machine readable. The interpret-evaluate approach usually emphasizes readability and direct implementability of the definition over efficiency. It is our proposal that by using Two-Level Grammar, it is convenient to give two different definitions, each being more suitable to a different evaluation method. We claim that giving both a compiler-oriented and an interpreter-oriented definition serves the purpose of consistent and complementary definitions [Hoar74] for completeness.

5.2. Denotational Semantics Using Two-Level Grammar

The use of Two-Level Grammar for denotational definitions is quite natural. The meta-variables corresponding to the syntactic and semantic domains of the standard denotational definition are represented by TLG functors. The proper elements of a domain and constructed compound domains are defined by additional functors and atomic elements. The compound domains of a denotational specification are represented as discussed in Chapter 3. The semantic clauses (meaning functions) are represented by the Two-Level Grammar rules. Depending on whether our definition is interpreter-oriented or compiler-oriented, the rules take a different style. We now discuss the two different methods in detail.

5.2.1. Interpreter-Oriented Denotational Semantics

Our method of writing interpreter-oriented definitions is to encode the denotational definition directly as Two-Level Grammar rules. For simplicity, we assume that the syntactic arguments to the semantic equations are a Two-Level Grammar representation of the syntax tree. Every semantic equation will be fully parameterized so the general transformation is:

F[X] = λα.E → F(X,α,Y) :- *evaluate* E *as* Y.

Since we assume that all necessary input values will be provided to the interpreter, the rules can be executed in a straightforward call by value order, the normal order in Two-Level Grammar.

As an example of interpreter-oriented denotational semantics, consider the formal denotational semantics of the assignment statement, statement composition, and while loop shown previously. By way of comparison, the interpretive semantics are defined in Two-Level Grammar as:

statement ID := EXP maps store STORE1 input file INPUT_FILE output file OUTPUT_FILE
into store STORE2 input file INPUT_FILE output file OUTPUT_FILE :
expression EXP maps STORE1 into VALUE,
update store STORE1 with ID bound to VALUE giving STORE2.

statement STATEMENT1 ; STATEMENT2 maps CONFIGURATION1 into CONFIGURATION2 :
statement STATEMENT1 maps CONFIGURATION1 into CONFIGURATION3,
statement STATEMENT2 maps CONFIGURATION3 into CONFIGURATION2.

statement while COMP loop STATEMENT end loop maps CONFIGURATION1 into CONFIGURATION2 :
select store from CONFIGURATION1 giving STORE,
comparison COMP maps STORE into true,
statement STATEMENT maps CONFIGURATION1 into CONFIGURATION3,
statement while COMP loop STATEMENT end loop maps CONFIGURATION3
into CONFIGURATION2.

statement while COMP loop STATEMENT end loop maps CONFIGURATION into CONFIGURATION :
select store from CONFIGURATION giving STORE,
comparison COMP maps STORE into false.

It can be seen in these rules that intermediate values of configurations are represented as Two-Level Grammar variables and the final results are returned at the top level. The assignment and composition rules are almost identical to the formal notation. The while loop rule has factored out the fix-point combinator into the inductive (recursive) and basis (termination) steps of a recursive function.

5.2.2. Compiler-Oriented Denotational Definition

The goal of a compiler-oriented definition is to translate the concrete syntax (i.e. the textual form) of the source program into denotational machine code. The principal functions we are concerned with and their Two-Level Grammar representations are:

1) Lambda abstraction - λx . E. A function is defined with formal parameter x and body E. This is converted into the TLG data structure **fun x = > E.**

2) Function application – f x. The function f (usually a lambda abstraction) is applied to argument x. This operation is represented in Two–Level Grammar as **apply f to x.**

3) Let expression – *let* x = E_1 *in* E_2 . The expression E_1 is substituted for all free occurrences of x in E_2. This is equivalent to the composition of abstraction and application, i. e. $(\lambda x.E_2)(E_1)$. In TLG, this is represented without change by **let x = E1 in E2.**

4) Conditional expression – $E_1 \rightarrow E_2, E_3$. E_1 is evaluated to a Boolean result. If it is true then the value of the conditional expression is E_2; if it is false, then the returned value is E_3. The Two–Level Grammar representation of this operation is **if E_1 then E_2 else E_3.**

5) Fix–point combinator – *fix* F. F is a functional of the form $\lambda f.g$, where *f* is a function appearing in the body of *g* such that *f* will be applied when *g* is applied to some argument. The application of the fix–point combinator simulates a recursive application of the *g* function. Formally, the fix–point combinator is defined as: *fix* F = F(*fix* F). It is represented in TLG as **fix F.**

6) Built–in functions. There are also a number of standard arithmetic and relational functions which are defined in the denotational machine code. These have the usual semantic interpretations and are defined straightforwardly in Two–Level Grammar.

Compare the following compiler–oriented denotational definitions in Two–Level Grammar with the standard and Two–Level Grammar interpreter–oriented definitions given previously.

```
statement ID := EXP maps to
        fun configuration = >
            let store = select 1st component from configuration in
                store update store with ID bound to apply EXP_DENOTATION to store
                input file select 2nd component from configuration
                output file select 3rd component from configuration :
    expression EXP maps to EXP_DENOTATION.

statement STATEMENT1 ; STATEMENT2 maps to
        fun configuration1 = >
            let configuration2 = apply STATEMENT_DENOTATION1 to configuration1 in
                apply STATEMENT_DENOTATION2 to configuration2 :
    statement STATEMENT1 maps to STATEMENT_DENOTATION1,
    statement STATEMENT2 maps to STATEMENT_DENOTATION2.

statement while COMP loop STATEMENT end loop maps to
        fix (fun f = >
            fun configuration = >
                if apply COMP_DENOTATION to select 1st component of configuration then
                    f (apply STATEMENT_DENOTATION to configuration)
                else
                    configuration) :
    comparison COMP maps to COMP_DENOTATION,
    statement STATEMENT maps to STATEMENT_DENOTATION.
```

Here it is seen that the Two-Level Grammar rules return the lambda code expressed in Two-Level Grammar form for the defined construct instead of actual values (i.e. machine configurations), again illustrating the difference between the two approaches. We have defined a complete programming language using this approach and found that the TLG definition of denotational semantics compares very favorably with a standard denotational definition with respect to both readability and facilitation of the compiler generation process. Our transformation system can convert this representation directly into a compiler for the specified programming language which emits denotational machine code in TLG form.

5.3. A Two-Level Grammar Lambda Machine

The denotational machine code produced from our compiler-oriented definition is not executable in Two-Level Grammar directly. However, it is possible to define a lambda calculus machine in TLG which can execute all of the denotational expressions generated by the compiler. This definition is essentially a formal definition of the semantics of the lambda calculus using predicate logic, i.e. Two-Level Grammar. Consider the formal semantics of the six components of denotational machine code described in the previous section.

1) Lambda abstraction - **fun x = > E.** This is not executable until it is applied to an argument expression.

2) Function application - **apply f to x.** There are three rules to interpret function application in our machine.

   a) **apply fun X = > E2 to E1 giving Result : let X = E1 in E2 giving Result.**

   In the case of simple application of a lambda abstraction $\lambda X.E_2$ to an argument expression E_1, the semantics are to substitute all free occurrences of X in E_2 by E_1. This is accomplished by the let predicate which executes a let expression in the standard way (see below).

   b) **apply fix F to E giving Result : apply (apply F to fix F) to E giving Result.**

   To apply the fix-point function *fix*(F) to the expression E, *fix* should first be applied to its argument functional F and then the resulting function should be applied to E.

   c) **apply (apply F to G) to E giving Result :**

   apply F to G giving F_compose_G, apply F_compose_G to E giving Result.

   This rule is a follow-up to the previous operation. F represents the functional to which *fix* is being applied and G is *fix* (F). G should be substituted *unevaluated* for all free occurrences of the function argument to F (e. g., *f* in $\lambda f...$). This gives a new recursive invocation of F (**F_compose_G**) which can then be applied to E to get the result. Note that this rule embodies

a call by name computation order. If G is evaluated before substituting into F, the result will be an infinite loop.

3) Let expression – **let x = E1 in E2.** Because this rule is used by apply, it is the central rule in our semantics of the lambda calculus. It traverses the structure of E_2 in order to substitute E_1 for every free occurrence of x. Some examples of the rules which evaluate let are:

let X = E1 in let X = E3 in E4 giving RESULT :
let X = E1 in E3 giving E5, let X = E5 in E4 giving RESULT.

let X = E1 in let Y = E3 in E4 giving RESULT :
where X != Y,
let X = E1 in E3 giving E5,
let X = E1 in E4 giving E6,
let Y = E5 in E6 giving RESULT.

let X = E1 in (apply F1 to F3) giving RESULT :
let X = E1 in E3 giving E4, let X = E1 in F1 giving F2, apply F2 to E4 giving RESULT.

These rules correspond to the following:

a) *let* x = E_1 *in let* x = E_3 *in* E_4. The variable x is bound to expression E_1 in expression E_3 which then binds to the second instance of variable x in E_4.

b) *let* x = E_1 *in let* y = E_3 *in* E_4. In this case, variable x has scope in both E_3 and E_4 while variable y has scope in E_4.. The instantiation of the variables within these scopes is accomplished by the three let rules.

c) *let* x = E_1 *in* F_1 E_3. This is a function application. Variable x has scope in both F_1 and E_3. The instantiation is done, followed by the application.

4) Conditional expression – **if E1 then E2 else E3.** This rule is implemented by adding another argument to contain the result. It is straightforward to evaluate E_1 and then return the appropriate expression.

5) Fix-point combinator – **fix F.** The details of the implementation of this have already been described in the discussion of the apply rule.

6) Built-in functions. Many of the built-in functions in our lambda machine also have equivalent built-in Two-Level Grammar arithmetic and relational operators. To handle these, it is only necessary to generate the appropriate calls to the Two-Level Grammar operators.

Besides the combinator-reduction machine described above, we have also defined reduction machines for the denotational machine code using a variety of evaluation strategies – strict evaluation, fully lazy evaluation using graph reduction, and compilation into super-combinators. Of these, the strict evaluator is naturally the most efficient, but it can not handle the fix-point combinator or more general

functional constructions. For these, the fully lazy evaluator works very well and the graph reduction is implemented using logic programming techniques which fully utilize the features of unification nondeterminism. Compilation into super-combinators allows the denotational machine code to be compiled into Two-Level Grammar rules, thereby achieving a Two-Level Grammar program equivalent to the original source program. More details about the general implementation techniques may be found in [Peyt87]. Our specific techniques for implementing these using Two-Level Grammar are described in the next section.

5.4. Two-Level Grammar Implementation of Functional Programs

The capability of Two-Level Grammar for defining a lambda machine suggests that TLG may be used as an implementation language for functional programming languages. In this section, we explore this point further, formally specifying lambda reduction machines using the evaluation strategies of strict and lazy evaluation, fully lazy evaluation using graph reduction, and compilation into supercombinators are all defined in Two-Level Grammar using techniques unique to logic programming. It is also shown how TLG may be used to efficiently implement polymorphic type checking.

Using logic programming techniques to implement functional languages is itself an interesting topic for research. Conventional logic programming languages do not directly support either higher-order functions or lazy evaluation. Instead, their distinguishing characteristics are unification, bi-directionality of "arguments," and nondeterminism. In Chapter 3, we proposed that higher-order functions and lazy evaluation could be realized using the logic programming characteristics of Two-Level Grammar. We find that the inherent logic programming nature of TLG offers some advantages in the incorporation of these functional constructs. In order to make these constructs executable we will study the use of Two-Level Grammar to define reduction machines for λ-calculus using a variety of evaluation strategies - strict and lazy evaluation, fully lazy evaluation using graph reduction, a combination of strict and lazy evaluation using a fixed set of lazy combinators, and compilation into super-combinators. It will be shown that unification and nondeterminism are especially suitable as powerful mechanisms to increase both the elegance and the efficiency of TLG implementation of λ-calculus. Currently, we have developed prototype TLG implementations of all of the aforementioned λ-machines, taking as much advantage as possible of TLG features which may be efficiently interpreted by allowing structure sharing, dynamic rule generation, and direct access to predicates in the rule database. These prototypes have been translated into both Prolog and C using the techniques of Chapter 4. While these virtual lambda machines currently run on a sequential machine, they have been designed in such a way as to maximize parallelism. In addition to accomplishing the direct task of incorporating the desired features into Two-Level Grammar, our work also provides

new insight into the implementation of functional programming languages and the relationship between functional and logic programming languages and how they may interact with one another.

The sections which follow will respectively introduce the necessary preliminaries about λ–calculus reduction in Two–Level Grammar, including Two–Level Grammar definitions of strict and lazy evaluation, fully lazy evaluation using graph reduction, strict evaluation with a fixed set of combinators and super–combinators, some of which may be evaluated lazily, and the use of logic programming to realize polymorphic type inference.

5.4.1. Strict and Lazy Evaluation

The pure λ–calculus is the theoretical foundation of any functional programming language. Therefore, our study on implementation of functional programming languages can be simplified to the implementation of λ–calculus. We propose that the semantics of pure λ–calculus can be straightforwardly expressed in Two–Level Grammar, whether by strict or lazy evaluation.

First, consider strict evaluation of β–reduction ($(\lambda x.E)\ M \rightarrow E\ [M\ /\ x]$), where M is reduced prior to substitution for x. The operational semantics of the reduction rules are expressed by the following TLG rules straightforwardly:

```
strict beta reduce apply fun X = > E to M with ENVIRONMENT1 giving RESULT :
        strict beta reduce M with ENVIRONMENT1 giving M_VALUE,
        update environment ENVIRONMENT1 with X bound to M_VALUE giving ENVIRONMENT2,
        strict beta reduce E with ENVIRONMENT2 giving RESULT.
```

Each strict beta reduce rule has three components: 1) the expression to be reduced, 2) the environment containing the bindings of variables to values, and 3) the result of the reduction. The environment can take several different forms but is generally a last–in first–out list of free variable–value associations, where the values may be constants or λ–abstractions. It is constructed by the update function which takes an environment ENVIRONMENT1, a variable X, and the value to be bound to X (M_VALUE), and returns a new environment, ENVIRONMENT2, with the updated binding. Completing the set of rules makes the above definition executable; it is an extremely concise interpreter for strict reduction.

Normal order β–reduction may also be defined using Two–Level Grammar. Consider the definition below, adapted from [Peyt87].

```
lazy beta reduce apply fun X = > E to M giving RESULT :
        substitute X with M in E giving RESULT2, lazy beta reduce RESULT2 giving RESULT1.
```

The lazy beta reduce is the top–level rule, being used to reduce its first argument into a value which is returned by the second argument. Note that there is no rule for evaluating free variables, since they cannot be the result of a reduction. The rules for the substitute function are not shown but may be developed straightforwardly. It is also possible to combine strict and lazy evaluation together. This can

allow us to get the high efficiency from the strict evaluation and realize lazy combinators such as the fix point combinator.

The above definition is totally executable, accomplishing normal order reduction. The simplest implementation strategy for substitution in lazy evaluation uses tree reduction, which has the potential for combinatorial explosion of both time and space. This can be overcome by the graph reduction strategy discussed in the next section.

5.4.2. Fully Lazy Evaluation Using Graph Reduction

Fully lazy evaluation guarantees that the argument expression is evaluated at most one time and there are no duplicate versions of an expression introduced by reduction. Also, the expression is evaluated only when it is needed to complete the reduction of the outermost redex. This not only improves the time and space efficiency but also allows us to deal with infinite data structures or undefined items. Our implementation strategy is to use pointers instead of substitutions of the arguments. For Two–Level Grammar, we use two special techniques to realize this strategy: 1) simulate pointers using an association list indexed by variable names, which is actually an environment similar to that used in SECD machines [Land64], and 2) use Two–Level Grammar unification to perform both local and global substitutions of the arguments with the values to which they are to be bound.

5.4.2.1. Association List

We have two techniques to realize the association list. The simplest one is to represent identifier–value pairs in a list, indexed by the variable's name. Initially these values will be unevaluated expressions. However, when the value of an identifier is first needed, the corresponding expression will be evaluated and the list will be updated by a reduced value. If the same identifier's value is needed again, the reduced one will be used, so that the same evaluation procedure will be avoided. We can improve the efficiency of this procedure by using the global unification technique mentioned in the next section. If we handle the list as a stack, we can solve the name capture problem. The other technique is to generate a TLG rule for each identifier–value pair. The association list here is the set of rules generated dynamically. The advantage of this method is that the accessing time of the association list is constant when we execute our system in parallel. The price that we pay for our parallel mechanism is the need to substitute each identifier with a location counter in order to distinguish the variables in different scopes, especially variables with the same name. However, this becomes worthwhile after we have the local unification technique which is discussed in the following, because the substitution can be done very efficiently by using TLG's unification characteristic. Furthermore, the location counter mechanism solves the name capture problem automatically.

5.4.2.2. Unification

For improving the efficiency of the implementation mentioned above, we introduce two unification techniques: local unification which "unifies" the Two–Level Grammar rule's local variable, and global unification which "unifies" the parent tree of the TLG variable. The following example shows how local unification works. If we have a λ–expression E1 = (λx.(λy.((λx.x + y)1) + x)x)2 and we want to change the outer scope x into z, namely by doing an α–conversion, we can use local unification. We have a Two–Level Grammar predicate called locally unify which has the function:

locally unify x and z in E1 giving E2 ⇒ E2 = (λz.(λy.((λx.x + y)1) + z)z)2

This rule works efficiently because it uses Two–Level Grammar's natural unification characteristic, so that all three occurrences of x are replaced by z in parallel.

Global unification is used for expressions that are not "visible" to the predicate at the moment it is being unified. For example, during the lazy graph reduction of expression E2, the evaluator will not find out z is needed until it traverses the subtree of ((λx.x + y)1) + z. At this point, we search the association list to get the value of z; as soon as the system finds out the value is 2, it recognizes immediately that this is a normal form and will not take any further reduction. For efficiency considerations, it is better to bind this 2 to all other z's of E2 at the same time if they are in the same scope. However, at this point in the reduction, the other z is not in the subtree and hence cannot be instantiated using the local unification technique mentioned above. To solve this problem, we can take advantage of TLG unification and synthesize this 2 to its parent tree. The idea is very similar to the evaluation of attribute grammar. By using this technique, we can distribute the value obtained at any leaf to all the nodes of the tree.

Global unification is a crucial mechanism in our implementation of fully lazy graph reduction. It guarantees that for each identifier, the system searches the association list only when it is needed and searches at most one time. Once the identifier is found, the system will not update its value until a normal form is obtained. This will also reduce the work of creating intermediate values in the association list.

5.4.2.3. Realizing Higher Order Functions in Two–Level Grammar

A higher order function is different from a first order function in the sense of argument domain. Higher order function arguments can themselves be functions including other higher order functions. Using our fully lazy graph reduction technique mentioned above, we build up the association list by binding the identifiers to their values. In the case of a higher order function, the value could be a λ–abstraction instead of constant. Once the value is needed, we search the association list for the value.

If the value contains a λ–abstraction and it is not in weak head normal form [Peyt87], we reduce it to be a normal form and substitute it in the expression. In this case, we reduce the order of the function by 1. As the procedure continues, we finally reduce the higher order function to the first order function.

5.4.3. Compilation into Super–Combinators

The interpretive graph reduction has the disadvantage of requiring a complex set of tests to determine how a λ–expression is to be reduced. Many of these tests can be avoided if the λ–expressions are made more uniform, in which case they can even be compiled into a fixed set of instructions to perform the graph reduction. Toward this end, Hughes [Hugh82] developed the concept of fully lazy λ–lifting, whereby λ–expressions may be transformed into supercombinators which are suitable for compilation into a fixed set of instructions. This work has opened up the possibility of abstract and real machines to execute the set of instructions, the most notable of which is the G–machine ([Augu84], [John84], [Kieb85]). Our approach is to use the same idea as Hughes for generating supercombinators, followed by translation into code which will execute the supercombinators. Unlike the G–machine approach, however, we will generate code expressed in Two–Level Grammar. It will be shown that supercombinator reduction rules can be directly represented in TLG, and furthermore, the reduction can be implemented very efficiently by the techniques of Chapter 4. First, we give some preliminary definitions.

A *supercombinator* is defined by Peyton Jones [Peyt87] as

$S $= \lambda x_1 . \lambda x_2 . . . . \lambda x_n . E$

where

(i) E is not a λ–abstraction

(ii) $S contains no free variables

(iii) any λ–abstraction in E is also a supercombinator

(iv) $n \geq 0$

The supercombinator reduction rule is typically written as:

$S $x_1 x_2 ... x_n \rightarrow E$

Now we make the following observation. Supercombinator reduction for a complete set of arguments can be expressed exactly by a Two–Level Grammar rule:

reduce S of X1 and X2 and ... and Xn giving RESULT : reduce E giving RESULT.

where E is an expression containing the variables **X1, X2, ..., Xn**. When **S** is invoked with n arguments, these arguments will be immediately instantiated in **E** through unification. At the implementation level, this means that every occurrence of **X1, X2,** etc. will be associated with a pointer to the instantiation.

Furthermore, as desired in lazy evaluation, these arguments will remain unevaluated until needed by a reduction in E. This expression will then be further reduced until a **RESULT** is obtained. As a simplest example of the technique, consider the supercombinator:

$$\$ADD\ x\ y = x + y$$

This can be straightforwardly expressed in Two–Level Grammar as:

reduce add of X and Y giving RESULT : where X + Y = RESULT.

Because **where** is a strict function, **X** and **Y** will be evaluated immediately and the **RESULT** returned as a ground value. Supercombinators may also be called with more or less than the number of arguments required. In this case, the type of rule described above will not match the call to reduce. Therefore, we need additional rules to evaluate these calls correctly.

reduce S of X1 and X2 and ... and Xn and ARGUMENT_LIST giving RESULT :
reduce S of X1 and X2 and ... and Xn giving E,
reduce E of ARGUMENT_LIST to RESULT.

reduce E to E.

The first rule evaluates the supercombinator redex with the appropriate number of arguments and then attempts to apply the result to the remaining arguments. The second rule, which must be placed at the end of all the reduce rules because it is the default case, essentially guarantees that all supercombinator applications with less than the required number of arguments remain unevaluated.

It can be seen that the implementation of supercombinator reduction is straightforward. It should also be pointed out that this reduction can be performed quite efficiently using our Two–Level Grammar implementation techniques. Furthermore, the reduction rules can be accessed almost directly since each supercombinator has a distinct name which tags the reduce rules which are applicable. Finally, on a parallel machine, the pattern matching of the reduce rules would be done in parallel to instantaneously access the desired rule.

Our method to abstract free variables and maximally free expressions from a general λ–expression is a straightforward TLG implementation of the algorithms described in [Hugh82] and [Peyt87]. The primary innovation is the fact that supercombinators are generated as TLG rules, meaning that our transformation system can map these directly into C procedures. No further work, outside of possible optimizations to the code, needs to be done. The generated C supercombinator programs execute the fastest of all the methods we have described.

5.5. Polymorphic Type Inference

In addition to λ-reduction, it is also well-known that polymorphic type checking in functional languages can also be done using unification [Miln78]. These rules can be stated very succinctly in Two-Level Grammar:

infer type of apply F to X as TYPE2 :
infer type of F as apply to TYPE1 giving TYPE2,
infer type of X as TYPE1.

infer type of fun X = > E as apply to TYPE1 giving TYPE2 :
infer type of X as TYPE1, infer type of E as TYPE2.

Additional built-in functions and constants can be typed as follows:

infer type of + as apply to integer giving apply to integer giving integer.
infer type of = as apply to TYPE giving apply to TYPE giving boolean.
infer type of cons as apply to TYPE giving apply to list of TYPE giving list of TYPE.
infer type of INTEGER as integer.

Applying these rules to any functional expression will yield the appropriate types, including a polymorphic type, represented as an uninstantiated type variable in the above system. The rules above work for ordinary λ-calculus without *let* and *letrec* constructions. Adding these to our system introduces several complications due to the typing of variables. The same variable may have different types in different instances. We solve this problem by building an association list which binds variables to types. Types are represented as a pattern with uninstantiated variables representing polymorphic types. For example, the type of the S combinator:

$(* \rightarrow ** \rightarrow ***) \rightarrow (* \rightarrow **) \rightarrow * \rightarrow ***$

is represented in Two-Level Grammar as:

apply to (apply to TYPE1 giving apply to TYPE2 giving TYPE3)
giving apply to (apply to TYPE1 giving TYPE2)
giving apply to TYPE1 giving TYPE3.

The variables having the same name must have the same type when instantiated. However, if the S combinator is applied to different arguments, it is possible that these variables will take different instantiations in different places. For a more detailed discussion of this subject, the reader is referred to [Peyt87]. We have developed an extensive set of Two-Level Grammar rules for polymorphic type inference in λ-expressions with *let* and *letrec* expressions and a wide variety of built-in functions. The primary representations used are:

let V1 = E1 and V2 = E2 and ... and Vn = En in E
letrec V1 = E1 and V2 = E2 and ... and Vn = En in E

Our type checker first sequences through the variable bindings, building the association list. Then the type of the expression E is computed. Finally, the types are unified to verify that they are correct.

5.6. Summary

All of the Two–Level Grammar definitions described in this chapter have been implemented in Prolog and C. Using this methodology, we have been able to develop a large number of compilers very quickly. The C implementations of these compilers has proven to be far superior to compilers for the same language coded in ML and Prolog and are comparable to those that are coded by hand. Furthermore, the correctness of these compilers is more easily proven because of the logic foundations underlying the formal specifications.

Our Two–Level Grammar specifications of denotational semantics allow the compilers to be executed in parallel wherever possible according to the rules for parallel implementation of logic programs. This work can be extended to consider additional aspects of parallel implementation of denotational semantics–directed compilation. In particular, it would be interesting to give Two–Level Grammar definitions of other forms of language constructs such as tasks, communicating processes, and logic programming itself. It is also of interest to consider logic program transformational aspects (e. g., see [Hogg84]) of our definitions, especially the denotational semantics–directed synthesis of source programs from high level specifications and the synthesis of the denotational definitions themselves.

A number of novel techniques for implementing functional programming languages using Two–Level Grammar have also been presented. By applying theses techniques, we have implemented several versions of a λ–machine using strict and lazy tree reduction, strict and lazy graph reduction and compilation into super–combinators. The development of a general λ–calculus reduction system in Two–Level Grammar extends earlier work done for Prolog by Narain [Nara86] who defined a method of delaying evaluation of predicates and Abramson [Abra86] who defined a combinator reduction machine in Prolog. Implementing functional programming languages in Two–Level Grammar allows further development of the interface between the two language paradigms of functional and logic programming that can be used for a wide variety of applications (see [DeGr86]) for a survey of additional work in this area).

REFERENCES

[Abra86] Abramson, Harvey, "A Prological Definition of HASL: A Purely Functional Language with Unification-Based Conditional Binding Expressions," in [DeGr86], 1986, pp. 73–129.

[Abri79] Abrial, J. R., Schuman, S. A. and Meyer, B., *Specification Language Z*, Massachusetts Computer Associates, Inc., Boston, MA, 1979.

[Aho86] Aho, Alfred V., Sethi, Ravi and Ullman, Jeffrey D., *Compilers. Principles, Techniques, and Tools*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[Augu84] Augustsson, Lennart, "A Compiler for Lazy ML," *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984, pp. 218–227.

[Back60] Backus, John W., et al., ed., "Report on the Algorithmic Language ALGOL 60," *Communications of the ACM 3*, 5 (May 1960), 299–314.

[Bans88] Bansal, Arvind K. and Sterling, Leon, "An Abstract Interpretation Scheme for Logic Programs Based on Type Expression," *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, 1988.

[Baue85] Bauer, Friedrich L., et al., *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP*, Springer-Verlag Lecture Notes in Computer Science, Vol. 183, Berlin, 1985.

[Bjor82] Bjorner, Dines and Jones, Cliff B., *Formal Specification and Software Development*, Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1982.

[Brya86a] Bryant, Barrett R., et al., "Two-Level Grammar as a Programming Language for Data Flow and Pipelined Algorithms," *Proceedings of the IEEE Computer Society 1986 International Conference on Computer Languages*, 1986, pp. 136–143.

[Brya86b] Bryant, Barrett R., Edupuganty, Balanjaninath and Hull, Lee S., "Two-Level Grammar as an Implementable Metalanguage for Axiomatic Semantics," *Computer Languages 11*, 3/4 (1986), 173- 191.

[Brya88a] Bryant, Barrett R., et al., "Two-Level Grammar: Data Flow English for Functional and Logic Programming," *Proceedings of the 1988 ACM Computer Science Conference*, 1988, pp. 469–474.

[Brya88b] Bryant, Barrett R. and Edupuganty, Balanjaninath, "Enhancements in the Two-Level Grammar Functional Programming Language," *Proceedings of the 1988 International Computing Symposium*, 1988, pp. 157–162.

[Brya89] Bryant, Barrett R. and Pan, Aiqin, "Rapid Prototyping of Programming Language Semantics Using Prolog," *Proceedings of COMPSAC '89, the Thirteenth Annual International Computer Software and Applications Conference*, 1989, pp. 439–446.

[Burs77] Burstall, R. M. and Darlington, John, "A Transformation System for Developing Recursive Programs," *Journal of the Association for Computing Machinery 24*, 1 (January 1977), 44–67.

[Burs81] Burstall, R. M. and Goguen, J. A., "An Informal Introduction to Specifications Using CLEAR," in *The Correctness Problem in Computer Science*, ed. R. S. Boyer and J. Strother Moore, Academic Press, Inc., London, 1981, pp. 185–213.

[Chan73] Chang, Chin-Liang and Lee, Richard C., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, Inc., Boston, Mass., 1973.

[Chur41] Church, Alonzo, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, NJ, 1941.

[Clea77] Cleaveland, J. C. and Uzgalis, R. C., *Grammars for Programming Languages*, Elsevier North-Holland, New York, 1977.

[Cloc87] Clocksin, William F. and Mellish, Christopher S., *Programming in Prolog*, 3rd ed., Springer-Verlag, Berlin, 1987.

[Cone87] Conery, John S., *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, Boston, Massachusetts, 1987.

[Dahl87] Dahl, Ole-Johan, "Object-Oriented Specification," *Research Directions in Object-Oriented Programming*, eds. Bruce Shriver and Peter Wegner, MIT Press, Cambridge, MA, 1987, pp. 561-576.

[Darl82] Darlington, John, "Program Transformation," in *Functional Programming and its Applications: An Advanced Course*, eds. J. Darlington, P. Henderson, and D. A. Turner, Cambridge University Press, Cambridge, England, 1982, pp. 193-215.

[Debr87] Debray, Saumya K., "Static Inference of Modes and Data Dependencies in Logic Programs," Technical Report TR 87-24, Department of Computer Science, University of Arizona, Tucson, Arizona, 1987.

[DeGr86] DeGroot, Doug and Lindstrom, Gary, *Logic Programming. Functions, Relations, and Equations*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[Edup85] Edupuganty, Balanjaninath and Bryant, Barrett R., "Two-Level Grammars for Automatic Interpretation," *Proceedings of the 1985 ACM Annual Conference*, 1985, pp. 417-423.

[Edup87] Edupuganty, Balanjaninath, *Two-Level Grammar: An Implementable Metalanguage for Consistent and Complementary Language Specifications*, Ph. D. Dissertation, Department of Computer and Information Sciences, The University of Alabama at Birmingham, June 1987.

[Edup89] Edupuganty, Balanjaninath and Bryant, Barrett R., "Two-Level Grammar as a Functional Programming Language," *The Computer Journal* 32, 1 (February 1989), 36-44.

[Feat82] Feather, Martin S., "A System for Assisting Program Transformation," *ACM Transactions on Programming Languages and Systems* 4, 1 (January 1982), 1-20.

[Feat87] Feather, Martin S., "A Survey and Classification of Some Program Tranformation Approaches and Techniques," in [Meer87], 1987, pp. 165-195.

[Fuch87] Fuchi, K. and Furukawa, K., "The Role of Logic Programming in the Fifth Generation Computer Project," *New Generation Computing 5*, 1 (1987), 3-28.

[Gogu79] Goguen, J. A. and Tardo, J. J., "An Introduction to OBJ: a Language for Writing and Testing Formal Algebraic Program Specifications," *Proceedings of the Conference on Specifications of Reliable Software*, 1979, pp. 170-189.

[Hoar74] Hoare, C. A. R. and Lauer, P. E., "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," *Acta Informatica 3* (1974), 135-153.

[Hoar85] Hoare, C. A. R. and Shepherdson, J. C., *Mathematical Logic and Programming Languages*, Prentice-Hall International, Englewood Cliffs, NJ, 1985.

[Hogg84] Hogger, Christopher J., *Introduction to Logic Programming*, Academic Press, London, 1984.

[Hopc79] Hopcroft, John E. and Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Co., Reading, MA, 1979.

[Hugh82] Hughes, R. J. M., "Super-Combinators: A New Implementation Method for Applicative Languages," *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, 1982, pp. 1-10.

[John75] Johnson, S. C., "YACC - Yet Another Compiler Compiler," C. S. Technical Report #32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.

[John84] Johnsson, Thomas, "Efficient Compilation of Lazy Evaluation," *Proceedings of the 1984 ACM Conference on Compiler Construction*, 1984, pp. 58-69.

[Kern88] Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, 2nd ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

[Kers86] Kerschberg, Larry, ed., *Expert Database Systems, Proceedings from the First International Workshop*, Benjamin/Cummings Publishing Co., Menlo Park, CA, 1986.

[Kieb85] Kieburtz, Richard B., "The G-Machine: A Fast, Graph-Reduction Evaluator," *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1985, pp. 400–413.

[Kowa85] Kowalski, R., "The Relation Between Logic Programming and Logic Specification," in [Hoar85], 1985, pp. 11–27.

[Land64] Landin, Peter J., "The Mechanical Evaluation of Expressions," *The Computer Journal 6* (1964), 308–320.

[Lisk86] Liskov, Barbara and Guttag, John, *Abstraction and Specification in Program Development*, MIT Press, Cambridge, MA, 1986.

[Malu82] Maluszynski, J. and Nilsson, J. F., "Grammatical Unification," *Information Processing Letters 15*, 4 (1982), 150–158.

[Malu84] Maluszynski, J., "Towards a Programming Language Based on the Notion of Two-Level Grammar," *Theoretical Computer Science 28* (1984), 13–43.

[Mann74] Manna, Zohar, *Mathematical Theory of Computation*, McGraw-Hill Book Co., New York, 1974.

[Mann79] Manna, Zohar and Waldinger, Richard, "Synthesis: Dreams => Programs," *IEEE Transactions on Software Engineering SE-5*, 4 (July 1979), 294–328.

[Mann80] Manna, Zohar and Waldinger, Richard, "A Deductive Approach to Program Synthesis," *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 90–121.

[Marc76] Marcotty, Michael, Ledgard, Henry and Bochmann, Gregor V., "A Sampler of Formal Definitions," *Computing Surveys 8*, 2 (1976), 191–276.

[Meer87] Meertens, L. G. L. T., ed., *Program Specification and Transformation*, Elsevier Science Publishers, Amsterdam, 1987.

[Miln78] Milner, Robin, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Science 17*, (1978), 348–75.

[Miln90] Milner, Robin, Tofte, Mads and Harper, Robert, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[Nara86] Narain, Sanjay, "A Technique for Doing Lazy Evaluation in Logic," *Journal of Logic Programming 3* (1986), 259–276.

[Oste86] Osterhaug, Anita, *A Guide to Parallel Programming*, Sequent Computer Systems, Beaverton, Oregon, 1986.

[Paga79] Pagan, Frank G., "ALGOL 68 as a Metalanguage for Denotational Semantics," *The Computer Journal 22*, 1 (1979), 63–66.

[Paga81] Pagan, Frank G., *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[Pan89] Pan, Aiqin and Bryant, Barrett R., "Logic Programming Implementation of Functional Programming Languages," *Proceedings of TENCON '89, Fourth IEEE 10th Region International Conference*, 1989, pp. 174–178.

[Pan90a] Pan, Aiqin and Bryant, Barrett R., "Denotational Semantics-Directed Compilation Using Prolog," *Proceedings of SAC-90, 1990 ACM Symposium on Applied Computing*, 1990, pp. 122–127.

[Pan90b] Pan, Aiqin and Bryant, Barrett R., "Two-Level Grammar as a Specification Language for an Intelligent Database Query System," submitted for publication.

[Pepp84] Pepper, Peter, ed., *Program Transformation and Programming Environments*, Springer-Verlag, Berlin, 1984.

[Peyt87] Peyton Jones, Simon L., *The Implementation of Functional Programming Languages*, Prentice-Hall International, Englewood Cliffs, N. J., 1987.

[Schm86] Schmidt, David A., *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, MA, 1986.

[Sint67] Sintzoff, M., "Existence of a van Wijngaarden Syntax for Every Recursively Enumerable Set." *Annales de la Societe Scientifique de Bruxelles 81*, 2 (1967), 115–118.

[Sund87] Sundararaghavan, K. R., Edupuganty, Balanjaninath and Bryant, Barrett R., "Towards a Two–Level Grammar Interpreter," *Proceedings of the 25th Annual Conference of the Southeast Region of the ACM*, 1987, pp. 81–85.

[Thak87] Thakkar, S. S., ed., *Selected Reprints on Dataflow and Reduction Architectures*, IEEE Computer Society Press, Washington, D. C., 1987.

[Turn85] Turner, David A., "Functional Programs as Executable Specifications," in [Hoar85], 1985, pp. 29–54.

[Turn90] Turner, David A., "An Overview of Miranda," in *Research Topics in Functional Programming*, ed. David A Turner, Addison–Wesley Publishing Co., Reading, MA, 1990, pp. 1–16.

[Ullm88] Ullman, Jeffrey D., *Principles of Database and Knowledge–Base Systems, Volume I*, Computer Science Press, Rockville, MD, 1988.

[Vuil73] Vuillemin, J., *Proof Techniques for Recursive Programs*, Ph. D. Dissertation, Stanford University, Stanford, California, 1973.

[Wegn80] Wegner, Lutz Michael, "On Parsing Two–Level Grammars," *Acta Informatica* 14 (1980), 175–193.

[Wijn65] van Wijngaarden, A., Orthogonal Design and Description of a Formal Language, Technical Report MR 76, Mathematisch Centrum, Amsterdam, 1965.

[Wijn74] van Wijngaarden, A., "Revised Report on the Algorithmic Language ALGOL 68." *Acta Informatica* 5 (1974), 1–236.

[Wise86] Wise, Michael J., *Prolog Multiprocessors*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[Zobe87] Zobel, J., "Derivation of Polymorphic Types for Prolog Programs," *Proceedings of the 4th International Conference on Logic Programming*, 1987, pp. 817–838.

GRADUATE SCHOOL
UNIVERSITY OF ALABAMA AT BIRMINGHAM
DISSERTATION APPROVAL FORM

Name of Candidate Aiqin Pan

Major Subject Computer and Information Sciences

Title of Dissertation Automatic Transformation of High-Level Logic Specifications Into High-Performance Target Code

Dissertation Committee:

Barrett R. Bryant, Chairman

Robert M. Hyatt

Edwin L. Battistella

Warren D. Jones

Director of Graduate Program Warren D. Jones

Dean, UAB Graduate School Anthony Barnard

Date 1/16/91

PS-1428