

ABSTRACT

Java Bytecode Compilation for
High-Performance, Platform-Independent Logical Inference

Ashish Arte

Mentor: David B. Sturgill, Ph.D.

Automated reasoning systems are powerful computer programs capable of solving complex problems. They are characterized as computationally intensive having high performance requirements. Very few reasoning systems have been implemented in Java so far; its performance is regarded as an impediment to its use as a programming language for computationally intensive applications such as automated reasoning. In this thesis we discuss techniques that motivate the use of Java as the underlying platform to design a framework for high-performance logical inference. The techniques are centered around the idea of using a specialized compiler that can generate Java classes which contain Java bytecodes customized for performing reasoning efficiently. The benefit of generating bytecodes customized for logical inference is reflected in the improved performance observed from the experiments conducted.

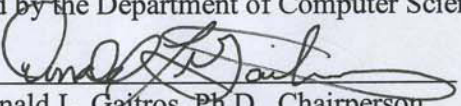
Java Bytecode Compilation for High-Performance, Platform-Independent Logical
Inference

by

Ashish Arte, B.E.

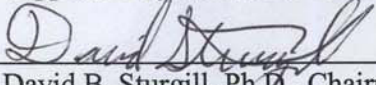
A Thesis

Approved by the Department of Computer Science

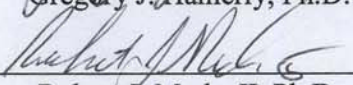

Donald L. Gaitros, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

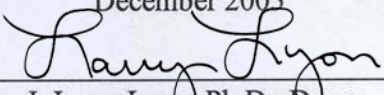
Approved by the Thesis Committee


David B. Sturgill, Ph.D., Chairperson


Gregory J. Hamerly, Ph.D.


Robert J. Marks II, Ph.D.

Accepted by the Graduate School
December 2005


J. Larry Lyon, Ph.D., Dean

Copyright © 2005 by Ashish Arte

All rights reserved

TABLE OF CONTENTS

LIST OF FIGURES	vii
LIST OF TABLES	x
ACKNOWLEDGMENTS	xi
DEDICATION	xii
1 Introduction	1
1.1 Inference in First-Order Logic	2
1.1.1 Syntax and Semantics	2
1.1.2 Using First-Order Logic	4
1.1.3 Forward and Backward Chaining	6
1.1.4 Factors Affecting Performance Of Logical Deduction	9
1.2 Logical Reasoning Systems in Java	10
1.2.1 The Java Logic Interpreter	11
1.2.2 The Logic-to-Java Compiler	12
1.2.3 The Logic-to-Bytecode Compiler	12
1.3 Document Overview	13
2 Related Work	14
2.1 First-Order Reasoning and Resolution	14
2.1.1 Resolution Principle	14
2.1.2 Unification	15
2.1.3 Resolution-Refutation	18
2.1.4 Resolution Strategies	21
2.1.5 Model Elimination	23
2.1.6 Logic Programming Paradigm	26
2.1.7 Warren Abstract Machine	28

2.2	First-Order Logic Theorem Provers	30
2.2.1	Prolog Technology Theorem Provers	30
2.2.2	Theorem Provers in Java	32
3	Logical Inference In Java	35
3.1	System Architecture	35
3.1.1	Java Representation of Logic Formulae	35
3.1.2	Generic Unification	38
3.1.3	Efficient ME Extension	40
3.1.4	Efficient ME Reduction	44
3.1.5	Search Mechanism	45
3.1.6	System Components	47
3.2	Java Logic Interpreter	48
3.2.1	Preprocessor	48
3.2.2	Inference Engine	50
3.3	Execution as Logical Inference	52
3.3.1	Mechanism for ME Extension	52
3.3.2	Compiling Rule Instances	55
3.3.3	Logic-to-Java Compiler Architecture	65
3.4	Refinements	67
3.4.1	Identical-Ancestor Pruning Rule	67
3.4.2	Rule Indexing	69
3.4.3	Rule Caching	70
4	The Java Inference Engine	72
4.1	Compiling for Java Virtual Machine	72
4.1.1	Java Virtual Machine Architecture	72
4.1.2	Java Class File Format	77

4.1.3	JVM Instruction Set	79
4.2	Logic-to-Bytecode Compiler	83
4.2.1	Compiling Java Classes	84
4.2.2	System Components	96
4.3	Refining Compiled Logic	96
4.3.1	Bytecode Refinements	97
4.3.2	Dereference Loop Refinements	101
4.3.3	Stack-Based Term Decomposition	105
5	Evaluation	116
5.1	Test Problem Suite	116
5.1.1	Class 1: Long-Skinny Proof Trees	117
5.1.2	Class 2: Fat-Bushy Proof Trees	118
5.1.3	Class 3: Long Variable Bindings	119
5.1.4	Class 4: Complex Terms	120
5.1.5	Class 5: Propositional Logic	122
5.1.6	Test Environment	123
5.2	Configuring the Logic-to-Bytecode Compiler System	124
5.2.1	Rule Caching	124
5.2.2	Bytecode Refinements	126
5.2.3	Dereference Loop Refinements	130
5.2.4	Stack-based Term Decomposition	130
5.2.5	Efficient Logic-to-Bytecode Compiler Configuration	132
5.3	Interpreted versus Compiled Execution	134
5.4	Standard versus Specialized Compiler	136
5.5	Experiment With TPTP Problem Suite	138

6	Conclusion	141
6.1	Future Work	143
	BIBLIOGRAPHY	146

LIST OF FIGURES

1.1	Logical Connectives	3
1.2	Derivation Techniques Illustration Axioms	4
1.3	Forward-Chaining Technique Proof Tree	5
1.4	Backward-Chaining Technique Proof Steps	8
1.5	Backward-Chaining Proof Tree	9
2.1	Example Terms Illustrating Unification	15
2.2	Blocks World Domain Example	19
2.3	Axioms Representing Blocks World	20
2.4	Proof Tree Illustrating Resolution-Refutation	21
2.5	Use Of Resolution-Refutation To Answer Questions	22
2.6	Model Elimination Proof Sample Clauses	25
2.7	Proof Tree Illustrating Model Elimination	26
3.1	Java Class Representation of First-Order Logic Constructs	36
3.2	Pseudocode of Occurs Check Algorithm	39
3.3	ME Extension Modeled As Tree Expansion	41
3.4	ME Reduction Modeled As Tree Operation	42
3.5	ME Reduction Pseudocode	45
3.6	The Java Logic Interpreter System Components	48
3.7	Clause Set To Illustrate The Java Logic Interpreter Proof Process . .	49
3.8	First Two Proof Steps Of Java Logic Interpreter	50
3.9	Last Two Proof Steps Of Java Logic Interpreter	51
3.10	Procedures Generating Java Code To Build Rules	55
3.11	Java Code Generated To Build A Rule	57
3.12	Procedures Generating Rule's Inference Instructions	59

3.13	Inference Instructions Generated As Java Code	64
3.14	The Logic-to-Java Compiler System Components	65
3.15	Contrapositives Of A Clause Set	66
3.16	Clause Set To Illustrate Identical-Ancestor Pruning Rule	67
3.17	Proof Illustrating Identical-Ancestor Pruning Rule	68
4.1	Internal Architecture Of the Java Virtual Machine	73
4.2	JVM Local Variables Section	75
4.3	JVM Operand Stack Usage	76
4.4	Java Class File Format	77
4.5	Method's Bytecode Stream	81
4.6	Inference Instructions Generated As Java Bytecodes	96
4.7	The Logic-to-Bytecode Compiler System Components	97
4.8	Efficient Variable Allocation Refinement	98
4.9	Store and Fetch Refinement	99
4.10	Duplicate Load Elimination Refinement	100
4.11	Mnemonics Of A While Loop	102
4.12	Techniques To Implement Dereference Loop	103
4.13	Tree representation of a term	105
4.14	Unification modeled as concurrent tree traversal	106
4.15	Stack-Based Term Decomposition - A	111
4.16	Stack-Based Term Decomposition - B	112
4.17	Stack-Based Term Decomposition - B	114
5.1	The Class 1 Clause Set And Goals	116
5.2	Proof Illustrating Class 1 Long-Skinny Tree	117
5.3	The Class 2 Clause Set And Goals	118
5.4	Proof Illustrating Class 2 Short-Bushy Trees	119

5.5	The Class 3 Clause Set And Goals	120
5.6	Proof Illustrating Class 3 Long Chains Of Variable Bindings	121
5.7	Term Illustrating Class 4 Problems	122
5.8	The Class 4 Clause Set And Goals	123
5.9	Plot To Evaluate Rule Caching	126
5.10	Plot To Evaluate Bytecode Refinements	128
5.11	Plot To Evaluate Stack-Based Term Decomposition	132
5.12	Plot Of Java Logic Interpreter v/s Logic-to-Java Compiler System . .	136
5.13	Plot Of Logic-to-Java Compiler v/s Logic-to-Bytecode Compiler . . .	138
5.14	Plot For TPTP Problem Library	139

LIST OF TABLES

5.1	Test Results For Rule Caching	125
5.2	Test Results For Bytecode Refinements	127
5.3	Test Results For All Dereference Loop Implementations	129
5.4	Test Results For Stack-Based Term Decomposition	131
5.5	Test Results For Efficient Logic-to-Bytecode Compiler Configuration	133
5.6	Test Results For Java Logic Interpreter and Logic-to-Java Compiler .	135
5.7	Test Results For Standard Compiled And Custom Compiled	137

ACKNOWLEDGMENTS

I gratefully acknowledge my advisor, Dr. David Sturgill, whose expertise in this field helped me improve my technical knowledge as well as my writing abilities. He took a lot of trouble to read my thesis over and over again, improving it each time. I really appreciate his patience, which enabled me to put my research in writing. This thesis would not have been possible without his inspiration.

I extend a special thanks to my committee members Dr. Hamerly, Dr. Marks, for taking time to read my thesis at such short notice. I would also like to thank all the faculty and the ever-helpful staff of the computer science department for providing us an excellent environment to study computer science. I take this opportunity to thank my colleagues at Open Sky Software, Jean, Steve, Paul and Henry, for their encouragement and support and allowing me to make those countless trips to Baylor during office hours.

Finally, thank you to Laxmi, who now has a special place in my life, whose motivation has helped me complete this thesis.

*To my parents, who have always been the source of inspiration and support. I can
never thank you enough for your sacrifice and love.*

CHAPTER ONE

Introduction

Logic is concerned with the principles of reasoning and valid inference. Automated reasoning deals with the mechanization of formal reasoning using logical principles and notations. Automated reasoning systems are powerful computer programs capable of solving complex problems. They solve problems irrespective of the problem domain, thus allowing people from various fields to take advantage of general-purpose automated reasoning systems (Bundy 1999). For example, hardware engineers use automated reasoning systems to validate circuit designs, compiler designers use automated reasoning principles to perform runtime validation of programs, and mathematicians use them to solve complex problems from mathematical domains (van Caneghem and Warren 1986).

In spite of their power, very few application domains extensively use general-purpose automated reasoning systems. One reason for this is that automated reasoning systems are not intuitive and do not learn easily from past problem solving experience. Another important factor that inhibits the widespread use of automated reasoning systems is the high degree of computation overhead involved in automated deduction (David A. Plaisted 2000). Two common approaches to compensate for this overhead are to design systems that execute faster and to introduce parallelization in the system. The design of our reasoning system provides a suitable platform for employing both these approaches for automated reasoning. Its architecture-independent Java-based implementation provides support for parallel processing in heterogeneous environments. Its execution speed is improved by exploiting the similarities between the operational principles of a powerful reasoning strategy and those of Java's runtime environment. A specialized compiler is designed that exploits the similarities by generating Java bytecodes customized for the logical reasoning domain. Java programs

typically incur more overhead than native-compiled languages like C++. Refinements introduced for the custom generated bytecodes expedite the inference operations employed by the reasoning system, to compensate for some of Java’s runtime overhead.

1.1 Inference in First-Order Logic

Our reasoning system performs logical reasoning on problems defined in first-order logic. The two issues faced when selecting a language are expressiveness and complexity of reasoning. Languages that are very expressive are accompanied by very complex inference procedures, while languages that allow efficient inference are not very expressive. First-order logic balances these two issues relatively well. This section presents an informal introduction to first-order logic with the help of an example that uses family relationships as its domain. For a more formal definition of first-order logic refer to the collection of papers in (Cartwright and McCarthy 1979).

1.1.1 Syntax and Semantics

The domain of family relationships includes facts such as “Homer is the father of Lisa” and “Marge is the wife of Homer.” Clearly, the objects in our domain are people. In first-order logic such objects are represented by *constant symbols*. We employ the convention of starting constant symbols with a lower-case letter. For example, *homer*, *marge*, and *lisa* can be used to represent the family members Homer, Marge, and Lisa respectively. First-order logic uses *function symbols* to describe the relation between different domain objects. Each function symbol refers to a many-to-one mapping between objects in the domain. For example, function symbol *father(lisa)*, could be used to represent “Father of Lisa.” *Variables* in first-order logic stand for arbitrary constants and allow us to make general statements about the domain. For instance, *father(X)* can be used to generalize *father(lisa)* to represent father of any object *X*. *Terms* in first-order logic are expressions composed of constant symbols, variables,

Not	\neg
And	\wedge
Or	\vee
Is Implied By	\leftarrow
Implies	\rightarrow
For All	\forall
There Exists	\exists

Figure 1.1. Logical Connectives

function symbols, and terms themselves. For example, t and $f(t_1, \dots, t_n)$ are terms where each t_i is a term.

Another basic element of first-order language is the *predicate*. Predicates capture relationships between domain objects. Just like function symbols, predicates can take arguments. Predicates combine terms into statements called *literals*. For example, the literal $loves(marge, lisa)$ can be used to say Marge loves Lisa. A more general statement: $loves(mother(X), X)$ can be used to say every mother loves her child. In both these examples, *loves* is a predicate symbol.

A *logic formula* is either a literal, negated literal, or two literals joined by one of the logical connectives **And**, **Or**, or **Is Implied By**. Figure 1.1 summarizes the symbols used for the logical connectives in place of their English variants. Logical formulae allow us to make statements about the relation between the truths of various literals. For example, the formula: $elder(X, Y) \vee elder(Y, X) \vee same_age(X, Y)$ can be used to say that either X is elder than Y , Y is elder than X , or X and Y are of the same age. Here $elder(X, Y)$ is called the *subformula* of $elder(X, Y) \vee elder(Y, X) \vee same_age(X, Y)$

Variables in first-order logic have a special characteristic that they are either *free* or *bound*. A variable is a free variable if the truth-value of the formula in which it is used depends upon the value of the variable. Conversely, if the truth-value of a formula does not depend upon the values taken by a variable, then it is a bound

A1.	$\forall X \text{ man}(X) \rightarrow \text{likes}(X, \text{donuts})$	- All men like donuts.
A2.	$\forall X, \forall Y \text{ likes}(X, Y) \wedge \text{man}(X) \rightarrow \text{eats}(X, Y)$	- All men eat everything they like.
A3.	$\forall X, \forall Y, \forall Z \text{ is_off}(tv) \wedge \text{at_home}(X) \wedge \text{plays}(Y, Z) \rightarrow \text{listens}(X, Z)$	- If X is at home, and Y plays Z and the tv is off, then X listens to Z .
A4.	$\forall X, \forall Y, \forall Z \text{ listens}(X, Y) \wedge \text{eats}(X, Z) \rightarrow \text{happy}(X)$	- X is <i>happy</i> if X listens to Y and X eats Z .
A5.	$\text{man}(\text{homer})$	- Homer is a man.
A6.	$\text{at_home}(\text{homer})$	- Homer is at home.
A7.	$\text{plays}(\text{lisa}, \text{saxophone})$	- Lisa plays saxophone.
A8.	$\text{is_off}(tv)$	- TV is off.

Figure 1.2: Facts from a domain, used to demonstrate the logical deduction procedure in first-order logic, represented as axioms. First-order logic representation of each axiom along with the description of the fact it represents.

variable. For example, in: $\forall X \text{ } p(X) \rightarrow r(X, Y)$ X is a bound variable and Y is free.

A logical formula is said to be *true* if the relation referred to by the predicate symbol holds between the objects referred to by its arguments. A formula connected by **And** is a *conjunction*. This formula is true if each of its subformulas are true, otherwise it is false. A formula connected by **Or** is a *disjunction*. The formula is true if any one of its subformulas are true, and false otherwise. A formula prefixed by **Not** is true if and only if the formula is false. A formula connected by **Is Implied By** is called an *implication*. Implications are also known as *rules* or *if-then* statements. For example, $R \leftarrow (P \vee Q)$ is an implication, where $P \vee Q$ is its *premise* and R is its *conclusion*. It means that if either P , Q , or both are true then R is true.

A formula under the scope of the universal quantifier \forall is true if its subformulae are true for all assignments of the variable to entities in the domain. A formula under the scope of the existential quantifier \exists is true if its subformulae are true for at least one assignment of the variable to an entity in the domain.

1.1.2 Using First-Order Logic

Using the terminology just introduced, Example 1.1 illustrates how to logically derive a new formula from a set of logical formulae called *axioms*. Axioms capture the

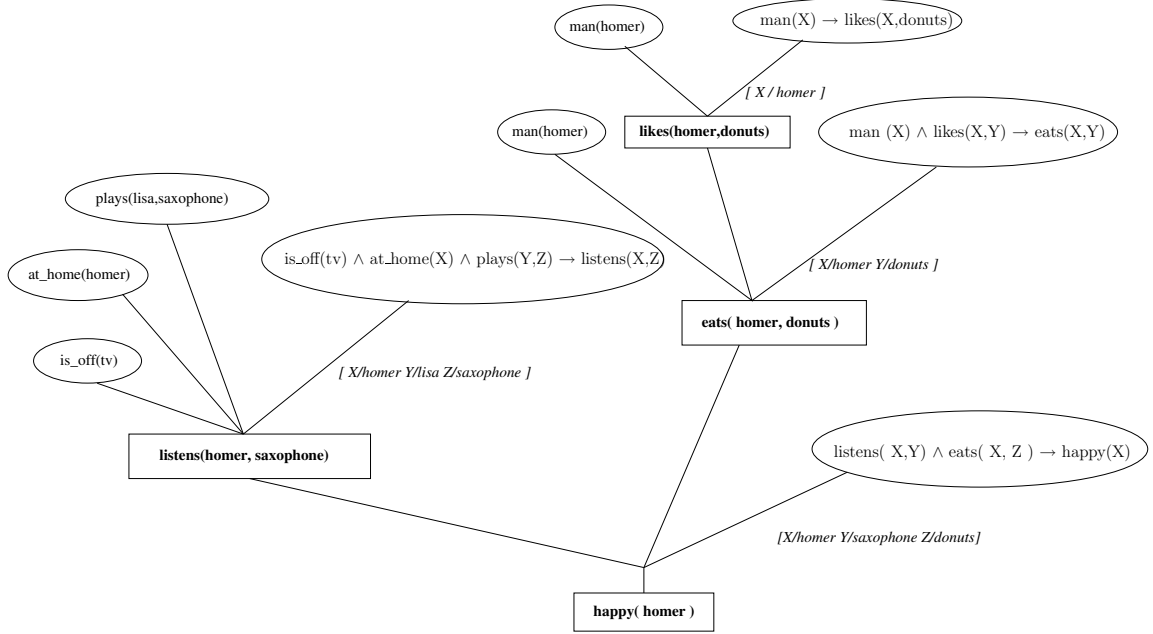


Figure 1.3: A forward-chaining proof tree to show $happy(homer)$ is true. At each inference step, the axiom appearing in an oval is a fact and the axiom in the box is the derived conclusion. Variables replaced and the values they replace are shown enclosed in brackets.

basic facts about a domain and define other concepts in terms of the basic facts. For example, in Figure 1.2 the first axiom can be used to represent the sentence all men like donuts and the second axiom to represent the sentence all men eat everything they like. This collection of axioms that describe the problem domain is called the *theory*.

Example 1.1: Consider the set of axioms of Figure 1.2 as true statements about the domain. Our goal is to logically derive that the formula $happy(homer)$ is true. The proof involves just four steps. In the first step, we use the axiom A1 and replace variable X with the constant symbol $homer$ to generate the formula $man(homer) \rightarrow likes(homer, donuts)$. Here we derive a specialized instance of a general formula by replacing one of its variables with a value. Since it is known from axiom A5 that $man(homer)$ is true, we can logically deduce that the formula $likes(homer, donuts)$ is true. In the second step, we replace variables X

and Y of axiom A2 with values *homer* and *donuts* respectively to generate the formula $likes(homer, donuts) \wedge man(homer) \rightarrow eats(homer, donuts)$. Using the formula, $likes(homer, donuts)$, derived in step 1 and the truth of axiom A5, we logically deduce that $eats(homer, donuts)$ is true. In the third step, we replace variable X with *homer*, Y with *lisa*, and Z with *saxophone* in axiom A3 to generate the formula $at_home(homer) \wedge plays(lisa, saxophone) \wedge is_off(tv) \rightarrow listens(homer, saxophone)$. We then use the truth of axioms A6, A7, and A8 to logically derive that $listens(homer, saxophone)$ is true. In the fourth and final step, we replace variables X with *homer*, Y with *saxophone*, and Z with *donuts* in axiom A4 to generate the formula $listens(homer, saxophone) \wedge eats(homer, donuts) \rightarrow happy(homer)$. Since we know that $listens(homer, saxophone)$ and $eats(homer, donuts)$ are true, we can logically conclude that $happy(homer)$, which is our goal, is also true. In this manner, we logically derive a new fact by replacing variables in the general axioms of the theory with specific values. Figure 1.3 shows the proof in a tree-shaped form.

A set of axioms *logically implies* a goal if, in every interpretation which assigns each of the axioms true, the goal is also assigned true. Logical implication is *semi-decidable* in first-order predicate logic. This means that if a goal follows from a set of axioms, there is an automated procedure which can determine that fact. In addition, any such procedure will not terminate for some combination of goal and axioms.

A first order proof procedure is *complete* if, for a set of axioms and a goal it eventually returns *true* when the axioms logically imply the goal. A first order proof procedure is *sound* if it never returns *true* for goals which are not logical implications of the axioms.

1.1.3 Forward and Backward Chaining

The proof procedure of Example 1.1 proceeds from the basic axioms towards the desired goal, deriving new formulae along the way. This technique is called *forward-*

chaining and is usually used when deriving conclusions based on new facts added to the theory. Forward-chaining is appropriate in situations where each new formula generated by the axioms of the theory contributes towards proving the goal. However, in practice it has certain limitations: First, forward-chaining uses an inference mechanism which may generate new formulae at each stage of the proof. This data-driven strategy is not directed towards solving any particular problem and makes forward-chaining unsuitable when there is a goal to prove. Second, it may be difficult to predict which of the new formulae or combination of formulae will lead towards proving the goal. An alternative approach is to start with the goal and attempt to find evidence to prove the goal. This strategy is called *backward-chaining*. Given a goal to prove, the backward-chaining strategy will first check to see if the goal matches the existing axioms in the theory. If it does, then that goal is proved. If it doesn't, then it will look for axioms whose conclusions match the goal. Choosing one such axiom, it attempts to prove any facts in the preconditions of the axiom using backward-chaining by setting these facts as new goals to prove.

Example 1.2: Figure 1.4 shows the steps for deriving $happy(homer)$ is true, from the axioms of Figure 1.2, using backward-chaining. The tree should be read depth-first and left to right. The derivation begins by looking in the set of axioms for an axiom which matches exactly with the goal $happy(homer)$. The set does not contain such an axiom. Replacing variable X by $homer$ in axiom A4, could help us prove the goal. However, to prove $happy(homer)$ by replacing X by $homer$, we have to prove that $eats(homer, Y)$ and $listens(homer, Z)$ are true. Thus, in order to prove the goal, we now have to prove two new goals or more precisely two new *subgoals*. Figure 1.4(a) shows the derivation step that generates these subgoals. We continue the derivation with the subgoal $eats(homer, Y)$. Again, the input set does not contain a direct match for this subgoal. However, we can use axiom A2 to prove it if we replace X by $homer$. As seen in Figure 1.4(b), this substitution generates two more subgoals $man(homer)$

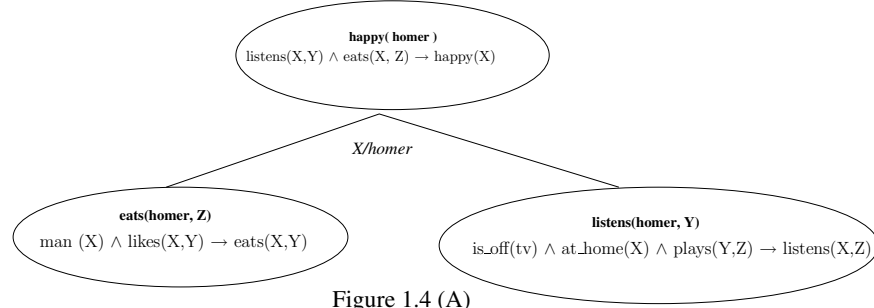


Figure 1.4 (A)

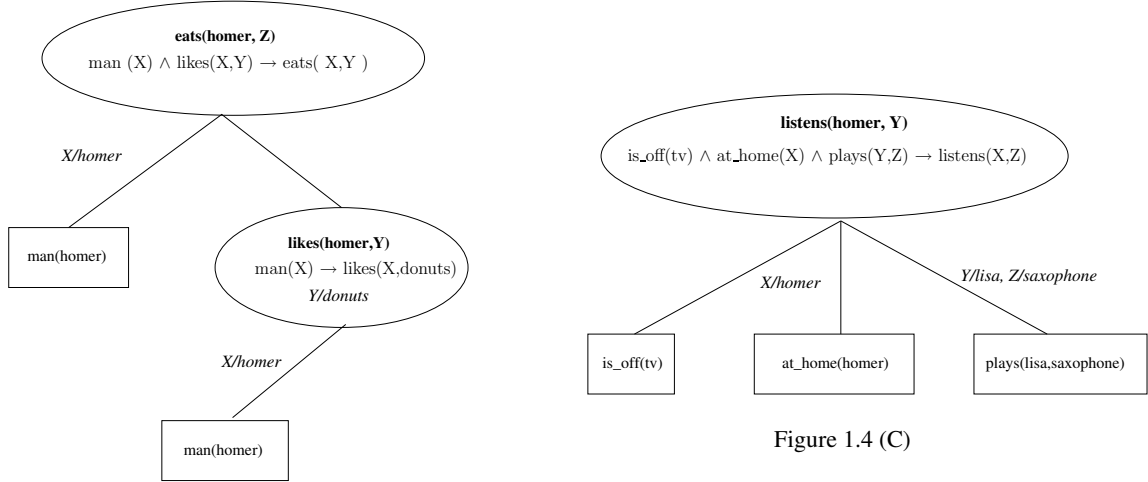


Figure 1.4 (B)

Figure 1.4 (C)

Figure 1.4: Each derivation step performed during a backward-chaining proof represented as a tree. The root of each tree is a conclusion to be derived. The nodes are the premises that derive the goal. The substitution associated with each node is written below the goal.

and $likes(homer, Y)$. Axiom A5 of the theory confirms that $man(homer)$ is true and so we proceed to the subgoal $likes(homer, Y)$. To prove this subgoal, we use axiom A1 and replace variable X by $homer$ and Y by $donuts$. This substitution generates another subgoal $man(homer)$. But $man(homer)$ is one of the given axioms of the theory, axiom A5. Therefore, $likes(homer, donuts)$ is true. Since, $man(homer)$ and $likes(homer, donuts)$ are true, logically implies that $eats(homer, Y)$ is true when Y is $donuts$. We then proceed to the subgoal $listens(homer, Z)$ derived in the first step. To prove this subgoal, we use axiom A3 and replace X by $homer$, Y by $lisa$ and Z by $saxophone$. This substitution further generates three new subgoals $at_home(homer)$,

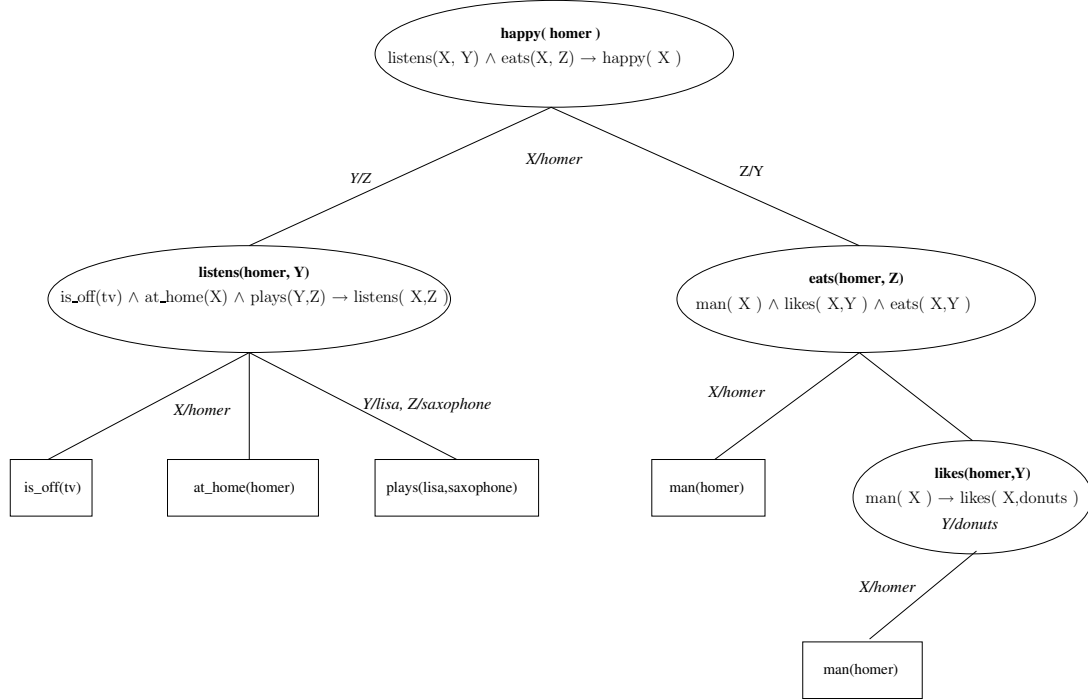


Figure 1.5: The proof tree for deriving $happy(homer)$. The root of each tree is the subgoal to be derived. Subgoals are shown enclosed in ovals. The nodes are the facts used to derive each subgoals. Facts from the theory are shown enclosed in boxes.

$plays(lisa, saxophone)$ and $is_off(tv)$. Each of these subgoals is true since they are the facts stated by axioms A6, A7 and A8 respectively as shown in Figure 1.2. The truth of these axioms implies that $listens(homer, Z)$ is true when Z is $lisa$. Thus, we prove that the two subgoals $listens(homer, Y)$ and $eats(homer, Z)$ are true, implying that our original goal $happy(homer)$ is true when Y is $donuts$ and Z is $saxophone$. Figure 1.5 shows the completed proof tree.

1.1.4 Factors Affecting Performance Of Logical Deduction

Examples 1.1 and 1.2 use the inference rule that given two logical formulae $p(a)$ and $p(X) \rightarrow q(X)$, if $p(a)$ is true it implies that $q(a)$ is also true. The key to this derivation is the *unification operation*, that makes the two formulae $p(a)$ and $p(X)$ identical by replacing the variable X with value a . A typical proof procedure

performs numerous substitutions in its attempts to unify terms. Thus, unification is performed a large number of times (Baader and Snyder 2001). As the complexity of the problem increases, the number of unifications performed also increases. Hence, an unification implementation forms an important factor for improving performance.

If we formulate the process of finding a truth of a goal, such as in Example 1.2, as a search process then the proof of Figure 1.5 is the solution to a search problem. Just like any search problem, a complex theory with a large number of axioms could result in long proofs as well as introduce large branching factor, which makes proving a goal a complex search problem. Therefore, the use of a smart search procedure which never overlooks a solution but avoids unnecessary search where possible is another important factor in determining the efficiency of the proof procedure. These factors and many more make automated-theorem proving a time and resource consuming process (Bundy 1999).

1.2 Logical Reasoning Systems in Java

Our reasoning system is designed to perform efficient reasoning in first-order logic. It is a sound and complete reasoning system for full first-order logic. The main proof procedure is based on Model Elimination (ME) and is implemented using Prolog-style compilation technique. The proof process is realized as a search procedure based on backtracking. Completeness of the search procedure is guaranteed by performing depth-first iterative-deepening bounded search (Stickel 1986).

The system, with its implementation in Java, is designed to exploit the similarities between the operating principles of Java runtime environment and logical deduction procedures. We choose an interpreted approach whereby inference steps are compiled into executable Java Virtual Machine (JVM) instructions, rather than a full-compilation which compiles into native machine code. While the interpreted approach of an abstract machine cannot match execution of natively compiled ma-

chine code in terms of efficiency, this approach provides a higher degree of portability. This allows us to build a framework for a system that can potentially perform logical reasoning, in parallel, on a network of computers, independent of their underlying architecture.

1.2.1 The Java Logic Interpreter

As a first step towards developing an efficient logical reasoning system in Java, we design the *Java Logic Interpreter*. Our main goal in designing this elementary reasoning system is to develop an efficient representation for first-order logical formulae in Java. The preprocessor incorporated within the Java Logic Interpreter performs this task. It first converts the logic formula that describes a problem into clauses and then represents each clause with a separate Java class. In order to generate this representation it utilizes specially designed Java classes for each type of first-order logic construct. Using these Java classes the preprocessor builds an in-memory representation of each clause from the theory. A Java-based inference engine operates on this representation of the clauses to prove goals based on the Model Elimination inference procedure.

In addition to generating an efficient representation of logic formulae in Java, the Java Logic Interpreter is characterized by its implementation of a generalized unification procedure. This procedure takes as input two first-order terms and attempts to unify them. For instance, in the first step of Example 1.2, given the goal $happy(homer)$ and the axiom $listens(X, Y) \wedge eats(X, Z) \rightarrow happy(X)$, the generalized unification procedure interprets the terms $happy(homer)$ and $happy(X)$ to replace variable X from the axiom with $homer$. Replacing the variable X by the constant symbol $homer$ derives $listens(homer, Y) \wedge eats(homer, Z) \rightarrow happy(homer)$, a specialized instance of the axiom. The generalized unification procedure is composed of a set of instructions that attempt to unify any pair of first-order terms and hence

is designed to be generic in nature. At each inference step, the Java Logic Interpreter invokes this generalized unification procedure to derive new facts or conclusions.

1.2.2 *The Logic-to-Java Compiler*

The inference procedure implemented by the Java Logic Interpreter interprets, at runtime, the structure of every pair of terms at each inference step. This process is similar to the execution of interpreted programming languages, which is inherently slow. Our second reasoning system, called the *Logic-to-Java Compiler*, overcomes this drawback. The Logic-to-Java Compiler consists of a sophisticated logic compiler. For each clause in the theory, the logic compiler generates customized inference instructions based on the structure of the clause's literals. The instructions implement specific steps to unify a literal with any term at runtime. To generate customized inference instructions for each clause at compile time, the logic compiler represents each literal from every clause with a separate Java class. This class contains instructions to build the structure of the literal and the clause when the Java class is instantiated at runtime. In addition, the each Java class also contains an inference method consisting of precompiled instructions to unify the literal, the class represents, with any term. At runtime, the Logic-to-Java Compiler attempts to unify a subgoal term and a literal term by invoking the literal's customized inference method. Thus, by precompiling inference instructions for each literal, the Logic-to-Java Compiler performs much of the computation once at compile time, which the Java Logic Interpreter otherwise performs repeatedly at runtime. This approach of executing precompiled inference procedures instead of employing generalized unification steps allows the Logic-to-Java Compiler implement an efficient inference procedure.

1.2.3 The Logic-to-Bytecode Compiler

While designing the Logic-to-Java Compiler we noticed that there is a natural correspondence between the execution principles of Java runtime environment and the ME inference procedure. Therefore, we extended the design of Logic-to-Java Compiler to develop our third system, called the *Logic-to-Bytecode Compiler*. The Logic-to-Bytecode Compiler has an enhanced logic compiler which directly generates Java classes, instead of the two step approach employed by the logic compiler of Logic-to-Java Compiler. The logic compiler of Logic-to-Java Compiler generates a Java class in two steps. It first generates a Java source file representing the literal and its clause and then using a standard Java compiler compiles the source file into a binary Java class used by its inference procedure at runtime. The alternative approach, of directly generating binary Java classes to represent each literal and its class, employed by the Logic-to-Bytecode Compiler not only eliminates the need for a standard Java language compiler, but also allows it to introduce machine-level refinements within each custom-generated inference procedure. Generating refined inference instructions allows the Logic-to-Bytecode Compiler to derive proofs efficiently and compensate for some of Java's runtime overhead, at the same time maintaining portability of the system across different platforms.

1.3 Document Overview

In the next chapter, we discuss various concepts related to the Model Elimination inference procedure necessary to understand our reasoning systems. Chapter 3 describes the design and operation of the Java Logic Interpreter and Logic-to-Java Compiler. Subsequently, in Chapter 4, we present the design of the Logic-to-Bytecode Compiler along with the Java bytecode optimizations we introduced. We conclude the report with an assessment of our work and an outlook to the future of automated reasoning with Java.

CHAPTER TWO

Related Work

In this chapter, we present an introduction to the automated reasoning paradigm that forms the basis of Model Elimination, the inference procedure employed by our three systems. We also briefly describe some existing Model Elimination implementations from the perspective of developing an efficient reasoning system for first-order logic.

2.1 *First-Order Reasoning and Resolution*

Automated reasoning systems use a variety of inference mechanisms. Some use natural deduction techniques (similar to those used by humans), while others employ machine-oriented techniques. We restrict our discussions to the inference mechanisms necessary to understand our reasoning system.

2.1.1 *Resolution Principle*

In 1965, Robinson introduced the resolution inference principle, a single machine-oriented inference rule for deriving proofs in first-order logic (Robinson 1965a). It is a powerful principle in the sense that it alone, as a single inference principle, forms a complete reasoning system for first-order logic. The resolution procedure is an algorithm that identifies whether a formula is valid using the resolution inference principle.

The resolution procedure applies to only those logical formulae which are expressed as a conjunction of *clauses*. A clause is a disjunction of positive and negative literals. The resolution inference principle has the following form:

$$\frac{(\alpha \vee \beta)(\neg\beta \vee \gamma)}{(\alpha \vee \gamma)}$$

- | | |
|-------------|--------|
| <i>i)</i> | $x(1)$ |
| <i>ii)</i> | $x(1)$ |
| <i>iii)</i> | $x(Y)$ |
| <i>iv)</i> | $x(Z)$ |

Figure 2.1. Example Terms Illustrating Unification

where $(\alpha \vee \beta)$ and $(\neg\beta \vee \gamma)$ are two clauses, representing arbitrary disjuncts of literals. The above rule says that for two clauses $(\alpha \vee \beta)$ and $(\neg\beta \vee \gamma)$, if one of the literals from clause $(\alpha \vee \beta)$ matches with the negation of a literal from the other clause $(\neg\beta \vee \gamma)$, then we can infer a third clause $(\alpha \vee \gamma)$, which is a disjunction of all the literals from the two clauses except for the two complementary literals β and $\neg\beta$. The derived clause $(\alpha \vee \gamma)$ is called the *resolvent*. The literals β and $\neg\beta$ are called *resolving literals*. This rule known as *ground resolution* is suitable only for propositional logic. Before we present the version of the resolution inference rule suitable for first-order logic, we introduce *unification*.

2.1.2 Unification

Unification is a fundamental concept behind the resolution inference rule for first-order predicate logic. It is a process that establishes whether two logical terms can be made syntactically equivalent to one another through variable substitution. Here is a simple example: Terms *(i)* and *(ii)* of Figure 2.1 are the same, so we say that they unify.

Now consider the two terms *(ii)* and *(iii)*. In this case, the terms are not identical; however, if we replace the variable Y with 1 then the two terms unify. This replacement, represented as $Y/1$, is called a *binding* where the variable Y binds to the value 1. Finally we consider the terms *(iii)* and *(iv)*. Again, these terms are not equivalent. We could unify them by making arbitrary bindings like $Y/1$, $Z/1$. However, a more general binding like Y/Z would allow us to unify the terms *(iii)* and

(iv) without making any unnecessary commitment on the values for these variables.

A *binding list* is a set of bindings of the form V_i/t_i , where V_1, V_2, \dots, V_n are variables and t_1, t_2, \dots, t_n are terms. The result of applying a binding θ to a term k , denoted as $k\theta$, is a term obtained by replacing every occurrence of variable V in k by term t , for each V/t pair in θ . A term k is an *instance* of a term j if there is a binding list θ such that $k = j\theta$. A term k is a *common instance* of terms j_1 and j_2 if there are substitutions θ_1 and θ_2 such that $k = j_1\theta_1$ and $k = j_2\theta_2$. A term k is *more general* than a term j if j is an instance of k but k is not an instance of j . A *unifier* of two terms k_1 and k_2 is a binding θ that makes the terms identical, $k_1\theta = k_2\theta$. If a unifier of two terms exists, then the terms are said to *unify*. A *most general unifier (MGU)* of two terms k_1 and k_2 is a binding θ that unifies k_1 and k_2 such that the common instance $k_1\theta$ is as general as any other common instance of k_1 and k_2 . That is, for any other unifier θ' of k_1 and k_2 , $k_1\theta$ is as general as $k_1\theta'$ and $k_2\theta'$.

When two terms j and k are to be unified, they are compared. If they are both **constants** then the result of the unification is a success if they are equal, else it is a failure. If j and k are **variables** then j binds to k or vice versa and unification succeeds. If j is a **variable** and k a **term** and if j occurs in k then unification fails, otherwise j binds to k and unification succeeds. If j and k are terms of the form $f_x(x_1, x_2, \dots, x_n)$ and $f_y(y_1, y_2, \dots, y_n)$, then they unify if their functors f_x and f_y are identical, both the terms have same number of parameters and the i^{th} parameter of term j successfully unifies with i^{th} parameter of k . For instance, consider the unification of two terms $grandFather(X, Y)$ and $grandFather(bob, Z)$. Their unification begins by comparing the predicate symbol *grandFather* of the two terms. This symbol is identical in the two terms, hence unification proceeds to each sub-term of the two terms. The first sub-term pair is X and bob resulting in the binding X/bob . Next, we bind variable Y to Z . Since there are no more sub-terms and each of the sub-terms unified successfully, the unification of two terms $grandFather(X, Y)$ and

Algorithm 2.1: Unification

Input: Two arbitrary first-order terms j, k to be unified

Output: θ , the most general unifier of j and k , or failure.

Algorithm: Unify (j, k)

Initialize θ to be empty.

Case:

 If j and k are two identical constants or variables:
 return success

 Else If j is a variable that does not occur in k :
 bind j to k and insert (j/k) into θ
 return success

 Else If k is a variable that does not occur in j :
 bind k to j insert (k/j) into θ
 return success

 Else If j is $f(y_1, y_2, \dots, y_n)$, k is $f(z_1, z_2, \dots, z_n)$:
 for $i = 1$ to n
 result = Unify(y_i, z_i)
 if result is failure
 return failure
 return success

 Else
 Return failure

End Case

grandFather(bob, Z) succeeds with $(X/bob, Y/Z)$ as the MGU.

Algorithm 2.1 is the recursive unification algorithm to determine the MGU for any two first-order terms (Robinson 1965a). An important thing to note in it is the unification of a variable that *occurs* within a term. A variable X occurs in a term e if, X is variable that binds to a term e containing the variable X . If we attempt to bind X and e , the binding results in a cyclic structure which may cause the unification to loop for ever. Hence, the algorithm performs an *occurs check* before binding a variable to a term, to ensure that the variable does not occur within the term. The *occurs check*, if performed for all variables at each reasoning step, can affect the speed of the

inference process. Implementation of *occurs check* is necessary for the completeness.

2.1.3 Resolution-Refutation

The general resolution inference rule for first-order logic states that given two variable disjoint clauses ¹ $(C_1 \vee l_1)$ and $(\neg l_2 \vee C_2)$, where C_1 and C_2 are disjunctions of literals and l_1 and l_2 are literals, we deduce resolvent $(C_1 \vee C_2)\theta$ from $(C_1 \vee l_1)$ and $(\neg l_2 \vee C_2)$, where θ is the MGU of the two complementary unifiable literals l_1 and l_2 .

$$\frac{(C_1 \vee l_1)(\neg l_2 \vee C_2)}{(C_1 \vee C_2)\theta}$$

A clause in first-order logic is valid if and only if at least one of its literals is true. Thus, an empty clause is always false, since it has no true literal. A special case of the resolution rule is when C_1 and C_2 are empty. In this case, the resolution inference rule applied to the complementary unifiable literals l_1 and l_2 yields an empty clause, indicating that the input clause set consisting of clauses $(C_1 \vee l_1)$ and $(C_2 \vee \neg l_2)$ is an invalid clause set. Robinson proposed the use of this technique of detecting an invalid input clause set by deriving an empty clause to prove theorems in first-order logic using resolution.

Accordingly, in order to prove a goal from an input clause set T , we negate the goal and add it to T . Let us call this new set of clauses T_1 . Applying resolution to the clause set T_1 , we attempt to derive an empty clause. If the resolution yields an empty clause, we can deduce that the negation of the goal is invalid. Since it is known that the clause set T itself is valid, by contradiction, we can say that the goal statement is a valid clause of the theory T . This process of proving theorems by contradiction using resolution is called *resolution-refutation* (Robinson 1965a). The resolution-refutation inference procedure is *sound*, meaning that it will not deduce an empty clause if the input clause set is valid. It is also complete, in that it will always derive an empty clause for every invalid input clause set (Robinson 1965a).

¹ Two clauses are variable disjoint if they share no common variables

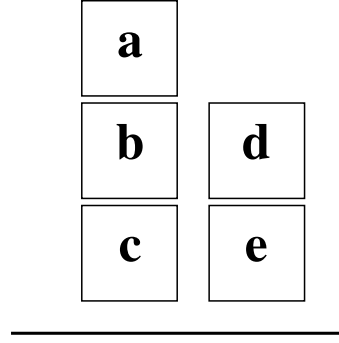


Figure 2.2. Blocks World Domain Example

A straightforward resolution-refutation proof procedure proceeds as follows. If there are n input clauses, temporarily designate clause n as the *focus clause*. Begin by resolving the focus clause n against each of the clauses until a resolvent is obtained. This resolvent clause, labeled $n + 1$, becomes the new focus clause. This clause is now resolved against all the other clauses until another resolvent is obtained. This becomes the new focus clause and the pattern continues. If an empty clause is obtained at any stage, the refutation is successful. Otherwise, if some focus clause m obtained by resolving clause $m - 1$ with some clause j creates no new clause, the procedure *backtracks*. Then the clause $m - 1$ is relabeled as the focus clause and clause $m - 1$ is resolved against those clauses not previously tried, beginning with clause $j + 1$. The first retained resolvent is labeled $m + 1$. Clause $m + 1$ now becomes the focus clause and the process continues. To illustrate this procedure consider a simple domain from the blocks world shown in Figure 2.2. This domain consists of five blocks a , b , c , d , and e arranged in two stacks. The first stack has blocks a , b , and c , where block b is placed on c and a is on b . In the second stack block d is placed on block e .

The blocks world translates into the axioms shown in Figure 2.3(a). To prove something interesting, we add the axioms of Figure 2.3(b) which reflect some possible relationships between the blocks. For example, the axiom (iv) defines predicate *above* that is true for any two blocks X and Y if X is above Y in a stack of blocks.

- i)* $on(a, b)$
- ii)* $on(b, c)$
- iii)* $on(d, e)$

(a) Axioms From The Blocks World .

- iv)* $\forall X, Y \neg on(X, Y) \vee above(X, Y)$
- v)* $\forall X, Y, Z \neg on(X, Y) \vee above(Y, Z) \vee above(X, Z)$

(b) Axioms Defining Properties Of Blocks World.

Figure 2.3. Axioms Representing Blocks World

We now use the resolution-refutation proof procedure to answer the question “is a above c ?” Since resolution is a refutation procedure, we try to prove that the clause $above(a, c)$ is a valid clause of the input set by proving that when the set of clauses of Figure 2.3 are combined with the negation of $above(a, c)$, that is $\neg above(a, c)$, they derive an empty clause.

Figure 2.4 shows this proof. In the first step we resolve the goal $\neg above(a, c)$ with axiom (v) to get the resolvent ($\neg on(a, Y) \vee \neg above(Y, c)$). We then resolve this resolvent with axiom (i) to derive $\neg above(b, c)$. In the third step, we resolve $\neg above(b, c)$ with axiom (iv) to derive $\neg on(b, c)$. Finally, we use axiom (ii) to derive an empty set. Since the resolution yields an empty clause, we deduce that the negation of the goal $above(a, c)$ is invalid. Thus, by contradiction, we can say that the goal statement is a valid clause of the blocks world.

In addition to proving a goal, the resolution-refutation proof procedure also generates additional information. For example, as resolvents are generated, the variables in the goal statement are instantiated in a way that reflects the reasoning process. Thus, in the process of proving a goal, a resolution proof procedure also constructs (via a sequence of unifications) terms which express more information. To illustrate how this works, we use the set of clauses from Figure 2.3 to find a block which is

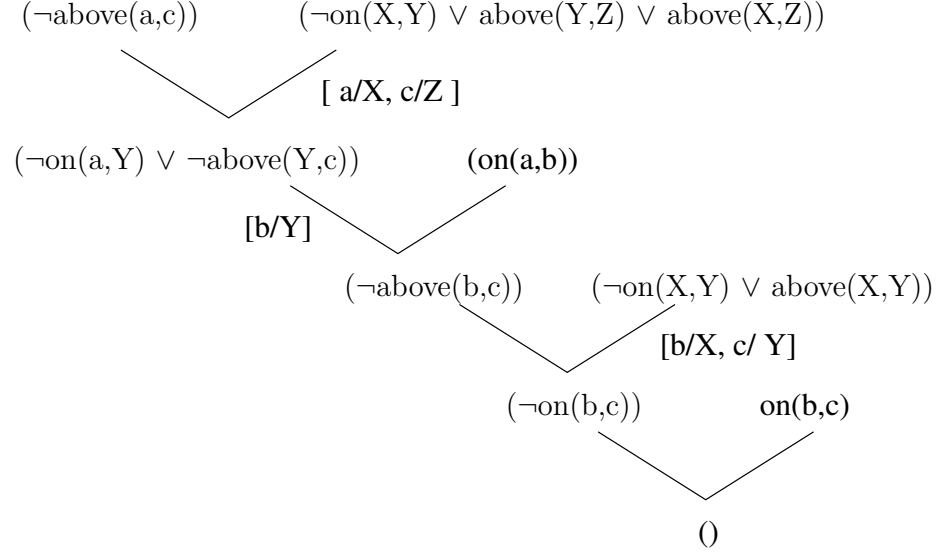


Figure 2.4: A resolution-refutation proof that proves $\text{above}(a, c)$ follows from the facts of Figure 2.3. Each step in the proof derives a new fact by resolving two complementary unifiable literals from existing facts. Variable substitutions for each step are shown enclosed in brackets.

above block c . To do this, we prove that $\exists W \text{above}(W, c)$ is a consequence of our input clauses. Similar to Figure 2.4, its derivation begins by negating $\text{above}(W, c)$. The derivation, shown in Figure 2.5, results in an invalid set when W binds to a . Thus, by contradiction, we understand that there must be a block above c and the binding of variable W to a shows that a is in fact such a block.

2.1.4 Resolution Strategies

Deriving proofs by resolution-refutation generates a large number of intermediate resolvents. Typically, most of these resolvents are redundant since they do not help to prove the goal. Generating such resolvents utilizes lot of computational resources, thereby affecting the performance of the proof procedure (Shinghal 1992). Resolution-based reasoning systems commonly employ various strategies that reduce unnecessary deductions. One such strategy removes redundant clauses as soon as they appear in a derivation. Another strategy is to remove specific clauses in the

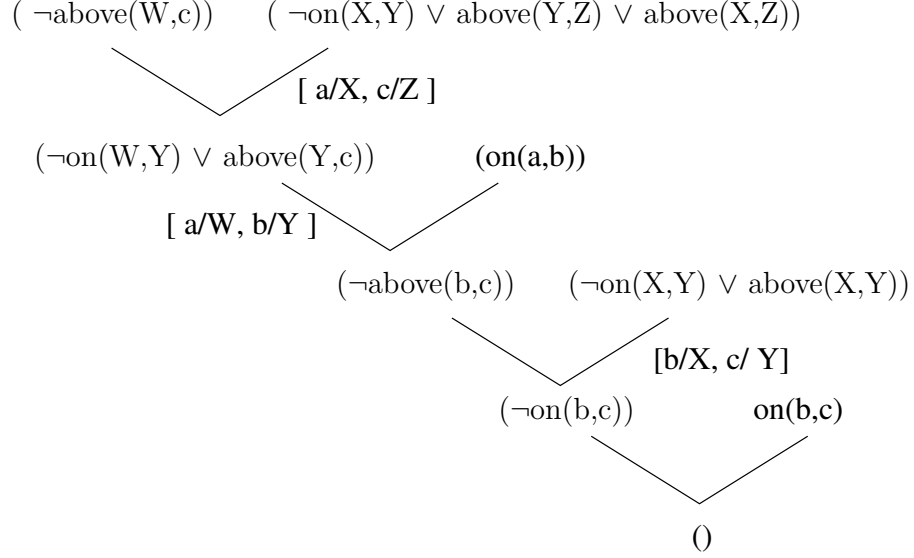


Figure 2.5: A resolution-refutation proof that proves $\exists W \text{above}(W, c)$ is a consequence of the facts of Figure 2.3. Derivation of the empty set and its resulting binding of variable W to a shows that there exists a block above c and a is that block.

presence of more general ones by a process known as subsumption (Robinson 1965b). However, unrestricted subsumption does not preserve the completeness of resolution.

Instead of removing redundant clauses, some strategies prevent their generation in the first place. The set-of-support strategy is one of the most powerful strategies of this kind (Wos, Robinson, and Carson 1965). Hyper-resolution is another strategy that reduces the number of intermediate resolvents by combining several resolution steps into a single inference step (Robinson 1983). Linear resolution, introduced by Loveland, is a strategy that restricts the kinds of derivations that are constructed and thus is a *restriction* on resolution (Loveland 1970, Luckham 1970). Similar to resolution-refutation, linear resolution starts by considering the negation of the goal clause as the focus clause. At each step the focus clause is resolved with another clause which is either an input clause or an ancestor of the focus clause, and the new resolvent becomes the focus clause. Such refutations are called *linear* because they consist of a single linear sequence of steps from the goal to the empty clause.

With the exception of unrestricted subsumption, all the strategies mentioned

so far preserve completeness. However, certain strategies are designed to compromise completeness for efficiency. *Unit resolution* and *input resolution* are two such refinements of linear resolution (Loveland 1968). In the former, one of the resolved clauses is always a single literal; in the latter, one of the resolved clauses is always selected from the original set to be refuted. Neither strategy is complete. Another strategy, called *ordered resolution*, imposes a form of partial ordering on the predicate symbols, terms, literals, or clauses occurring in the deduction. It treats clauses not as sets of literals but as sequences of linearly ordered literals. Ordered resolution is extremely efficient but, like unit and input resolution, is not complete.

Most resolution-based reasoning systems adopt one of these strategies or combine some of them to improve the efficiency of the proof procedure. Some strategies improve certain aspects of the deduction process at the expense of others. The choice of a strategy is often subject to the nature of the problem domain or the structure of the clauses in the theory. Next, we present a brief overview of the linear resolution strategy based on which we designed our reasoning system.

2.1.5 Model Elimination

The Model Elimination (ME) procedure was defined by Donald Loveland in 1968. The first major implementation of ME was completed by Fleisig et al (Loveland *et al.* 1974). Since then many reasoning systems have used ME as their underlying inference mechanism. A more detailed description of ME can be found in (Loveland 1968).

Like resolution, ME is also a refutation procedure that proves the validity of a logical formula by contradiction. It is a linear input procedure which can be viewed as a restriction of resolution, requiring that one of the two clauses that generates a resolvent be the clause most recently generated, and the other be a clause which is already present in the proof tree, such as an ancestor or a clause from the input clause

set. ME differs from resolution in its representation of clauses. In resolution a clause is a set of literals and a logic formula is a set of clauses. In ME a clause is a chain, an ordered list of literals. A formula consisting of n input clauses is represented by at least n *input chains*. Literals in a chain are either B-literals or A-literals. An A-literal is one that has been used in a derivation and may participate later in the derivation. Initially all literals in the input chains are classified as B-literals. An empty chain denotes an empty clause. Any processing on a chain during ME derivation occurs only on the leftmost literal of the chain.

We now discuss some terms related to chains necessary to understand the ME derivation. A chain is *admissible* if and only if it is preadmissible and the leftmost literal in the chain is a B-literal. If l is the leftmost literal in a chain C , then $C - \{l\}$ is the chain C with the leftmost occurrence of l removed. A chain is *preadmissible* if and only if:

- (1) complementary literals are separated by an A-literal.
- (2) no B-literal is to the left of an identical A-literal
- (3) no A-literal is identical or complementary to another A-literal.

ME defines two basic operations, *extension* and *reduction*, for deriving conclusions. *Extension* is like resolution, which joins an input chain to the left of the chain under consideration if the leftmost B-literal of the chain unifies with the complement of some literal of the input chain. The new chain is the instantiation of the current chain by the unifier with the unifying literal of the input chain dropped, and the other unifying literal promoted to A-literal. The instantiated literals added to the chain are classified as B-literals. For example, let C_1 be an admissible chain and let C_2 be an input chain. If there exists a MGU σ of the leftmost literal l_1 of C_1 and the complement of any literal l_2 of C_2 , then the extension operation extends C_1 by C_2 to form chain C_3 by promoting the leftmost literal of $C_1\sigma$ to an A-literal and placing $(C_2 - l_2)\sigma$ to

- 1) $p(X) \vee \neg q(Y)$
- 2) $\neg p(W) \vee r(U)$
- 3) $\neg p(a) \vee \neg r(T)$
- Goal: $p(a) \vee q(Z)$

Figure 2.6. Model Elimination Proof Sample Clauses

the left. Thus, all literals in C_3 retain their type except the complementary unifiable literal which is changed to an A-literal. If the resulting chain is not preadmissible then the extension is considered invalid. Otherwise, if the chain is a preadmissible chain with an A-literal leftmost (i.e., nonadmissible) then the leftmost A-literals are removed back to the leftmost B-literals, which then yields an admissible chain. The *reduction* operation removes the leftmost B-literal of a chain under consideration if it can be unified with the complement of an A-literal (its ancestor) of the chain. The new chain is the instantiation of the current chain by the unifier with the leftmost B-literal missing. Again, all A-literals to the left of the first B-literal are removed.

To illustrate the ME proof procedure, consider the sample set of clauses adapted from (Loveland 1968). Figure 2.7 shows the proof tree for this example to prove the goal $p(a) \vee q(Z)$. In the first step, the proof procedure performs the extension operation by unifying the leftmost B-literal $q(Z)$, from the initial chain $q(Z) \vee p(a)$, with $\neg q(Y)$ from the first clause. This operation joins the shortened input clause to the left of the chain and promotes the unifying literal $q(Z)$ from B-literal to A-literal resulting in a modified chain which is the modified goal. We denote A-literals by enclosing them in $[]$ brackets. In the second step, it performs an extension by unifying the leftmost B-literal $p(X)$ with $\neg p(W)$ from the second axiom. This results in a new chain $r(U) \vee [p(X)] \vee [q(Z)] \vee p(a)$. Next, the procedure performs an extension operation to unify the literal $r(U)$ with $\neg r(T)$ from the third axiom to derive a new chain $\neg p(a) \vee [r(U)] \vee [p(X)] \vee [q(Z)] \vee p(a)$. Note that each of the first three derivation steps results in a chain such that the leftmost literal is a B-literal. Observing that

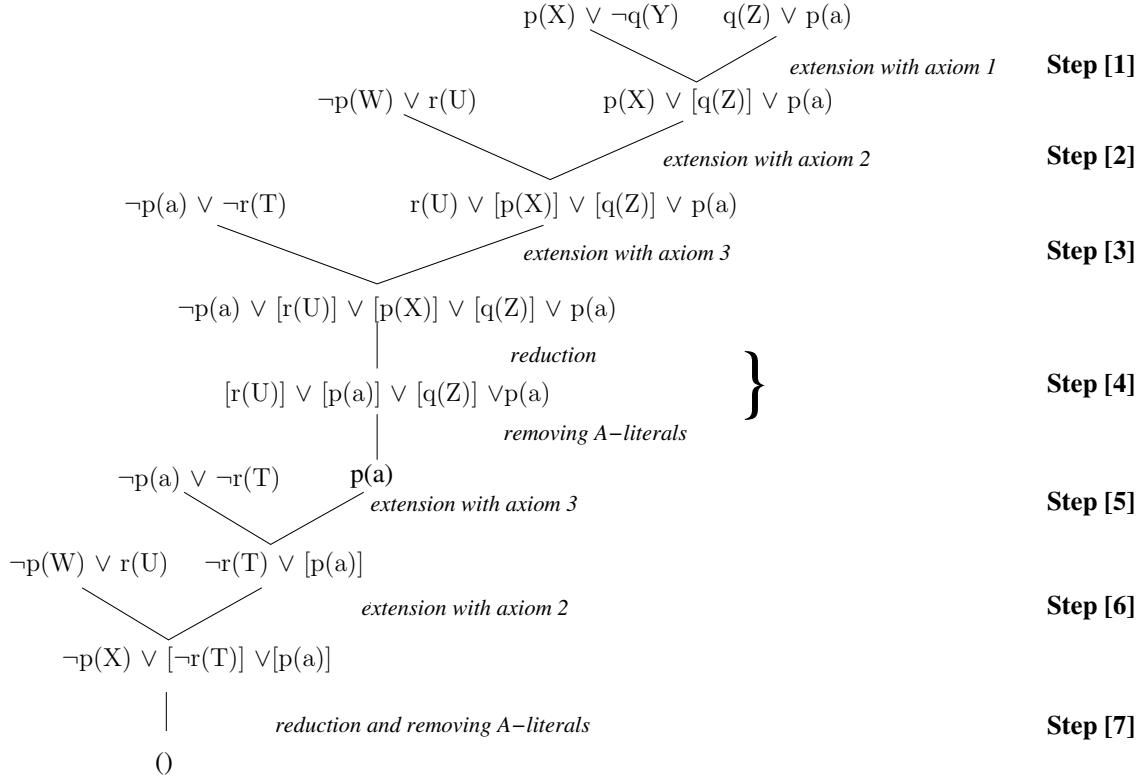


Figure 2.7: A ME proof that proves the goal $\neg q(Y)$ follows from the facts of Figure 2.6. A-literals are enclosed in square brackets. All other literals are B-literals. Each extension step attempts an unification with complementary literals from an input chain and the designated chain. Each reduction step attempts an unification of a B-literal and its complementary ancestor A-literal.

the chain obtained after the third step contains a B-literal $\neg p(a)$ that can be unified with the complement of an A-literal $[p(X)]$ generated in step 2, the fourth step of the proof procedure performs a reduction operation. This reduction operation results in a chain where all leftmost literals are A-literals reducing the chain to $p(a)$. In the fifth step, the proof procedure unifies the literal $p(a)$ with the literal $\neg p(a)$ from the third clause to derive a new chain by performing an extension. Again it performs an extension operation by unifying the literal $\neg r(T)$ from the chain with the literal $r(T)$ from the second clause. Finally, it performs a reduction operation on the chain derived in the sixth step and eliminates all the A-literals to derive an empty clause.

2.1.6 Logic Programming Paradigm

While implementing a theorem prover based on Model Elimination, it is possible to take advantage of existing systems. One such system is Prolog, by far the most popular logic programming language used as a reasoning system. Prolog is characterized by its very high efficiency.

Theorem proving in Prolog is different from resolution theorem proving, where the primary focus is on determining whether or not a given set of clauses is valid using refutability. In Prolog, the focus is on *answer extraction*, which means deriving an answer to a query made in the context of a program. This alternate viewpoint is drawn from the general observation that an answer that can be computed using a problem-specific algorithm can also be determined by a proof procedure that uses the axioms of a theory and models the problem as a theorem to be proven. Thus, Prolog programs seek a constructive proof of an existentially quantified theorem where the existential variables are instantiated so as to make the theorem true (Kowalski 1986).

Prolog's answer extraction begins with a modification of the first-order language. A logic program in Prolog is a set of formulae composed from a subclass of first-order formulae called *Horn clauses*. A Horn clause is a disjunction of literals containing at the most one positive literal. A *definite clause* is a Horn clause with a single positive literal. It has the form $A_1 \vee \neg A_2 \vee \dots \vee \neg A_n$, where A_1 is the single positive literal, also called the head literal and the remaining literals are negative literals. The *query* is a negative Horn clause composed of only negative literals. In Prolog, the problem format is of the form $P \supset Q$, where P is the *program* made up of a set of definite Horn clauses and Q is the *query* of the form $\exists \bar{x} (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n)$, where $\exists \bar{x}$ means the existential closure of all variables in the expression it quantifies.

The proof procedure of Prolog is a restriction of Model Elimination. It is structured as linear resolution for Horn clauses by eliminating the ME reduction rule. This leaves only ME-extension as an inference rule, which makes no use of A-literals.

This eliminates the need for two types of literals – A-literals and B-literals. Since there are no A-literals, ME-extension is simply the binary resolution inference rule. Thus, within the Horn clause domain, ME collapses to a linear input resolution-refutation procedure. This restriction of ME to Horn clauses is called *SLD-resolution*, for linear resolution with a selection function for definite clauses. A detailed explanation of the Prolog proof procedure can be found in (Kowalski 1986). The key statement of soundness and completeness for Horn clause logic programming is that for SLD-resolution every computed answer is correct and if a correct answer exists, then there exists a finite SLD-resolution deduction which yields a computed answer.

The efficiency of Prolog is primarily due to compilation of the input clauses into optimized inference instructions. This process consists mainly of generating special procedures for unification with the head of each clause. Prolog implementations typically use one property of SLD-resolution in particular to achieve greater efficiency – operations performed during SLD-resolution can be determined in advance depending on the structure of each clause. This information can be used for compiling the Prolog input clauses into inference instructions for a virtual or actual machine. The Warren Abstract Machine, which has an efficient realization for executing compiled instances of Prolog programs, has become the basis for most commercial and research Prolog implementations (Kaci 1991).

2.1.7 Warren Abstract Machine

There are primarily two main approaches to an efficient implementation of Prolog: interpreted and native execution. In the interpreted approach, logic programs are compiled into code for an abstract machine. The abstract machine interprets this code at runtime to execute the programs. For the second approach, programs are compiled directly into code understood by a target machine, which then directly executes the program. Native code tends to be faster and interpreted code tends

to be more portable (Aho and Ullman 1977). In 1983, David Warren proposed the design of an abstract machine called the Warren Abstract Machine (WAM) as an execution model for Prolog based on the interpreted approach (Kaci 1991). Since then WAM has become the *de facto* standard for Prolog implementation.

The WAM defines data structures for efficient representation of the elements of a logic program. It represents logic terms as tagged words, where each word contains a tag field and a value. The tag field contains the type of the term (constant, variable, list or structure). The value field is used for many purposes depending upon the type of term: it contains values of integers, address of unbound variables and compound terms like lists or structures. Unbound variables are implemented as self-referential pointers. When two variables unify, one of them is modified to point to the other. Therefore, during derivation it may be necessary to follow a chain of pointers to access a variable's values.

The WAM defines a stack-based model to execute Prolog programs. It defines four separate stacks – two for storing data-objects, one stack to support unification and one stack to support the interaction of backtracking and unification. It also defines two logical areas: one area acts as the code space and the other as a symbol table. The WAM also defines a simple yet powerful instruction set designed specifically for executing logic programs. For example, it has instructions like *get*, *put*, and *unify* to perform unification. For backtracking, it has *try*, *retry*, and *trust* instructions and for sequential control it has *call*, *return* and *jump* instructions. The data-structures, four stacks, and simple instruction set together form the core of WAM. Warren designed each element of the core to make use of as little memory as possible. The abstract machine further classifies its available memory into registers, short-term memory, long-term memory, and permanent memory for added efficiency. The execution model is optimized to make efficient use of each class of memory. The simple design and specialized optimizations make WAM an extremely memory-efficient ma-

chine. A detailed design of the architecture of WAM and its operation can be found in (Kaci 1991).

The Prolog proof procedure is complete only for definite clauses and not for full first-order logic. Since our reasoning system needs to be complete for full first-order logic, it does not make direct use of WAM or Prolog. Instead, we borrow various concepts underlying the design of a Prolog compiler for WAM and the similarities between the architecture of Warren Abstract Machine and the Java Virtual Machine to design a high-performance reasoning system in Java, complete for full first-order logic.

2.2 *First-Order Logic Theorem Provers*

In this section, we describe some of the first-order logic theorem provers which closely resemble the system we designed in methodology and implementation. These provers implement Model Elimination or its variant as their main inference procedure.

2.2.1 *Prolog Technology Theorem Provers*

The inference procedure of Prolog achieves high efficiency at the expense of certain restrictions on the input language and incompleteness for full first-order logic. However, it is possible to extend the Prolog technology to develop an efficient inference mechanism for full first-order logic. We summarize some existing theorem provers that extend the Prolog technology in this way.

2.2.1.1 *Prolog technology theorem prover.* The Prolog technology theorem prover (PTTP) is an extension to Prolog which is complete for full first-order predicate calculus (Stickel 1986). It differs from Prolog in its use of unification with the occurs check for soundness, iterative-deepening search instead of unbounded depth-first search to make the search strategy complete, and the addition of Model Elimination reduction rule to Prolog inferences which makes the inference system complete.

PTTP was shown to achieve a very high inference rate and demonstrated that for some problems it was an alternative to other resolution-based implementations. The PTTP system is implemented in Lisp and F-Prolog.

2.2.1.2 Prolog as an implementation language. The Model Elimination inference procedure and SLD-resolution have many similarities. Letz and Stenz (Letz and Stenz 2001) made use of these similarities to directly implement pure ME in Prolog. They developed a set of Prolog formulae written using the Prolog syntax. These formulae explicitly overcame the incompleteness of Prolog for full first-order logic. For instance, the formulae simulated the reduction operation of ME which is absent in Prolog. The formulae did not implement sound unification, instead they proposed the use of Prolog systems which provide the option for logic programming with sound unification. In this manner, they combined the advantages of Prolog, its high efficiency and execution of compiled formulae, with the completeness of ME to perform logical reasoning in full first-order logic using Prolog. However, this approach has one drawback - due to the limitations in Prolog, it is difficult to implement the advanced pruning strategies typically employed by most Model Elimination-based theorem provers.

2.2.1.3 Setheo and Partheo. Setheo (SEquential THEOrem prover) is a fully automatic theorem prover designed to prove the unsatisfiability of formulae in first-order logic (Ibens 1997). It is based on *connection tableaux* calculus, which is an integration of tableaux calculus, Model Elimination, and the connection method. PARTHEO is the parallel version of Setheo. It is another sound and complete or-parallel theorem prover for first-order logic (Schumann and Letz 1990). Partheo consists of a uniform network of sequential theorem provers communicating via message passing. Each sequential prover is implemented as an extension of Warren's abstract machine. Partheo is written in C and runs on a network of 16 transputers. It achieves

higher inference rates with its use of parallelism. It performs a distributed search to simultaneously explore the search space on multiple processors.

2.2.1.4 *Meteor*. Meteor is another Model Elimination theorem prover which can execute in parallel and distributed modes (Astrachan. 1992). In addition to its implementation of pure Model Elimination, Meteor uses a specialized search strategy called *caching and lemmaizing* to search for proofs which otherwise are not easily found using a depth-first iterative-deepening search strategy. Caching, as implemented in Meteor, refers to a mechanism that optionally replaces the normal search mechanism with a cache that has a lower computational cost, but yields identical results to search. The objective of caching is to make effective use of results discovered by past searches. Lemmaizing refers to using solutions or lemmas for proofs which have already been proved, in order to shorten the presentation and development of other proofs. Meteor stores certain previously proved solutions and uses these solutions to augment a normal search mechanism in cases where the derivation may require many inference steps. Caching is applicable only to theorems expressible as a Horn clause set. However, lemmaizing can be used with both Horn and non-Horn theorems. Caching and lemmaizing have enabled Meteor to prove theorems previously unobtainable by top-down model elimination theorem provers. A more complete explanation of caching and lemmaizing can be found in (Astrachan. 1992).

2.2.1.5 *Protein*. A PROver with a Theory Extension INTERface (PROTEIN) is a Prolog technology based first-order theorem prover over built-in theories (Baumgartner and Furbach 1994). Along with the various refinements implemented in PTTP, Protein supports a modification of the Model Elimination inference procedure which does not require generation of contrapositives of the clauses.

2.2.2 *Theorem Provers in Java*

Automated theorem provers have been extensively researched for the past five decades. With the advent of newer technologies and increasing computing power, new reasoning strategies are being developed and modifications being made to existing systems to exploit the features provided by newer technologies. One such technology, the Java programming language, is becoming increasingly popular as a programming language for scientific and engineering applications in research as well as industry. Attempts are being made to develop inference systems in Java. We describe two such systems developed to exploit various features provided by the Java language to perform logical reasoning.

2.2.2.1 *NetProlog.* NetProlog is an implementation of Prolog in Java (Carvalho, Pereira, and Julia 1999) . It is a logic programming system that generates binary code executable on a Java Virtual Machine (JVM). For each logic predicate, it generates a corresponding Java class which can be used in the same way as a regular class generated for the JVM.

In order to generate a Java classes from an input clause set, the NetProlog system uses an automatic data-flow inference procedure. This procedure, analyzes the structure of the given clause set, performs error checking and identifies the data associated with each predicate. For each predicate it then generates a corresponding Java class. These clauses are then executed in a manner similar to Prolog's inference procedure to derive proofs.

2.2.2.2 *Java Theorem Prover.* Java Theorem Prover (JTP) is a Java-based object-oriented modular architecture for hybrid reasoning (Fikes, Jenkins, and Frank 2003). It defines a library of general-purpose reasoning system components to support rapid development of reasoners and reasoning systems. The JTP is currently one of the few first-order reasoning systems implemented in Java.

The reasoning system in the JTP architecture consists of modules called reasoners. These reasoners are classified into two types depending upon their ways to process queries: Backward-chaining reasoners and forward-chaining reasoners. Each reasoner has two methods: *acceptable method* and *process method*. The acceptable method decides if the goal is suitable for being processed by the reasoner, while the process method implements the actual reasoning process by attempting to find the proof for the goal. Each process method performs reasoning for the goal or the subgoal based on Model Elimination inference strategy proposed by Loveland.

CHAPTER THREE

Logical Inference In Java

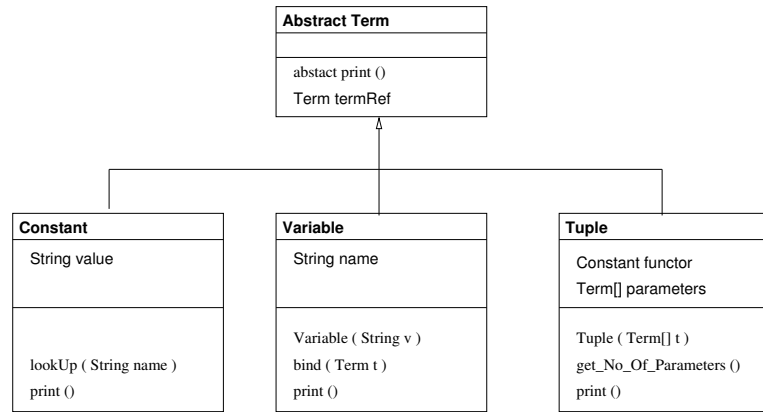
Our reasoning system is a ME-based theorem prover for full first-order logic. It is developed over several versions, with enhancements applied to each version. The first version is called the Java Logic Interpreter. It is an elementary reasoning system designed with the goal of developing an efficient representation of logical formulae in Java. The second system, the Logic-to-Java Compiler, has a logic compiler and is the enhanced version of the Java Logic Interpreter. This chapter begins with a discussion of the framework that serves as the foundation for these two reasoning systems. This is followed by a detailed description of the Java Logic Interpreter and the Logic-to-Java Compiler.

3.1 System Architecture

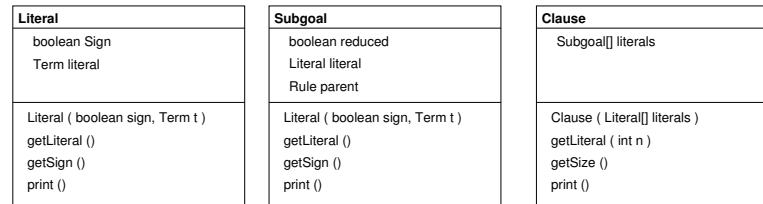
The ME inference procedure is a set of two operations, extension and reduction. The framework that serves as the foundation for our two systems facilitates the efficient execution of these two operations. This framework is made up of several components consisting of a Java representation of logical formulae, Java implementations of generic unification, ME extension and reduction, and a search strategy that ensures the ME proof procedure is complete.

3.1.1 Java Representation of Logic Formulae

First-order logic has several different types of constructs such as constants, variables, functions, and predicates. Each type of construct has certain properties and exhibits certain behavior. Our framework represents each type of construct with a separate Java class. Each Java class is a blueprint that defines the properties and methods common to all objects of a certain type of construct. For example, the



(a) The abstract class **Term** represents any term. The classes **Constant**, **Variable** and **Tuple** represent constants, variable, and function and predicates respectively.



(b) Java classes **Literal**, **Subgoal** and **Clause** representing literals, subgoals, and clauses

Figure 3.1. Java Class Representation of First-Order Logic Constructs

Constant class represents first-order logic constants. The following convention is used to name the classes: the class name is the same as the first-order logic construct with the first letter capitalized. The hierarchy of the Java classes mirrors the relation that exists between first-order logic constructs. Figure 3.1(a) shows this class hierarchy.

At the top of the hierarchy is the abstract class **Term**. **Constant** extends **Term** and contains a data member that stores the value of the constant symbol. For example, to represent constants *homer* and *lisa*, we create two instances of **Constant**, which store *homer* and *lisa* respectively as their data members. The **Constant** class is such that it creates exactly one instance for each unique constant symbol in the input clause set. This property facilitates an easy comparison of constant symbols without

having to retrieve and compare the actual values they represent. Accordingly, two constants are considered identical if they are the same instance of **Constant**.

Variable, which also extends **Term**, has one data member that stores the name of the variable. Our system represents all occurrences of a particular variable in a single clause by exactly one instance of **Variable**. For example, in the term $p(X, s(s(X)))$, only one instance of **Variable** is created to represent the two occurrences of variable X . As a result, during the derivation if one occurrence of X binds to a constant *homer*, then the other occurrence of X in the clause also reflects this binding. On the other hand, a variable with the same name occurring in two different clauses is represented by two different instances of the **Variable** class.

The class **Tuple** represents functions and predicates in first-order logic. **Tuple**, which is also an extension of **Term**, has two data members: one representing the functor and the other an array of **Term** instances representing the parameters of the function symbol or the predicate. For example, to represent term $likes(X, donuts)$, a **Tuple** instance is created with *likes* as its functor and the **Variable** X and **Constant** *donuts* as its parameters.

The **Term** class contains a data member called *termRef*. Hence, the three classes **Constant**, **Variable**, and **Tuple** that inherit from **Term** also contain this data member. When an instance of **Constant**, **Variable**, or **Tuple** is created, the *termRef* data member in the instance refers to the **Constant**, **Variable** or **Tuple** instance respectively. In case of a **Constant** and a **Tuple**, the value of *termRef* remains unchanged. However, in case of a **Variable** instance, its *termRef* refers to the **Term** that the variable binds to during unification.

Dereferencing is the process of identifying the value of the *termRef* data member of a **Term**. Dereferencing a **Constant** and **Tuple** instance yield the respective instances themselves. On the other hand, dereferencing a variable helps to identify whether it is free or bound. If the *termRef* of a **Variable** instance refers to the

Variable instance itself, then the variable is free. Otherwise, the variable is bound to the **Term** referenced by *termRef*. During a derivation, it is sometimes necessary to traverse a chain of *termRef* references to dereference a variable. For example, at a stage in the derivation, if a variable X is bound to another variable Y and Y is bound to variable Z and Z is bound to a constant *homer*, then dereferencing X entails traversing the chain of *termRef* references from the *termRef* of X to the *termRef* of Y to the *termRef* of Z and finally terminating to obtain *homer*. The **Literal** class, shown in Figure 3.1(b) is designed to represent literals. It has two data members: the sign of the literal and a **Term** instance to represent the literal term.

3.1.2 Generic Unification

Unification is a key component of the two ME operations, extension and reduction. During extension, a literal term is unified with a complementary literal term from the input clause set. During reduction, a literal term under consideration is unified with one of its complementary ancestor terms.

While performing the two ME operations, it is necessary to avoid situations where an unification between two terms builds an infinite structure. For example, consider the unification between the terms $data(Y, Y)$ and $data(X, name(X))$. In the first step, we unify variables X and Y . Next, we unify Y with $name(X)$, which results in an attempt to unify X with $name(X)$. This unification yields an attempt to unify $name(X)$ with $name(name(X))$ and so on. Such an unification, where a variable binds to a term containing that variable, results in a cyclic structure. The most common way in which this error might manifest itself is when the system tries to print out the binding for Y . This usually results in an attempt to print an infinite term $name(name(name(name(name(...$. To avoid this circularity, it is necessary that when attempting to unify a variable with a term, their unification should fail if the variable occurs in that term. The unification algorithm we implement performs the *occurs*

```

occurrenceCheck( Variable var, Term term ){
    while ( term.termRef != term )           // Dereference the variable
        term = term.termRef

    stack.push ( term )                      // Push the term under consideration on a stack

    term = stack.pop ()

    while ( term ) {
        if ( term == var )                  // If Variable var occurs in Term term then return true
            return true

        if ( term instanceof Tuple )
            for ( i = 1 to no of parameters ) // Perform occurs check for each parameter of Term
                stack.add ( tuple.body(i) )

        term = stack.pop ()                 // Pop the stack for next Term
    }

    return false                            // Variable var does not occur in Term term. Return failure.
}

```

Figure 3.2. Pseudocode to determine whether or not a variable occurs in a term

check when attempting to unify an unbound variable and a term. The algorithm to perform this check is shown in Figure 3.2. It takes as input a **Variable** instance *var* and a **Term** instance *term*. The result of the algorithm indicates whether the variable occurs in the term or not. The algorithm begins by dereferencing *term*. If the dereferenced *term* is an instance of a **Constant** then *var* does not occur in *term*. If the dereferenced *term* is an instance of a **Variable**, then *var* occurs in *term* if *var* and *term* are the same instance of **Variable**. Otherwise, if the dereferenced *term* is a **Tuple**, then the algorithm checks if *var* occurs in any of the tuple's parameters.

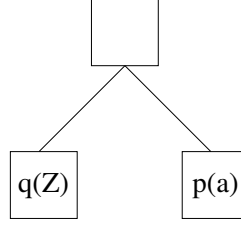
During the course of a derivation, unification is performed numerous times, as part of either the extension or the reduction operation. Our implementation of the unification algorithm performs the *occurs check* on each attempt of unifying a variable and a term. This can result in numerous invocations of the *occurs check* algorithm. Even though this could affect the performance, our algorithm implements it to guarantee completeness.

Given two **Terms** t_1 and t_2 to be unified, our Java implementation of the generic unification algorithm explained in Chapter 2 the algorithm proceeds as follows:

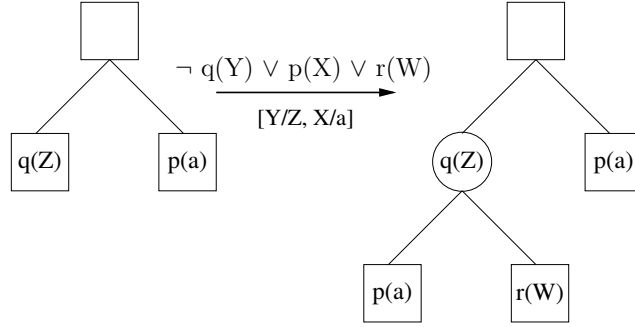
- (1) Dereference t_1 to yield term X and dereference t_2 to yield term Y .
- (2) Determine the types of term X and term Y .
- (3) If X and Y are **Constants**, then they are identical if they are each represented by the same instance of **Constant**, and their unification succeeds. Otherwise their unification fails.
- (4) If X is an unbound **Variable** and Y is a **Term**, then the algorithm binds variable X to term Y if X does not occur in Y .
- (5) If Y is an unbound **Variable** and X is a **Term**, then the algorithm binds variable Y to term X , if Y does not occur in X .
- (6) If X and Y are both **Tuples**, then the algorithm first checks if they have identical functors and the same number of parameters. If the functors differ or if they do not have the same number of parameters or both, then the unification fails. Otherwise, the algorithm checks if each i^{th} parameter from X unifies with the corresponding i^{th} parameter from Y . If this is the case, the unification succeeds.
- (7) For any other case the unification fails.

3.1.3 Efficient ME Extension

A chain in ME has a natural mapping to a tree (Letz, Schumann, Bayerl, and Bibel 1992). Each literal from a chain maps to a non-root node in the tree. For example, the chain $q(Z) \vee p(a)$, consisting of two literals, maps onto a tree as shown in Figure 3.3(a), where the two literals $p(a)$ and $q(Z)$ form the two nodes. Based on this mapping, ME extension and reduction can be modeled in terms of tree



(a) ME chain $q(Z) \vee p(a)$ modeled as a tree, where the two B-literals $q(Z)$ and $p(a)$ form the leaf nodes of the tree. The B-literals are enclosed in squares.



(b) ME extension modeled as tree expansion, where the instantiated literals $p(X)$ and $r(W)$ from an input chain form two new branches of the unifying literal. The input chain is shown above the arrow. Variables replaced and the values they replace are shown enclosed in brackets. The promoted A-literal $q(Z)$ is enclosed in a circle and B-literals are enclosed in squares.

Figure 3.3. Chain modeled as a tree and me extension as tree expansion

operations. The ME extension operation, which expands a chain by attaching the instantiated literals to the chain and promoting the unifying literal to an A-literal, can be modeled as expanding a tree. Each instantiated literal from an input chain forms a new branch of the node representing the newly promoted A-literal. For example, ME extension of the two chains $q(Z) \vee p(a)$ and $p(X) \vee r(W) \vee \neg q(Y)$ results in a new chain $p(X) \vee r(W) \vee [q(Z)] \vee p(a)$, where $[q(Z)]$ is the promoted A-literal and $p(X)$ and $r(W)$ are the instantiated literals. This operation maps onto the tree shown in the Figure 3.3(b) where the instantiated literals $p(X)$ and $r(W)$ forms two new branches of the node representing the A-literal $q(Z)$.

The ME reduction operation, which removes a B-literal if it unifies with a complement of an A-literal in the chain, can be modeled as closing a branch. A

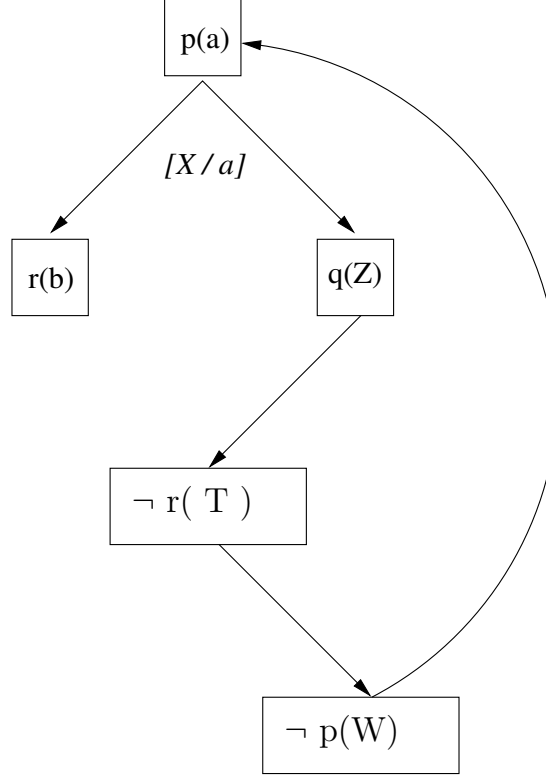


Figure 3.4: ME Reduction where the leaf node $p(W)$ unifies with its ancestor $p(a)$, modeled as closing a branch. The arrow pointing from $p(W)$ to $p(a)$ shows this ME reduction.

branch is considered to be closed if there exist two complementary unifiable nodes in the branch, one of which is a leaf node. For example, ME reduction of the chain $\neg p(a) \vee [r(U)] \vee [p(X)] \vee [q(Z)]$ causes the literal $\neg p(a)$ to unify with the complementary A-literal $[p(X)]$, resulting in the new chain $[r(U)] \vee [p(a)] \vee [q(Z)]$. This operation maps onto a tree as shown in Figure 3.4 where the arrow pointing from $p(W)$ to $p(a)$ shows the branch closed due to ME reduction. Note that ME extension can also result in the closing of a branch when a literal unifies with a unit clause from the input set.

At the start of an ME proof procedure, all the chains consist of B-literals. At each step, the proof procedure performs either ME extension or ME reduction on the leftmost B-literal of the goal chain under consideration. Extension adds new B-literals to the chain and reduction removes a pair of complementary A and B-literals from the chain. For each new B-literal attached to the chain, the proof procedure has to

perform extension or reduction to prove it. The proof procedure halts when the goal is reduced to an empty chain.

Our systems' implementation of ME is based on the tree representation of ME chains. At the start of the proof procedure, the literals from the goal form the leaf nodes of the proof tree. The proof procedure considers each leaf node in the tree as a subgoal of the original goal and attempts to prove it by employing extension or reduction. Extension of a node can result in a new set of leaf nodes, one for each instantiated literal, all having the extended node as their parent. The proof procedure treats each instantiated literal as a subgoal and proceeds to prove each subgoal. Reduction of a subgoal with a complementary ancestor on its branch closes that branch of the tree. Thus, a subgoal is considered proved if it unifies with a complementary ancestor or if it unifies with a unit clause from the input set in which case no new subgoals are generated. The proof procedure halts when it proves every subgoal in the tree.

For an efficient implementation of this tree-based ME proof procedure, our framework represents a clause by a Java class **Clause**. Since the proof procedure considers each instantiated literal from a clause as a subgoal, our framework represents each literal in a clause with a **Subgoal** instance. The **Subgoal** class consists of two data members. The first one, a **Literal** instance, represents a clause literal and the second data member, a flag, identifies whether the subgoal is proved by ME reduction. Figure 3.1(b) shows the Java class representation of the **Clause** and the **Subgoal**.

Given a set of input **Clauses** and a subgoal to prove, the ME extension implemented by our proof procedure begins by examining a **Clause** from the input clause set. It attempts to unify a **Subgoal** with a complementary **Literal** from the **Clause** with the subgoal. If the unification fails, it either goes onto the next **Subgoal** with a complementary **Literal** of that **Clause** or, if there are no more **Subgoals** in the **Clause**, it considers the next **Clause** from the input set. If the unification

succeeds, the subgoal under consideration is extended by the remaining instantiated **Subgoals** of the clause, with the **Subgoal** with the unifying **Literal** omitted. The proof procedure proceeds by attempting to prove each instantiated **Subgoal**.

3.1.4 *Efficient ME Reduction*

During the proof process, before attempting an extension, the ME proof procedure always attempts ME reduction of each subgoal. The ME reduction operation states that a subgoal can be removed if it unifies with the complement of any of its ancestors in the proof tree. In order to access a subgoal's ancestors, the proof procedure needs an effective mechanism to easily traverse a branch of proof tree. Therefore, the implementation of the **Subgoal** class is modified to add a third data member, a reference to its parent **Clause**. Figure 3.5 shows our algorithm to perform ME reduction. It is similar to traversing a linked list, where each node in the list is the ancestor **Subgoal**. At each step in the traversal, an unification between the current **Subgoal** and its complementary ancestor is attempted to determine if the subgoal can be reduced. The unique design of the **Clause** and **Subgoal** classes allows the proof procedure to efficiently traverse the ancestors of each subgoal. In order to reach the ancestor of the subgoal, our algorithm looks up the reference to the **Subgoal** that unified with the parent **Clause** of the **Subgoal** under consideration. Next, it attempts to unify these two **Subgoals**. If the unification fails, the algorithm continues traversing up the branch until it either finds a complementary unifiable ancestor or reaches the root, at which point it cannot attempt further reductions with the subgoal. At any point if the unification succeeds, the algorithm marks the current **Subgoal** as reduced and then proceeds with the proof.

```

me_reduction ( Subgoal subgoal ) {
    Rule parent
    Subgoal par_sub // Local variables

    parent = subgoal.parent // Get the parent Rule of the Subgoal under consideration

    while( parent ) {
        par_sub = parent.subgoal // Get the Subgoal the parent last unified with

        Literal par_literal = par_sub.literal // Get its literal

        Literal goal_literal = subgoal.literal // Get the literal for the subgoal

        if( par_literal.sign != goal_literal.sign ) // Verify that the parent is complementary

            if( Unify( goal_literal.term, par_literal.term ) ) // Unify the literal with the subgoal under consideration

                subgoal.reduced = true // If they successfully unify then mark the current Subgoal reduced
            else

                subgoal.reduced = false

        parent = par_sub.parent // Otherwise continue traversing up
    }
}

```

Figure 3.5: Pseudocode of ME Reduction to determine if a subgoal unifies with one of its ancestors. The unique design of the **Clause** and **Subgoal** classes makes traversing a **Subgoal**'s ancestors efficient.

3.1.5 Search Mechanism

In our discussion so far, we considered that at each inference step there is only one clause that contained a literal complementary to the subgoal literal and the unification between the subgoal and such a literal always succeeded. However, an input clause set can contain more than one clause with a literal that is complementary to the subgoal. Thus, at any point in the proof if unification between the subgoal under consideration and a clause fails, we can always choose another clause and continue with the proof. If we formulate the problem of finding a proof as a search process, then the proof of the subgoal is the solution to the search problem. This means that deriving a proof is actually a process of searching for a proof by executing a series of inference steps. As the system searches for a proof, it can run into a scenario

where the unification between the subgoal and all the available clauses fails. In such a scenario, the proof process backtracks to the parent subgoal of the subgoal under consideration. It attempts the unification of this parent subgoal with those clauses from the input set with which it has not attempted unification so far. If no such clause exists, it further backtracks to its parent subgoal and the process continues. Thus, in order to develop an automated reasoning system, we have to design the system such that it can perform a proof search by selecting valid clauses and appropriate inference operations and then executing inference steps. Such a system must also have an effective backtracking mechanism.

Although ME is a complete proof procedure in that there is always an ME derivation of an empty clause from an invalid set of input clauses, a complete search strategy must be employed to ensure that such a derivation is found. Thus, we need a search strategy that explores the entire search space of possible solutions to find a proof. The simplest search strategy that will always find a solution is breadth-first search (BFS). BFS explores the search space completely on one level before proceeding to the next level. The main drawback of BFS is that the memory required for the search grows exponentially with the search depth. Another approach is the depth-first search (DFS), whose space complexity is only linear in the search depth. However, DFS may not terminate if the search goes down an infinitely long branch. If it does find a solution at a higher depth first, it ignores a possibly better solution closer to the root of the search tree. Hence, we use a strategy called *iterative-deepening*, which combines the advantages of DFS and BFS (Korf 1985). The idea is to perform a series of depth-first searches, each with a depth-limit that is greater than the previous iteration, until a proof is found. In other words, first search the entire breadth in a depth-first manner to depth one, then search the entire breadth in a depth-first manner to depth two and so on. Iterative-deepening has minimal storage requirements, being in essence a depth-first strategy. At the same time it guarantees

that the proof will always be found.

Our reasoning system has two data structures that maintain information needed to perform iterative-deepening search. The first data structure, referred to as the *goal stack*, stores the subgoals that need to be proved. At the start of the proof, the goal stack has just one element - the goal to be proved. At each step, our proof procedure pops a subgoal off the goal stack, and performs ME extension or reduction on it. In case of a successful extension, it pushes the new subgoals onto the goal stack. After a successful reduction, the proof procedure continues by popping the next subgoal off the goal stack. The proof process terminates when there are no more subgoals on the goal stack or the search procedure reaches the upper-bound of the depth-limit. In case an ME extension or reduction fails, the search process needs to backtrack to that ancestor in the search tree whose successful extension spawned the current failed search path. The second data structure, referred to as the *trail*, facilitates this backtracking. It is implemented by the **Trail** class. During each unification, the **Trail** records information, such as all the variable bindings created for the extension or reduction, sufficient to restore the goal stack to its state prior to the unification. At any point in the proof, if the search path fails, the reasoning system uses the information stored in the **Trail** to restore the state of the goal stack. As it backtracks, the **Trail** helps to undo the variables bindings related to the failed search.

3.1.6 System Components

Our system has three primary components: a *preprocessor*, an *inference engine*, and an *output module*. The preprocessor consists of a parser that accepts as input a set of clauses representing a logic formula, analyzes each clause, and represents it in Java using the specially designed Java classes. The inference engine uses these **Clauses** to derive proofs using the two ME operations. The proof is realized as a search

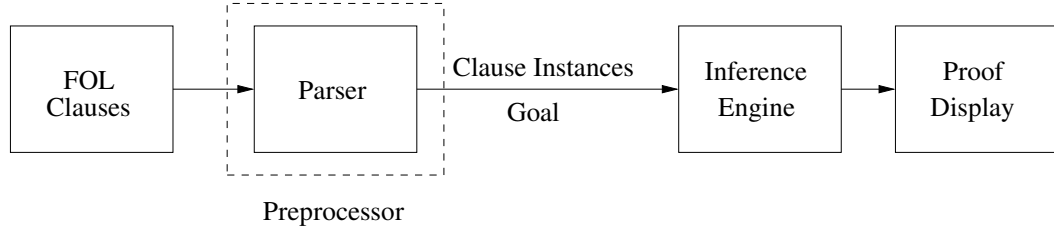


Figure 3.6. The Java Logic Interpreter System Components

procedure based on backtracking. The output module gives the user information about the result of the proof. The control flow between the three components is shown in Figure 3.6.

3.2 Java Logic Interpreter

The *Java Logic Interpreter* is a simple implementation of an ME based theorem prover. The primary aim in designing this elementary system is to build a framework suitable for developing a high-performance reasoning system in Java. For this section, we use the clauses in Figure 3.7 and the goal: $\neg \text{eats}(\text{ziggy}, \text{fish})$, which means “Does ziggy eat fish?”, to explain the features of the Java Logic Interpreter. The preprocessor, inference engine and output module together form the Java Logic Interpreter.

3.2.1 Preprocessor

The main function of the preprocessor is to generate a Java representation of the given set of input clauses. In order to generate such a representation we have a *parser* in our preprocessor. The parser takes as input the set of clauses and analyzes each clause individually. It generates the Java representation for each clause in four steps. In the first step it tokenizes each clause using logical connectives as its delimiters. This step splits the clause into its literals. In the second step, the parser tokenizes each literal into its sign and a term. In the third step, the parser classifies each

term. If the term starts with a lower-case letter and has no parameters, the parser

- i) $cat(ziggy)$
- ii) $\neg cat(X) \vee likes(X, fish)$
- iii) $\neg cat(X) \vee \neg likes(X, Y) \vee eats(X, Y)$

Figure 3.7. Clause Set To Illustrate The Java Logic Interpreter Proof Process

recognizes that it is a constant. If the term starts with a upper-case letter and has no parameters, the parser identifies it as a variable. In all other cases, the parser classifies the term as a tuple and repeats the third step to classify the tuple's functor and parameters. Once the parser has classified each term, in the fourth step it creates their Java representations using the Java classes designed for the various first-order logic constructs.

We illustrate this process for the clause $\neg cat(X) \vee likes(X, fish)$ of Figure 3.7. In the first step, the parser splits the clause into two literals, $\neg cat(X)$ and $likes(X, fish)$. In the second step, it splits literal $\neg cat(X)$ into *negative* and $cat(X)$ and $likes(X, fish)$ into *positive* and $likes(X, fish)$. In the third step, it classifies $cat(X)$ as a tuple with cat as its functor and X as its parameter, where X is a variable. Similarly, it classifies $likes(X, fish)$ as a tuple having $likes$ as its functor and $X, fish$ as its parameters, where X is a variable and $fish$ is a constant. It represents cat by an instance of **Constant**, X by an instance of a **Variable** and instantiates a **Tuple** to represent the term $cat(X)$. Next, it represents $likes$ and $fish$ by two **Constant** instances. However, it does not create a **Variable** instance for X from $likes(X, fish)$, since one was already created for the first occurrence of X in the clause. The parser then instantiates a **Tuple** to represent $likes(X, fish)$. It creates two **Literal** instances, one for $cat(X)$ with a negative sign and another for $likes(X, fish)$ with a positive sign. Finally, it instantiates a **Clause** containing the two **Literal** instances. The parser repeats this process for every clause from the input set. The collection of **Clause** instances is known as the **Theory**.

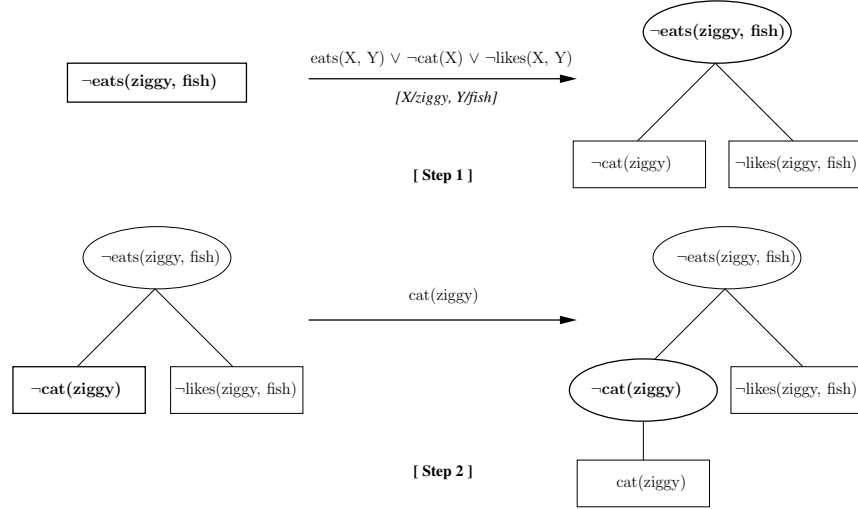


Figure 3.8: Tree representation of the first two steps taken by the Java Logic Interpreter to prove the goal $\text{eats}(\text{ziggy}, \text{fish})$. At each step, the subgoal under consideration is shown in bold. Subgoals that are proved are enclosed in ovals and all other subgoals are enclosed in boxes.

3.2.2 Inference Engine

The inference engine takes as input the **Theory** generated by the parser and the goal $\text{eats}(\text{ziggy}, \text{fish})$. It employs the ME inference procedure to refute the negated goal $\neg\text{eats}(\text{ziggy}, \text{fish})$, by contradiction.

- (1) Unifying $\neg\text{eats}(\text{ziggy}, \text{fish})$ with the literal $\text{eats}(X, Y)$ from the third clause binds X to ziggy and Y to fish . This ME extension, shown in Figure 3.8, creates two new subgoals $\neg\text{cat}(\text{ziggy})$ and $\neg\text{likes}(\text{ziggy}, \text{fish})$.
- (2) The inference engine attempts to unify the subgoal $\neg\text{cat}(\text{ziggy})$ with clause (i) from the **Theory**. The unification succeeds but this ME extension, as shown in Figure 3.8, does not generate any new **Subgoals**. Hence, the inference engine proceeds to the next subgoal $\neg\text{likes}(\text{ziggy}, \text{fish})$.
- (3) Identifying that the clause (ii) could help prove this **Subgoal**, the inference engine extends the **Subgoal** by unifying the literals $\neg\text{likes}(\text{ziggy}, \text{fish})$ and $\text{likes}(X, \text{fish})$. This generates yet another subgoal, $\neg\text{cat}(\text{ziggy})$.

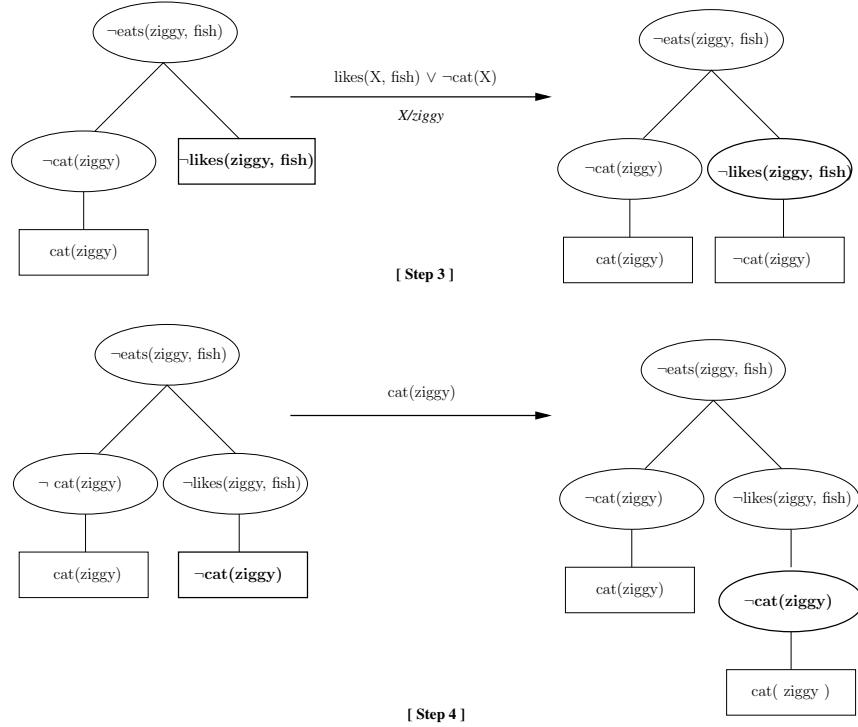


Figure 3.9: Tree representation of the steps taken by the Java Logic Interpreter to prove the goal $\text{eats}(\text{ziggy}, \text{fish})$. Proving the negation of the goal and the generated subgoals to be true reduces the negated goal to an empty clause. This proves that the negation of the goal is invalid, thus proving the goal by contradiction.

- (4) For the **Subgoal** $\neg \text{cat}(\text{ziggy})$, as shown in Figure 3.9 the inference engine repeats the process of step 2 by unifying it with **Clause** (i).
- (5) Since each subgoal is proved and there are no more **Subgoals** to prove, the inference engine determines that the proof process for the negated goal $\neg \text{eats}(\text{ziggy}, \text{fish})$ is complete. Therefore, inference engine, by contradiction, concludes the goal $\text{eats}(\text{ziggy}, \text{fish})$ to be proved.

The Java Logic Interpreter gets its name from its interpreted mode of operation. At each unification step, it interprets the structure of the two terms it attempts to unify them. This process is similar to the execution of interpreted programming languages.

3.3 Execution as Logical Inference

For each extension and reduction operation the Java Logic Interpreter performs the unification by invoking the generic unification algorithm. The input restriction of ME guarantees that for each extension operation, one of the literals is always from the input clause set. For each clause, since its structure is known even before the actual proof procedure begins, the steps the generic unification algorithm would take for a possible extension with any subgoal can be predicted in advance. The Java Logic Interpreter does not make use of this advance knowledge. Hence, we design our second reasoning system, the *Logic-to-Java Compiler*, which takes advantage of the compile-time knowledge of the structure of each clause to implement an efficient ME extension operation.

3.3.1 Mechanism for ME Extension

The ME extension operation is similar to a Prolog inference operation with sound unification. The efficiency of Prolog is primarily due to the compilation of the input clauses, which are Horn clauses, into optimized inference instructions. This process mainly consists of compiling a sequence of instructions which implement unification with the heads of the Horn clauses. We borrow this concept underlying the efficiency of the Prolog interpreter and adapt it to implement efficient ME extension for full first-order logic. The architecture of the Logic-to-Java Compiler includes a *logic compiler* which generates optimized inference instructions for each **Clause**, similar to Prolog.

The first step in adapting Prolog's technique to implement efficient ME extension, since ME is not restricted to definite clauses, is to create a representation for a clause that maintains completeness for full first-order logic. For this purpose, the logic compiler generates *contrapositives* for each clause as follows: for every literal in a clause, the logic compiler generates a *rule* with that literal as the *head*, and the rest of

the clause as the *body*. For example, given a clause: $\neg cat(X) \vee likes(X, fish)$, which can be taken to mean that all cats like fish, the logic compiler generates two contrapositives $\neg cat(X) \vee likes(X, fish)$ and $likes(X, fish) \vee \neg cat(X)$. To perform ME extension, the inference engine of the Logic-to-Java Compiler examines a rule to see if only the term in its head literal is complementary to the subgoal term. If it is, the inference engine attempts to unify the head literal with the subgoal. If the unification succeeds, the inference engine attaches the body literals of the clause, instantiated by the unifier, to the unifying literal of the subgoal. It then proceeds to prove the newly instantiated literals by considering them as subgoals. This step of attempting the unification of a subgoal with the head literal of a rule is known as *applying a rule*. If the unification fails or the term in the head literal is not complementary to the subgoal term, the inference engine moves onto the next rule.

The next step in adapting Prolog's technique for efficient ME extension involves creation of custom inference routines. For each rule in the theory, the logic compiler generates a sequence of instructions that implement ME extension with that rule. These instructions implement the unification of any subgoal with a rule's head term and in the event of a successful unification, extend the subgoal based on the depth-first search strategy.

In order to generate the customized unification instructions, the logic compiler uses the compile-time knowledge of the structure of the rule's head term. It analyzes the structure of the term to lay out a set of steps that could unify the head term with any subgoal term. It generates these steps by translating the iterative steps of the generic unification algorithm into sequential steps. For example, if the head term is $p(a, X)$, the steps it lays out are as follows: first, determine the type of the subgoal term. For a subgoal term that is a tuple, it lays out instructions to verify that the subgoal term is a tuple with functor p and two parameters. Then, for each parameter of the head term's tuple the logic compiler lays out instructions to unify

that parameter with a corresponding parameter from the subgoal term. In case the subgoal term that is a variable, the logic compiler lays out the steps to verify that the variable is free and then unify it with the head term. For all other cases, the logic compiler generates steps to return failure. Thus, for each term appearing in the head term of a rule, the logic compiler generates unification steps based on the type of the term, taking into account that the type of the corresponding subgoal term can be a constant, variable, or a tuple.

In the Java Logic Interpreter, a subgoal can unify with any one of the literals from an input clause during an extension operation. As a result, the Java Logic Interpreter cannot predict which literal from a clause will participate in an extension and which literals will be instantiated. On the other hand, the Logic-to-Java Compiler always attempts unification with the head literal of a rule. In the event of a successful unification, the new subgoals always consists of the body literals of the rule. Thus, in addition to unification steps for each rule, the logic compiler also lays out steps to extend a subgoal. These steps push the body literals, which will be instantiated by the unification, onto the goal stack in a depth-first manner.

The logic compiler represents each rule with a separate **Rule** class. Each **Rule** class has two data members, one to represent the head literal and another to represent the body literals. The logic compiler compiles the sequence of unification and extension steps it generates for each rule into Java code and embeds it into a method, called the *apply* method, within each **Rule** class. Thus, by generating a customized apply method for each rule, the Logic-to-Java Compiler performs much of the computation, such as determining the functor and the number of parameters of the head term, type of each term in the head term and determining the subgoals to push on the goal stack, once at compile time. In contrast, the generic unification algorithm of the Java Logic Interpreter performs these steps repeatedly at runtime for each attempt of ME extension.

```

procedure emitConstructor( Literal head, Literal [ ] body )
    Map varMap = new HashMap();
    write( " public Rule_n extends Rule " )
    emitHead ( head, varMap )
    emitBody ( body, varMap );

procedure emitHead ( Literal head, Map varMap )
    write ( " constructHead " )
    emitTerm ( head.term, varMap )
    write ( " head = new Literal ( "+ head.sign +", t )" )

procedure emitBody ( Literal [ ] body, Map varMap )
    write ( " constructBody " )
    for i = 0 to body.length
        emitTerm ( body [ i ], term, varMap )
        write ( "Literal literal = new Literal ( " + body[i].sign + " , t )" )
        write ( " body [ i ] = new Subgoal( literal )" )

procedure emitTerm ( Term term, Map varMap )
    // Generate code that ensures only one instanc of Constant is created
    if ( term is a Constant )
        write ( " t = new Constant.lookup ( " + term.print() + " )" )

    // When term is a Variable ensure that only one is instantiated
    if ( term is a Variable )
        boolean exists = varMap.get( term.print() ) == null
        if ( !exists )
            write ( " t = new Variable ( " + term.print() + " )" )
            varMap.put( term.print(), term.print() );
        else
            write ( " t = " + (String)varMap.get( term.print() ) )

    if ( term is Tuple )
        Tuple tuple = (Tuple)term
        emitTerm ( tuple.functor, varMap )
        write( "Term functor = t " )
        write ( "Term[] params = new Term [" + tuple.noOfParameters() + "]" )
        for ( i = 0 to tuple.noOfParameters() )
            emitTerm ( tuple.getParameter ( i ), varMap )
        write ( " params [ i ] = t " )
        write( " t = new Tuple ( functor, params )" )

```

Figure 3.10: Procedures used by the Logic-to-Java Compiler to generate Java code to build a rule in the memory

This process of generating customized extension steps for the input clause set via rules permits efficient ME extension, without sacrificing completeness. Compiling the customized extension steps into an executable Java method allows the Logic-to-Java Compiler to expedite each ME extension operation. The following section describes the compilation procedure employed by the logic compiler.

3.3.2 Compiling Rule Instances

The standard technique of generating a Java class is to first generate a Java source file and then compile it into a Java class using a standard Java compiler. The logic compiler of the Logic-to-Java Compiler also follows the same technique. Taking the **Clause** instances generated by the parser as input, the logic compiler generates a separate Java source file for each rule, containing Java code that can create an instance of the rule in the memory and the rule-specific apply method.

3.3.2.1 *Constructor.* To create an instance of a rule in the memory from a Java class, its Java source file must contain Java code to build the literals that make up the rule. Each literal is a term with a sign. Hence, to build a literal the source file must contain Java code to build the term structure of each literal. In its simplest form, a term can be a constant. To build a constant, the Java source file will have code that instantiates a **Constant**. If the term is a variable, the Java source file will have code to instantiate a **Variable**. If the term is a tuple, then in addition to code that instantiates a **Tuple**, the Java source file will also have code to instantiate the tuple's functor, and each of its parameters. We design a recursive procedure called **emitTerm** which is responsible for generating Java code to build the term in memory based on the structure of the term. For instance, if the term is a constant, **emitTerm** generates code to instantiate a **Constant**. If the term is a variable, the procedure generates code to instantiate a **Variable**. If the term is a tuple, the process is slightly different. The **emitTerm** procedure first generates code to build the tuple's functor. Since each parameter is also a term, **emitTerm** then invokes itself recursively to generate the Java code to build each parameter. Lastly, it generates code to create an instance of **Tuple**. The **emitTerm** procedure ensures that exactly one instance of **Constant** is created for each unique constant in the input clause set. It also ensures that exactly one instance of **Variable** is created for each unique variable in a clause. In this manner, the **emitTerm** method generates code to build all the terms in each rule. The constructor method of each Java source file contains the Java code to build the rule. We designed a procedure called **emitConstructor** which generates the constructor method of each rule's Java class. For the sake of simplicity, we designed two separate methods that generate code to build the rule's head and body. The **emitHead** procedure generates Java code to instantiate the head literal, and the **emitBody** procedure generates Java code to instantiate the **Subgoal** instances for the body literals. As seen in Figure 3.10, the **emitConstructor** method

```

/**
 * Rule class file to represent the rule: All cats like fish.
 * likes( X, fish )  $\vee \neg cat( X )$ 
 */

public class Rule_1 extends Rule {

    // Generate references to represent the head literal and body literal of the rule.
    Literal head;
    Subgoal [ ] body;

    Term t_1; // Local variables
    Term t_2;
    Term t_3;
    Term t_4;

    Rule_1(){
        constructHead ();
        constructBody ();
    }

    constructHead () { // Build the HEAD

        t_1 = new Constant( "likes" );
        t_2 = new Variable( "X" );
        t_3 = new Constant( "fish" );
        Term functor = t_1

        Term [ ] params = { t_2, t_3 };
        Term tuple = new Tuple ( functor, params );
        head = new Literal( true, tuple );
    }

    constructBody () { // Build the list of SUBGOALS

        t_4 = new Constant( "cat" );
        Term [ ] params = { t_2 }
        Term functor = t_4

        Term t_6 = new Tuple( functor, params );
        Literal literal = new Literal ( false, t_6 )
        body [ 0 ] = new Subgoal( literal );
    }
}

```

Figure 3.11. Code generated to build the rule $likes(X, fish) \vee \neg cat(X)$ in memory.

generates the Java code by invoking the methods `emitHead` and `emitBody`. It should be noted that for readability the procedures shown in Figures 3.10 and 3.12 show steps relevant to this report and do not include details that read and write to actual Java source files.

We illustrate the process of generating the Java code for building a rule with the help of an example. Consider the rule $likes(X, fish) \vee \neg cat(X)$. Figure 3.11 shows the Java code generated by the logic compiler for this rule. The logic compiler begins generating the code by invoking the `emitConstructor` procedure which invokes the `emitHead` procedure to generate the Java code for the head literal. The `emitHead` begins by invoking `emitTerm` to first generate code to instantiate the term $likes(X, fish)$. The `emitTerm` procedure identifies that this term is a `Tuple` and first generates code

to instantiate the functor *likes*. For each of the tuples two parameters X and *fish* **emitTerm** invokes itself. When invoked for the term X , it identifies that the term is a variable and generates the Java code to instantiate a **Variable**. When invoked the second time for the term *fish*, **emitTerm** procedure identifies that it is a constant and generates the Java code to instantiate a **Constant**. Finally, **emitHead** concludes by generating Java code to instantiate a positive **Literal**. Following **emitHead**, **emitConstructor** invokes the **emitBody** procedure to generate Java code for the body literal *cat*(X) of the rule. The **emitBody** procedure follows a similar process like **emitHead** to generate the Java code for each body literal of the rule. The **emitTerm** when invoked for the variable X from the body, does not generate code to create a new instance of **Variable** representing X , since it is the second occurrence in the clause. Instead, it simply refers to the previously created **Variable** instance for X . The **emitBody** procedure differs from **emitHead** in that it generates Java code to instantiate **Subgoals** for each body literal to facilitate the use of that literal as a subgoal during the derivation process.

3.3.2.2 Apply method. The Java class for a rule consists of a *constructor method* used to build the rule and the *apply method* to apply the rule. While generating the Java source file for a rule, the logic compiler effectively uses the available knowledge about the structure of the head literal to create the apply method for the rule. For each rule it unfolds the unification steps that the generic unification procedure would follow to unify the rule with any subgoal and incorporates them as Java code into the *apply method*.

The input to the generic unification procedure is a pair of terms whose type is not known. To unify these terms, the procedure contains instructions that take into account every combination of the types of input terms such as constant and variable, two constants and so on. The logic compiler of the Logic-to-Java Compiler

procedure emitApply (Literal head)

```

write( " public apply ( Subgoal sg ) " )
emitUnify ( head.term )

```

procedure emitUnify (Term term)

```

if ( term is a Constant )
    matchConstant ( term )

```

```

if ( term is a Variable )
    matchVariable( term )

```

```

if ( term is Tuple )
    matchTuple ( term )

```

procedure matchVariable (Term term)

```

if( first occurrence of Variable )
    write( " term.bind( goal ) " )
else
    // Unify variable after performing occurs check
    write( " if ( Unify ( term, goal ) ) " )

```

procedure matchConstant (Term term)

```

write ( " if ( goal != term ) { " )
write( " if goal is Variable " )
write( " if ( ! occurrenceCheck ( term, goal ) ) " )
write( " goal.bind( term ) " )
write( " else " )
write( " return false " )

```

procedure matchTuple (Term term)

```

write ( " if ( goal is Tuple ) " )
write( " if ( term.functor == goal.functor ) " )
for ( i = 0 to term.body.length )
    write ( " Term tbody = tuple.body [ " i " ] " )
    emitUnify( term.body [ i ] )

```

Figure 3.12: Procedures the Logic-to-Java Compiler uses to generate inference instructions based on the type and the structure of the head term known at compile-time.

is aware that the head literal of any rule is always a predicate, represented by a **Tuple** instance. Moreover, the logic compiler is also aware of the types of first-order constructs that comprise the **Tuple**. Specifically, it knows that the **Tuple**'s functor is a **Constant** and knows that each parameter is either a **Constant**, **Variable** or **Tuple**. Each of these types require a different sequence of steps to unify with the types of terms found in the subgoal. Therefore, for each type of term in the head literal, the logic compiler predicts the sequence of unification steps with any subgoal, which we summarize below.

- Constants: Consider the case where the term from the head literal is a constant C and is attempting to unify with a subgoal term T . The first step is

to dereference T . If dereferencing T yields a constant, it can unify with C if C and the constant T are identical. If dereferencing T yields a variable, then bind the variable T to the constant C . In any other case, the unification between C and term T fails.

- **Variables:** Consider the case where the term from the head literal is a variable V . In order to unify V with a term T from the subgoal, the first step is to dereference V . However, for the first occurrence of V in the head literal, it is safe to assume that V is unbound and so it does not have to be dereferenced. Also, the occurs check for V and T need not be performed and V can bind to T . For the subsequent occurrences of V in the head literal, it is necessary to perform the dereferencing as well as the occurs check before attempting to unify it with T .
- **Tuples:** Consider the case where the term from the head literal is a tuple F unifying with a subgoal term T . The first step is to dereference the subgoal term T . If dereferencing T yields a variable, then verify that the variable does not occur in F . If the variable occurs in F , the unification fails. If the variable does not occur in F , then bind the variable T to tuple F . Now, if dereferencing T yields a tuple, it is necessary to perform additional checks for the two tuples to successfully unify: whether T and F have the same functor, same number of parameters, and starting from the first parameter of F and T , verify that each parameter of F unifies with the corresponding parameter of T .

We design three procedures `matchConstant`, `matchVariable`, and `matchTuple`, shown in Figure 3.12, which implement the steps described for unifying a constant, a variable, and a tuple respectively. In addition, we create a procedure `emitUnify`, which examines the structure of each term of the rule head and accordingly invokes

one of these three methods. The logic compiler generates the apply method for a rule by invoking a procedure called **emitApply**, which takes as input the head literal and invokes **emitUnify** to generate the Java code for unification.

Figure 3.13 shows the Java code generated by the logic compiler for the head term $p(a, q(X, Y), X)$ of a rule. The rule provides an excellent example of various combinations of terms that can appear in the head term of a rule. To generate this rule's apply method, the logic compiler invokes **emitApply** with the **Literal** instance representing $p(a, q(X, Y), X)$ as its input. The **emitApply** invokes **emitUnify** with the literals' term $p(a, q(X, Y), X)$ as its input. **EmitUnify** procedure identifies this term as a **Tuple** and invokes the **matchTuple** procedure. The **matchTuple** procedure generates instructions to compare the functor symbol p with the functor of subgoal term, to compare the number of parameters of tuple head and the subgoal term, and then for each of its three parameters a , $q(X, Y)$ and X invokes **emitUnify**. The procedure **emitUnify** when invoked for the first parameter a , identifies that the parameter is a **Constant** and invokes the **matchConstant** procedure. This procedure generates instructions to unify the constant with a subgoal term accounting for the case that it could be another constant or a variable. For the second parameter $q(X, Y)$, the **emitUnify** procedure identifies it as a **Tuple** and invokes **matchTuple**. Again, **matchTuple** generates instructions to match the functor q , to compare the number of parameters, and then for the two parameters X , Y invokes **emitUnify**. The **matchVariable** procedure when invoked for each of these variables identifies that they are the first occurrences of X and Y and generates instructions to bind them to a term from the subgoal without dereferencing or performing the occurs check. Finally, for the third parameter of the head term $p(a, q(X, Y), X)$, the **emitUnify** procedure identifies that it is a variable and invokes **matchVariable**. This procedure identifies the second occurrence of variable X and generates instructions to perform the dereferencing and the occurs check first before unifying it with the third parameter of a subgoal.

```

public boolean apply ( Subgoal sg, Trail trail ) {
    Term subgoal = sg.term ;
    TrailNode point = trail.snapshot ();
    Variable var ;
    Term term;
    {
        Tuple tuple = (Tuple)subgoal ;
        // Instructions to unify 'a'
        { // Retrieve the first parameter from subgoal term
            term = tuple.body[ 0 ] ;
            //////////////////////////////// Code
            while (term.termRef != term) { // generated
                term=term.termRef; // by
            } // matchConstant
            // Match if they are identical constants // for constant
            if (( term!=termFor.a ) ) { // 'a'
                // if the constants dont match //
                // check if the subgoal is a variable //
                if (term instanceof Variable ) { //
                    // Bind the subgoal term variable to constant a //
                    var = (Variable) term ; //
                    var.bind(t_2, trail ); //
                } //
                // First parameter does not match. return failure //
                else { //
                    trail.backtrack( point ); //
                    return false; //
                } //
            } //
        } ////////////////////////////////
        // Match the tuple q(X,Y). ///////////////
        { // Retrieve the second parameter // Code
            term = tuple.body[ 1 ] ; // generate

```

```

while (term.termRef != term) {                                     // by
    term=term.termRef;                                           // matchTuple
}                                                                 // for the tuple
// Compare if the terms are identical                             // 'q(X,Y)'
if( term!= term_for_q_x_y ) {                                     //
    // if the terms are not matching then                         //
    // they can be Variables or Tuples                           //
    if (term instanceof Variable ) {                             //
        // Bind after performing checks                         //
        if ( !occurrenceCheck (term,term_for_q_x_y ) ) {       //
            var = (Variable) term ;                             //
            var.bind(term_for_q_x_y, trail );                   //
        }                                                       //
        else {                                                  //
            trail.backtrack ( point );                          //
            return false;                                       //
        }                                                       //
    } // Both terms are tuples                                  //
    else if (terminstanceof Tuple ) {                             //
        Tuple tuple_1 = (Tuple) term;                          //
        // Compare their functors                               //
        if ( tuple_1.functor != functor_q ) {                 //
            trail.backtrack ( point );                          //
            return false;                                       //
        }                                                       //
    } /////////////////////////////////////////////////////////////////// Code
    // generated
    else { // X and Y are first occurrences                      // by
        { // Bind X to subgoal term without checking           // matchVariable
            term = tuple_1.body[ 0 ] ;                          // for first 'X'
            term_X.bind(term, trail );                          //
        } ///////////////////////////////////////////////////////////////////
    } // Code generated
    { // Bind Y to subgoal term without checking               // by
        term = tuple_1.body[ 1 ] ;                             // matchVariable
    }

```

```

        term_Y.bind(term, trail );                                // for first 'Y'
    } //////////////////////////////////////////////////////////////
}                                                                    //
}                                                                    //
else { // Terms do not match. Return failure                      //
    trail.backtrack( point );                                     //
    return false;                                              //
}                                                                    //
}                                                                    //
}                                                                    //
{ // Second occurrence of X                                       //
// Unify only after performing checks.                            //
    term = tuple.body[ 2 ] ;                                    // Code generated
    if ( !unify (term, term_X, trail ) ) {                       // by
        trail.backtrack( point );                               // matchVariable
        return false;                                           // for second
    }                                                            // occurrence of
}                                                                    // 'X'
                                                                    //
// Push the subgoal to the goal stack                             // Code to push
for ( int j = 0; j < body.length; j++ )                         // instantiated
    goalstack.push( body[ j ] );                                // subgoals on
// Prepare for ME reduction.                                     // goal stack
subgoal.rule = this;                                            //
                                                                    //

return true;
}

```

Figure 3.13: Java code of the apply method for the head term $p(a, q(X, Y), X)$ of a rule generated by logic compiler using the three methods `matchConstant`, `matchVariable`, and `matchTuple`

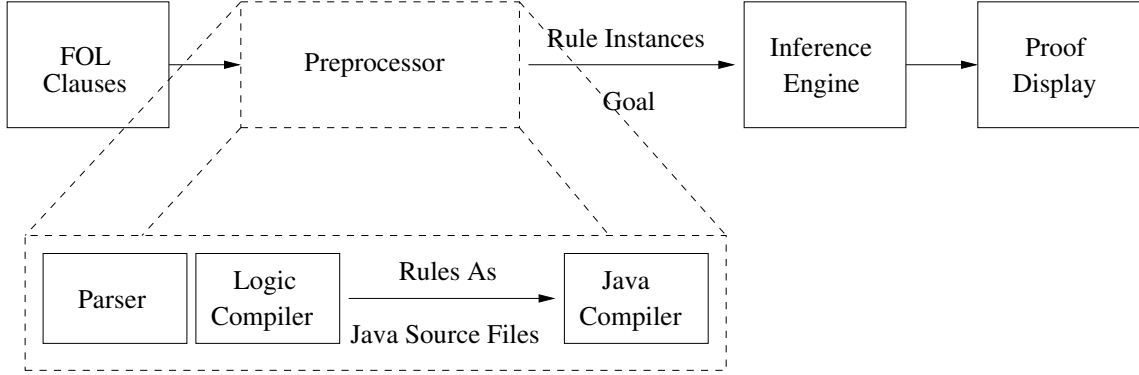


Figure 3.14. The Logic-to-Java Compiler System Components

Generating the apply method completes the process of creating a Java source file for the rule $likes(X, fish) \vee \neg cat(X)$. In this manner, the logic compiler generates Java source files for all the rules. Next, it compiles them using a standard Java compiler to generate corresponding Java class files. During the proof process, the inference engine of the Logic-to-Java Compiler applies a rule by invoking its apply method. When the inference engine invokes the apply method, the underlying Java Virtual Machine executes it to perform ME extension. Thus, creating rule-specific apply methods provides the Logic-to-Java Compiler with the ability to generate compiled machine-level inference instructions that are directly executable at runtime.

3.3.3 Logic-to-Java Compiler Architecture

The Logic-to-Java Compiler has the same underlying framework as that of the Java Logic Interpreter – it has the same parser, inference engine, and output module. It differs from the Java Logic Interpreter in two respects. The first is its modified preprocessor. The preprocessor of the Logic-to-Java Compiler has two phases: a parsing phase and a compilation phase. In the parsing phase, the preprocessor accepts as input a set of clauses and generates **Clause** instance for each clause. In the compilation phase, the logic compiler analyzes the clauses to generate *rules*. It generates a Java class for each rule containing the rule-specific apply method. The second difference

- 1) $cat(ziggy)$
- 2) $\neg cat(X) \vee likes(X, fish)$
- 3) $likes(X, fish) \vee \neg cat(X)$
- 4) $\neg cat(X) \vee \neg likes(X, Y) \vee eats(X, Y)$
- 5) $\neg likes(X, Y) \vee eats(X, Y) \vee \neg cat(X)$
- 6) $eats(X, Y) \vee \neg cat(X) \vee \neg likes(X, Y)$

Figure 3.15. Contrapositives Of A Clause Set

is the compiled, rather than interpreted nature of execution of the inference engine, which is a consequence of having rule-specific apply methods. Figure 3.14 shows the control flow between the four components of the Logic-to-Java Compiler.

3.3.3.1 Preprocessing. The input to the preprocessor of the Logic-to-Java Compiler is the set of clauses. In its parsing phase, the preprocessor analyzes these clauses and generates as output instances of **Clause**, one for each clause. In the compilation phase, the logic compiler takes the **Clause** instances as input, analyzes each clause and generates rules. Figure 3.15 shows all the contrapositives the logic compiler would generate for the clause set of Figure 3.7. Next, the logic compiler generates a Java source file containing Java code to build the rule and inference instructions to apply the rule. Lastly, the logic compiler invokes a standard Java compiler to compile the Java representation of each rule into Java class files. It then instantiates each **Rule** class to form the **Theory** which is used by the inference engine.

3.3.3.2 Proof steps. The inference engine takes the **Theory** and the goal to be proved as input. It employs the ME inference procedure to prove the goal. It operates on the negated goal in order to prove it by contradiction. At each ME extension step, during the proof process, the inference searches for a rule with a complementary predicate in the head. When it finds such a rule, the inference engine invokes the rule's apply method for the goal under consideration. The underlying JVM then

- i) $\neg q(X, Y) \vee p(X, Y)$
- ii) $\neg r(a, b) \vee \neg s(Y, c(Z)) \vee q(Y, Z)$
- iii) $\neg t(a) \vee \neg p(X, Y) \vee s(X, c(Y))$
- iv) $t(a)$
- v) $r(a, b)$
- vi) $\neg s(a, Z)$

Figure 3.16. An input clause set to illustrate identical-ancestor pruning rule.

executes the compiled instructions inside the `apply` method to either return failure or generate new subgoals.

3.4 Refinements

The main goal in designing the Java Logic Interpreter is to develop a framework to perform logical reasoning for first-order logic in Java. However, the inference procedure of the Java Logic Interpreter is inherently slow, owing to its interpreted nature. The reasoning procedure implemented by Logic-to-Java Compiler shows significant performance improvement over Java Logic Interpreter. This section presents some refinements implemented in the Logic-to-Java Compiler in order to further improve its performance without sacrificing the completeness of its proof procedure.

3.4.1 Identical-Ancestor Pruning Rule

There are several optimizations that can be applied to the search mechanism without affecting its completeness. Many of these optimizations are implicit in the original definition of ME as proposed by Loveland (Loveland 1968). The most widely used of these is the **identical-ancestor pruning rule** (IAPR).

In the course of deriving a proof for a particular subgoal S , if the proof procedure identifies that a subgoal G_2 is identical to any one of its ancestors G_1 in the proof tree, then the proof procedure discards G_2 and can initiate backtracking to proceed to the next subgoal. Discarding G_2 is justified because:

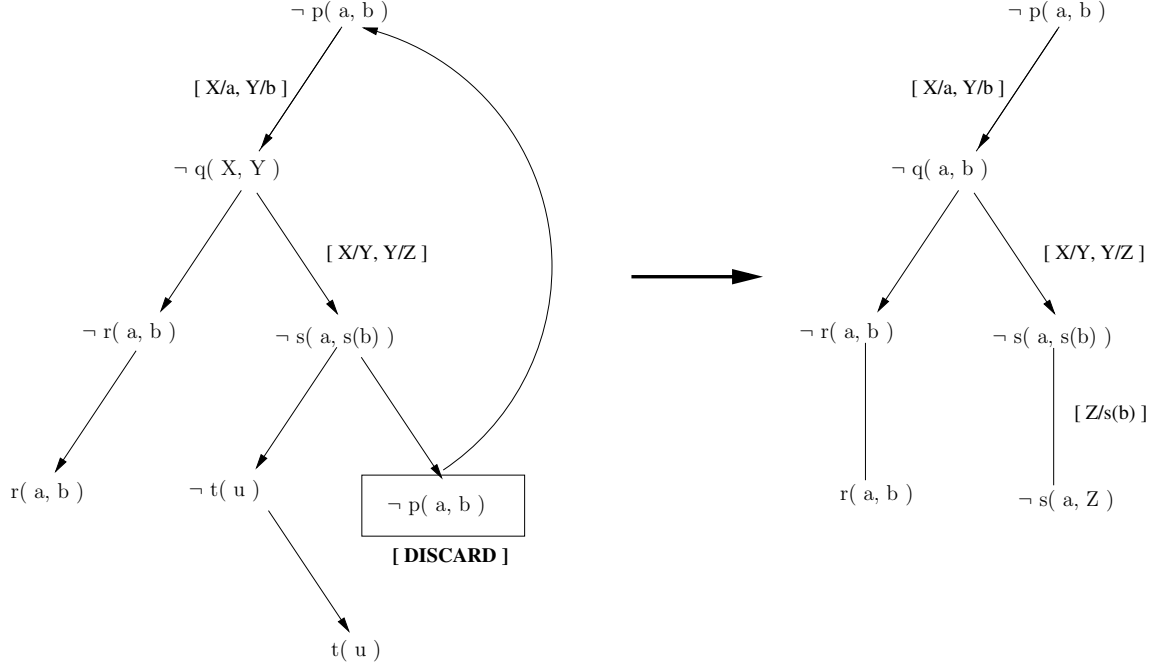


Figure 3.17: The tree representation of the steps taken by the Logic-to-Java Compiler to prove the goal $p(a, b)$. The arrow pointing from the second occurrence of $p(a, b)$ to its ancestor subgoal $p(a, b)$ indicates $\neg s(a, s(b))$ cannot be proved using rule (iii). The subgoal is eventually proved via rule (vi).

- There is no proof of G_2 in the search for the given theory.
- Whether or not there is a proof of G_1 using G_2 , there will be an equivalent proof of G_1 elsewhere in the search without using G_2 .

This technique of pruning the search space below G_2 is called *IAPR* (Astrachan and Stickel 1992). To illustrate this concept let us consider the proof of the negation of goal $p(a, b)$ using clause set shown in Figure 3.16.

Applying the first rule $\neg q(X, Y) \vee p(X, Y)$ to the subgoal $\neg p(a, b)$ produces the subgoal $\neg q(a, b)$ by binding X to a and Y to b . Next, applying $\neg r(a, b) \vee \neg s(Y, c(Z)) \vee q(Y, Z)$ to the subgoal $\neg q(a, b)$ generates two new subgoals $\neg r(a, b)$ and $\neg s(a, c(b))$. Working on the $\neg r(a, b)$ subgoal first, rule (v) of the clause set confirms that $\neg r(a, b)$ is true and we proceed to the subgoal $s(a, c(b))$. Rule (iii) could help us prove this subgoal if we bind X to a and Y to b . This step generates two new subgoals $\neg t(a)$

and $\neg p(a, b)$. Again, the rule (vi) $t(a)$ confirms that the subgoal $\neg t(a)$ is true and we can proceed to the subgoal $\neg p(a, b)$. However, this subgoal is identical to its ancestor subgoal, $\neg p(a, b)$. Hence, applying the identical-ancestor pruning rule for the second occurrence of the $\neg p(a, b)$ we initiate backtracking to discard the rule (iii) to prove $s(a, c(b))$. Another rule, (vi) can help us prove the subgoal $s(a, c(b))$. Apply rule (vi) to the subgoal binds variable Z to $c(b)$. Since there are no more subgoals to prove for the original subgoal $\neg p(a, b)$, we conclude that the goal is proved. Figure 3.17(a) shows the proof before initiating backtracking due to IAPR and Figure 3.17(b) shows the final proof.

3.4.2 Rule Indexing

At each ME extension step, the inference engine has to search for a **Rule** whose head term is complementary to the subgoal. If the **Theory** contains a large number of **Rules**, a linear search of the entire **Theory** for such a **Rule** will be inefficient. We implemented two enhancements to speed up the search for a rule with the head term that is complementary to the subgoal term.

The first is a grouping of **Rules** based on the structure of their head term. The logic compiler uses an index to perform the grouping of **Rule** instances. The key of this index consists of the sign of the head term, its functor symbol, and the number of parameters. The logic compiler groups together **Rule** instances with the same key. It organizes all the **Rule** instances into a data structure, containing one entry for each unique key in the **Theory**. At runtime, in order to find a **Rule** instance with a head term that can be unified with the subgoal term, the inference engine looks up the data structure with a key consisting of the sign opposite to that of the subgoal term, the functor symbol of the subgoal term and the same number of parameters. This lookup yields an index into the data structure, which the inference engine uses to obtain a set of **Rules**, all of which have the head terms complementary to the

subgoal term. The inference engine attempts ME extension only on this set of **Rules**, instead of performing a linear search of the entire theory.

The second enhancement is the addition of a new data member in the **Subgoal** class. This data member stores the value of its key – the sign of the literal the **Subgoal** represents, its functor symbol, and the number of parameters. The advantage of this enhancement is, to find the set of **Rules** that might unify with a given **Subgoal**. At runtime the inference engine does not have to construct the key for any **Subgoal** since it is already available inside the **Subgoal** instance. These two enhancements are collectively called *rule indexing*.

3.4.3 Rule Caching

Consider the proof of a subgoal S , using a theory consisting of rules $R_1 \dots R_n$. To prove the subgoal S , the inference engine looks up the theory for a rule that is complementary unifiable with S . Let R_1 be such a rule. The inference engine makes a copy of rule R_1 and uses it to perform an ME extension. Let this extension operation generate subgoals S_1 , S_2 , and S_3 . Let S_1 be proved using rule R_x and S_2 be proved using rule R_y . These two operations use the copies of R_x and R_y from the theory. Now consider the case where the inference engine cannot prove subgoal S_3 . In this case the inference engine back tracks and attempts to prove subgoal S with another rule. The backtracking operation leaves the inference engine with the instances of rules R_1 , R_x , and R_y . These **Rule** instances need not be discarded since they may be useful later in the proof process. The Logic-to-Java Compiler maintains a *Cache* of such **Rule** instances. The cache also implements rule indexing where, given a key, the cache returns the matching **Rule** instances. During the proof process, the inference engine first looks up the cache for the set of **Rules** with the key complementary to the **Subgoal** under consideration. It applies each of these **Rules** until one of them successfully unifies with the **Subgoal**. If none of the **Rules**

available in the cache successfully unify with the **Subgoal** or if there are no **Rules** in the cache, the inference engine looks up the **Theory** for the next available **Rule** with the complementary key. The **Theory** then creates a new instance of that **Rule** and the proof procedure continues. This caching of **Rule** instances reduces the overhead of creating and deleting unused **Rule** objects.

CHAPTER FOUR

The Java Inference Engine

This chapter presents the architecture of our third reasoning system, the Logic-to-Bytecode Compiler. It details various refinements implemented for this system.

4.1 *Compiling for Java Virtual Machine*

The Logic-to-Bytecode Compiler is characterized by a specialized logic compiler that generates Java classes without using a standard Java compiler. Designing such a compiler requires a thorough understanding of the architecture of the Java Virtual Machine (JVM). Our discussion of the JVM in this chapter is brief, but sufficient to understand the architecture of the logic compiler.

4.1.1 *Java Virtual Machine Architecture*

The *JVM* is a computer designed to load binary Java class files and execute the instructions, called *bytecodes*, that they contain. It is called virtual because it is an abstract computer defined by a specification. Various vendors provide concrete implementations of the JVM that adhere to this specification and each Java application runs inside a run-time instance of such an implementation. The JVM specification describes the behavior of a virtual machine instance in terms of subsystems, memory areas, data types, the Java class file format, and instructions (Lindholm and Yellin 1999). Figure 4.1 shows a block diagram of the JVM that includes the major subsystems and memory areas described in the specification. A runtime instance of the JVM is created when a Java application starts. Each JVM has a *class loader subsystem* which is responsible for loading classes and interfaces. A discussion of the *class loader subsystem* is beyond the scope of this report. To execute an application, the JVM organizes its memory into several *runtime data areas*. These data areas

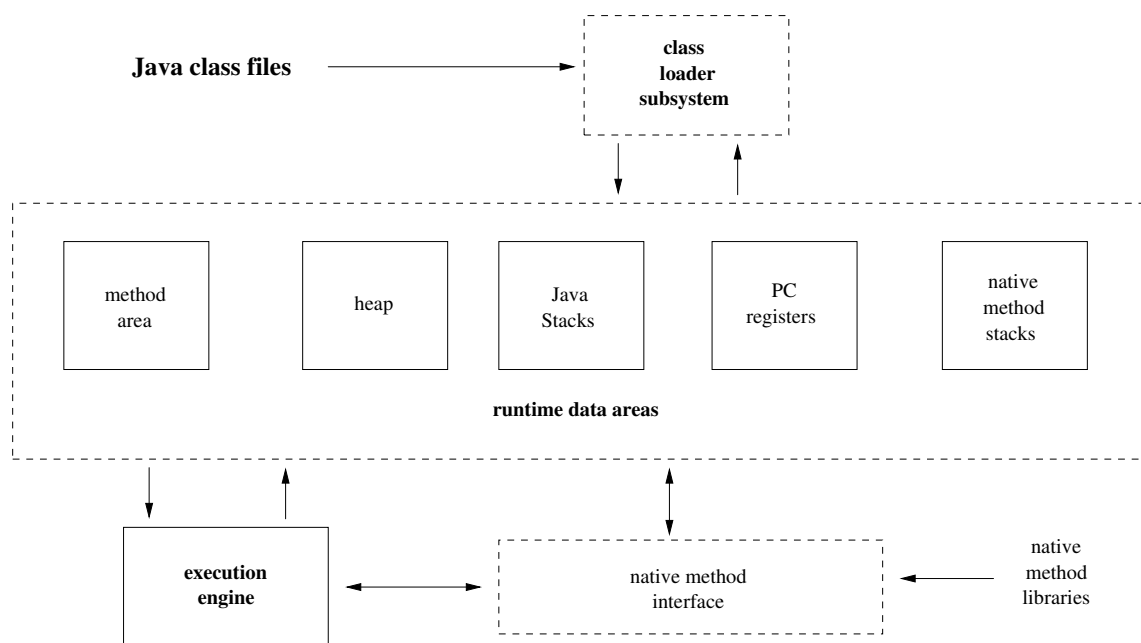


Figure 4.1: The internal architecture of the JVM. It is primarily divided into the *class loader subsystem* which loads the Java class files, the memory needed to execute a program organized into several *runtime data areas*, and the *execution engine*, a mechanism responsible for executing the instructions contained in the methods of loaded classes.

store items like instructions, information extracted from loaded class files, instantiated objects, method parameters, local variables, intermediate computation results, and return values associated with the application. Some runtime areas are unique to individual threads in the JVM, while others are shared among all the threads of an application. The *method area* and *heap area* are two runtime data areas that all threads of an application share. When the JVM loads a class file, it uses the method area to store the binary data contained in the class. The JVM uses the heap area to place all objects instantiated as it executes an application.

When an application starts executing, its initial class starts a thread inside the JVM. Each new thread receives its own *Program Counter (PC)* register and *Java stack*. The value of the PC register indicates the next instruction to be executed. The Java stack is composed of *stack frames*. Each frame stores the state of one method invocation such as its local variables, parameters with which the method was

invoked, and its return values. When a thread invokes a method, the JVM pushes a new frame for the method onto that thread's Java stack. When a method completes, the JVM pops and discards the frame for that method. Each stack frame has three parts: *local variables*, *operand stack*, and *frame data*.

The **local variables** section of a Java stack frame is an array of words. This array stores the parameters to the method and the local variables defined in the method. The parameters to the method are stored before the local variables in the array. The JVM accesses and modifies the local variables stored in the array by using an index. Values of type **int**, **float**, **reference**, and **returnAddress** occupy one entry each in the local variables array. Values of type **byte**, **short**, and **char** are converted into **int** before being stored in the local variables array. Values of type **long** and **double** occupy two consecutive entries in the array. Figure 4.2(a) shows a method **runInstanceMethod** which accepts three parameters and declares two local variables. Figure 4.2(b) shows the local variables section of the method's Java stack frame. The first parameter in the local variables array is of type **reference**. This is a reference to the instance that encapsulates the method **runInstanceMethod**. The second is a character which is stored as an **int** in the local variables array. The third value in the array is a **double** type which occupies two consecutive entries. The fourth value is the **Object 0** which is passed as a reference to the method. Since all objects in Java reside in the heap, the local variables and the operand stack always contain only object references and never an actual image of the object. Following the method's parameter in the local variables array are the two local variables, **boolean b** and **float f**, stored in the order of their declaration.

The **operand stack** is also organized as an array of words. The JVM uses the operand stack as its work space, since it does not have registers. This approach was taken by Java's designers to keep the JVM instruction set compact and to facilitate implementation on a variety of architectures. The JVM instruction set consist of

```

public int runInstanceMethod( char c, double d, Object O ) {
    boolean b;
    float f;
    return 0;
}

```

(a) Java code for an instance method that accepts three parameters, defines two local variables and returns 0.

INDEX	TYPE	PARAMETER
0	reference	hidden this
1	int	char c
2	double	double d
4	reference	Object O
5	int	boolean b
6	float	float f

(b) Arrangement of the method's parameters and the local variables on the local variables section of its stack frame.

Figure 4.2: Declaration of a simple Java method used to demonstrate the arrangement of its parameters and local variables in the local variables section of its Java stack frame

instructions to manipulate the operand stack. These instructions pop values from the operand stack, operate on them, and then push the result back on the stack. Figure 4.3 shows how a JVM uses the operand stack to add two local variables of type `int` and store the result in a third local variable. In this sequence of bytecodes, the first two instructions, `iload_0` and `iload_1`, push the integers stored in the local variable array at positions zero and one onto the operand stack. The `iadd` instruction performs the addition by popping these two integer values, adding them, and pushing their result back onto the operand stack. The fourth instruction, `istore_2`, pops the result of the addition off the stack top and stores it into the local variable array at position two.


```

iload_0 // push the integer in the local variable 0
iload_1 // push the integer in the local variable 1
iadd    // pop two integers, add them, push the result
istore_2 // pop int, store into local variable 2

```

Figure 4.3: Mnemonics generated by a Java compiler to add two local variables that contain **integers** and store the result **integer** in a third local variable.

In addition to the local variables and the operand stack, the Java stack frame includes **frame data**. This portion of the Java stack stores data to support constant pool resolution, normal method return, and exception dispatch. The *constant pool* contains entries for constants such as literal strings, final variable values, class names, and method names associated with the class or interface defined in a class file. Whenever the JVM encounters instructions that refer to an entry in the constant pool, it uses the frame data's pointer to the constant pool to access this information. Aside from constant pool resolution, the frame data assists the JVM with processing normal method completion by restoring, from the Java stack, the stack frame of the method from which the current method was called. If a method throws an exception, the JVM uses the exception table referenced by the method's frame data to determine how to handle the exception.

At the core of any JVM implementation is its *execution engine*. This is responsible for executing the Java bytecodes contained in the methods of loaded Java classes. The JVM specification defines the behavior of the execution engine in terms of its instruction set. Concrete implementations can use a variety of techniques such as interpretation, just-in-time compilation, native execution in silicon, or a combination of these techniques to execute the instructions. A run-time instance of the execution engine is a thread. Each thread of a running Java application is a distinct instance of the virtual machine's execution engine. In Section 4.1.3 we present an overview of the most commonly used instructions in our reasoning systems.

Type	Name	Count
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attribute_count

Figure 4.4: The table shows the ordered list of items that can appear in a class file. Items appear in the table in the order in which they appear in the class file. Each item has a type, name, and count. Type is either a name or one of the primitive unsigned bytes *u1*, *u2*, *u4*, and *u8*. The count indicates the number of items that appears in the class file.

4.1.2 Java Class File Format

The Java class file is a precisely defined binary file format for Java programs. Each class file represents a complete description of one Java class or interface. There is no way to put more than one class or interface into a single class file. The precise definition of the class file format ensures that any Java class file can be loaded and correctly interpreted by any JVM, no matter which system produced the class file or which system hosts the virtual machine. Figure 4.4 shows the major components of the class file in their order of appearance.

magic number or **0xCAFEBAFE** is the first four bytes of every Java class file. The magic number helps the JVM to distinguish Java class files from non-Java files.

minor_version and **major_version** are the next four bytes of the class file. The JVM uses these numbers to identify the format to which a particular class file adheres.

constant_pool_count and **constant_pool** contain the number of entries in the constant pool and constants such as literal strings, variable values, class names, and method names associated with each Java class respectively.

access_flags store several pieces of information about the class or the interface defined in the file. For example, this field indicates whether the file defines a class or an interface and whether it is public or abstract.

this_class and **super_class** contain the fully qualified names of the class or interface and the fully qualified name of the class's superclass respectively. Both **this_class** and **super_class** serve as indices into the constant pool.

interface_count and **interfaces** denote the number of direct superinterfaces of this class or interface and an array of indexes, respectively. Each entry in the **interfaces** array is an index into the constant pool that refers to the fully qualified name of the interface.

fields_count and **fields** describe the number of fields and the fields defined in a class respectively. **Fields** is a list of variable-length **field_info** tables, one for each field. Each **field_info** table contains a field's information such as name, descriptor, and modifiers.

methods_count and **methods** denote the number of methods and the methods explicitly defined in the class. The methods are described in a list of **method_info** tables. Each **method_info** table contains several pieces of information about the method, including the name, return type, and its argument types.

attributes_count and **attributes** describe the number of **attribute_info** tables appearing in the **attributes** list. Attributes come in many varieties. The JVM defines nine types of attributes. To correctly interpret Java class files, all JVM implementations must recognize three of these attributes **Code**, **ConstantValue** and **Exceptions**. The **Code** attribute contains a variable-length **code_attribute** table which defines the bytecode sequence and other information for every method defined in the class. The variable-length **Exceptions** attribute lists the exceptions that a method may throw. The fixed-length **ConstantValue** attribute appears in **field_info** tables for fields that have a constant value.

The detailed structure of the Java class file format is described in the Java Virtual Machine Specification. It is essential for any compiler that generates Java classes to conform to the Java class format. There are libraries and APIs available which, given a set of mnemonics, opcodes, and operands, translate them into binary Java class fields. The Logic-to-Bytecode Compiler uses the Jikes library provided by IBM to output the actual class files corresponding to the **Rule** instances generated by the logic compiler (Philippe Charles and Dave Shields and Vadim Zaliva).

4.1.3 JVM Instruction Set

The JVM specification defines the abstract specification of the Java virtual machine's execution engine in terms of an instruction set. For each instruction, it defines *what* an implementation should do when it encounters the instruction as it executes the bytecodes, but says very little on the actual implementation of the instruction.

The central focus of the instruction set is the operand stack. Most instructions push values, pop values, or both as they perform their functions. For example, to divide one local variable by another, the JVM must push both onto the stack, perform the division, and then store the result back in a local variable. Several goals guided

the instruction set's stack-centered design approach. Platform independence was one of the major goals. The stack-centered approach was chosen instead of the register-centered approach to facilitate efficient implementation of the JVM on architectures with few or irregular registers. This feature of the instruction set makes it easier to implement a JVM on a wide variety of host architectures.

Another motivation for Java's stack-centered instruction set is that compilers usually use a stack-based architecture to pass an intermediate, compiled form of a program to a linker/optimizer. The Java class file, which in many ways is similar to the Unix *.o* or Windows *.obj* file emitted by a C compiler, actually represents an intermediate, compiled form of a Java program. Hence, the stack-centered approach enables compilers to introduce runtime optimizations in conjunction with the execution engine's optimizations such as just-in-time compilation and adaptive optimization.

Another goal that guided the design of the instruction set was compactness. Compactness is important because it facilitates speedy transmission of class files across networks. The total number of opcodes is small enough so that opcodes occupy only one byte each. Lastly, the instruction set is designed keeping in mind Java's security model by providing the ability for bytecode verification. All bytecodes are analyzed and their integrity verified before they are executed to guarantee robustness and security.

A method's bytecode stream is a sequence of instructions for the JVM. Each instruction consists of a one-byte *opcode* followed by zero or more *operands*. The opcode indicates the operation to be performed. Operands supply additional information needed by the JVM to perform the operation specified by the opcode. The opcode itself indicates whether or not it is followed by operands and what form the operands take. Many instructions take no operands and therefore consist only of an opcode. For some opcodes, in addition to the operands that trail the opcode, a

```

public static void doMathForever(){
    int i = 0;
    for( ;; )
        i += 1;
        i *= 2;
}

```

(a) Java code for the method `doMathForever` which indefinitely performs two mathematical operations of adding 1 to a locally defined integer and then multiplying it by 2.

```

// Bytecode Stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
// Method void doMathForever()
// Left column: offset of instruction from beginning of method
| // Center column: instruction mnemonic and operands
| | // Right column: comment
0  iconst_0    // 03
1  istore_0    // 3b
2  iinc 0, 1   // 84 00 01
5  iload_0     // 1a
6  iconst_2    // 05
7  imul_0      // 68
8  istore_0    // 3b
9  goto 2      // a7 ff f9

```

(b) Method `doMathForever()`'s bytecode stream disassembled into mnemonics

Figure 4.5: The bytecodes of the method `doMathForever()` disassembled into JVM mnemonics as they would appear in a binary Java class file.

JVM may refer to data stored in other runtime data areas. When the JVM executes an instruction, it might use entries in the constant pool, entries in current frame's local variables array, or values sitting on top of the method's operand stack. Each type of opcode in the instruction set has a mnemonic. In typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by optional operand values.

For an example of a method's bytecode stream consider the method `doMathForever()` of Figure 4.5(a). The method defines an integer as a local variable and manipulates

it by adding 1 to it and then multiplying it by 2. The stream of bytecodes for `doMathForever()` can be disassembled into the mnemonics shown in Figure 4.5(b). The JVM specification does not define any official syntax for representing a method's mnemonics. The code shown is a representation similar to the output of the `javap` program of Sun's Java 2 SDK. The first instruction, `iconst_0`, pushes the integer 0 on the operand stack. Instruction `istore_0` stores this value in local variable 0. Next, the instruction `iinc` increments the local variable 0 by 1. The fourth instruction, `iload_0`, pushes the value in the local variable 0 on the operand stack. Then, the instruction `iconst_2` pushes the number 2 on the stack. Instruction `imul_0` then multiplies the two integers leaving their result on the stack top. Instruction `istore_0` stores the result in the local variable 0. This addition and multiplication operation repeats when the instruction `goto 2` executes. Note that the jump address for the `goto` instruction is given as an offset from the beginning of the method. It causes the JVM to jump to the instruction at offset two (the `iinc` instruction). The actual value of the instruction's operand in the bytecode stream is minus seven. To execute this instruction, the JVM adds minus seven to the current contents of the PC register. The result is the address of the `iinc` instruction at offset two.

The JVM instruction set has 255 instructions. The logic compiler needs only a small subset of these instructions to generate the Java class for each rule. We summarize the most commonly used instructions below:

Load and Store Instructions : The load and store instructions transfer values between local variables and the operand stack of a JVM frame. For example, instruction `iload` pushes an integer value held in a local variable onto the operand stack. Instruction `astore_<n>` pops a reference to an object or array off the stack and stores it in local variable `<n>`, where `<n>` is 0, 1, 2 or 3 a local variable number.

Arithmetic Instructions : The arithmetic instructions compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack. For example, the `iadd` instruction adds two integers currently on the operand stack.

Object Creation and Manipulation : The instruction `new` is one of the most commonly used instructions to create a new class instance. The instructions to create a new array are also used often, especially while instantiating a `Tuple` with parameters.

Operand Stack Management Instructions : These instructions, such as `pop`, `dup`, and `swap`, directly manipulate the operand stack.

Control Transfer Instructions : In the apply method of many rules, conditional branch instructions like `ifeq` or unconditional branch instructions like `goto` are also used.

Method Invocation and Return Instructions : Method invocation instructions such as `invokespecial` are used to invoke methods.

The JVM's execution engine runs by executing bytecodes, one instruction at a time. This process takes place for each thread of the application running in the JVM. An execution engine fetches an opcode and if the opcode has operands, it also fetches the operands. The engine executes the action requested by the opcode and its operands before proceeding to the next opcode. Execution of bytecodes continues until a thread completes, either by returning from its starting method or by not catching a thrown exception.

4.2 *Logic-to-Bytecode Compiler*

Our second reasoning system, the Logic-to-Java Compiler, operates by creating `Rule` instances for each rule and generating rule-specific inference routines based on

the structure of each rule head. We adapt this technique of creating specialized inference routines for each rule from Prolog. In Prolog, the routines are compiled to run on an instance of the Warren abstract machine (WAM), which defines a close mapping between the terminology of logical deduction and execution of Prolog programs. The WAM's instruction set is optimized for logical deduction. As a result, Prolog programs executing on a WAM exhibit high efficiency.

The JVM and the WAM have similar stack-based architectures. However, unlike Prolog, we cannot optimize the JVM's instruction set for logical deduction to achieve efficiency similar to Prolog. This is because the Logic-to-Java Compiler uses a standard Java compiler to compile `Rule` instances into corresponding Java classes. A standard Java compiler is designed for use in a variety of domains, not just logical deduction. Our solution is to design our third reasoning system, the Logic-to-Bytecode Compiler. The Logic-to-Bytecode Compiler has an enhanced logic compiler which translates a rule into a compiled binary Java class file, instead of a Java source file as generated by the Logic-to-Java Compiler. This process not only eliminates the need to compile the Java source files with a standard Java compiler, but also allows us to introduce bytecode-level refinements that take advantage of the stack-based operating principles of the JVM.

4.2.1 *Compiling Java Classes*

The logic compiler of the Logic-to-Bytecode Compiler generates a class for a rule by directly writing Java bytecodes to the Java class file. Generating Java classes in this manner does not change the function performed by each class of a rule. The advantage is that it allows us to refine the execution of the inference instructions of each rule in a manner that takes advantage of the stack-based execution principles of the JVM. To generate Java class files with this technique, we modify the procedures used by the logic compiler of the Logic-to-Java Compiler to generate class files. Similar to the

Logic-to-Java Compiler the **emitConstructor** of the Logic-to-Bytecode Compiler also uses three procedures to generate a rule's constructor. The **emitHead** procedure now generates bytecodes, instead of Java code, to instantiate a **Literal** for the head literal. Similarly, the **emitBody** procedure generates bytecodes to instantiate the body literals of a rule. Finally, the **emitTerm** procedure shown in Figure 4.2.1, which both **emitHead** and **emitBody** invoke, generates bytecodes to build the terms of each literal. The three procedures **emitHead**, **emitBody**, and **emitTerm** internally rely on the Jikes API to generate the actual bytecodes. This API consists of various classes and methods which accept as input JVM instruction mnemonics and operands for the desired instructions. They translate these mnemonic opcodes and their operands into their corresponding bytecodes and write them into a binary class file. Hence, procedures like **emitTerm** supply the mnemonics and operands for each JVM instruction of the constructor and the apply method of a rule to the Jikes API, which outputs the corresponding binary Java class files.

```

procedure emitTerm( Term term ) {
  if( term is Constant){
    // Push class reference from local variable 0
    encode( opc_aload_0 )
    // Push the constant's value onto stack
    encode( opc_ldc,((Constant)term).getValue())
    /* Call the static method lookup of class Constant
       to generate an unique instance of the constant */
    encode( opc_invokestatic,findMethod("Constant","lookup","(java.lang.String)")
    // Set the result as a value to a field 1
    encode( opc_putfield, field_1 )
  }else if ( term is Variable ){
    // Push class reference from local variable 0
    encode( opc_aload_0 );
    // Create a new object reference of class Variable
    encode( opc_new, findClass( "Variable" ) );
  }
}

```

```

    // Make a copy of reference on stack
    encode( opc_dup )

    // Push the variable's name on stack
    encode( opc_ldc,(( Variable)term).getName() )

    // Invoke Variable's constructor method

encode(opc_invokespecial,findMethod("Variable","<init>","(java.lang.String)") )

    //Assign object reference on the stack as a value to a field
    encode( opc_putfield, field_1 )
}else if ( term is Tuple ){
    Tuple tuple = (Tuple)term;
    // Generate term for the functor
    emitTerm ( tuple.functor );
    // For each of the tuple's parameter invoke emitTerm.
    for ( int i = 0; i < tup.body.length; i++ )
        emitTerm( tuple.body[ i ] );
    // Push the number of parameters of the tuple on the stack
    encode( opc_bipush, tuple.body.length );
    // Create a new array object of class Term
    encode( opc_anewarray, findClass( "Term" ));
    // Push references to all the tuple's parameters on the stack.
    for ( int i = 0; i < tup.body.length ; i++ ) {
        encode( opc_dup );
        // push the index of the parameter
        encode( opc_bipush, i );
        // put the class reference on the stack.
        encode( opc_aload_0 );
        // Push the field's reference on the stack that stores the parameter
        encode( opc_getfield, paramter_i );
        // store the Term in the array location at the index
        encode( opc_aastore );
    }
}

```

```

// Store the array reference temporarily
encode( opc_astore_1 );

encode( opc_aload_0 );
encode( opc_new, findClass( "Tuple" ) );
encode( opc_dup );
encode( opc_aload_0 );
encode( opc_getfield, functor );
encode( opc_aload_1 );
encode( opc_invokespecial,
        findMethod("Tuple","void","<init>","(Term,Term[])" ) );
encode( opc_putfield, local_term );
}
}

```

In order to generate the instructions to build the head literal of a rule, the `emitHead` procedure first invokes the `emitTerm` procedure. The `emitTerm` procedure accepts a `Term` as input and generates appropriate instructions based on the type of term. If the term is a constant, the instructions generated by `emitTerm` are:

```

// Push class reference from local variable 0
aload_0
// Push the string string onto the stack
ldc <String constant >
//
// Call the static method lookup of Constant class to
// generate an unique instance of the constant
// The result is available on the stack.
//
invokestatic Constant.lookup( <String> )
// Set the result as a value to a field constant
putfield <constant>

```

If the term is the first occurrence of a variable, the mnemonics generated by `emitTerm` are:

```
// Push class reference from local variable 0
aload_0
// Create a new object reference of class Variable
new <Variable>
// Make a copy of reference on stack
dup
// Push the string representing the variable on stack
ldc <String variable>
// Invoke variable's constructor method
invokespecial variable
// Assign object reference on the stack as a value to a field variable
putfield <variable>
```

For the second and the subsequent occurrence of a variable in a term, the compiler ensures that the instructions refer to the previously created variable instance by generating following set of instructions.

```
// Push class reference from local variable 0
aload_0
// Retrieve the previously create variable instance.
getfield <variable>
```

If the term is a tuple, `emitTerm` first generates mnemonics for the functor. Next, it generates the mnemonics for each of the term's parameters by invoking itself and then organizes the mnemonics for all the parameters into an array. Lastly, `emitTerm` generates the following mnemonics for the tuple:

```
// Push class reference from local variable 0
aload_0
// Create a new object reference of class Tuple
new <class Tuple>
// Make a copy of the object reference
```

```

dup
// Push another class reference from local variable 0
aload_0
//
// Push reference from of the field that stores the functor.
getfield <functor>
// Push another class reference from local variable 0
aload_0
// Push reference to the array that store the parameters.
getfield <parameters>
// Invoke the Tuple's constructor with the operands
// available on the stack top
// invokespecial Tuple
// Assign the instantiated object Tuple to a field tuple
putfield <tuple>

```

In this manner, **emitTerm** generates the instructions to build all the terms of a head literal. The **emitHead** procedure then generates instructions to instantiate a **Literal**. The **emitBody** procedure also follows a similar process. For each literal in the body, it first invokes **emitTerm** to generate the instructions to build the term and then generates the instructions to instantiate a **Subgoal** for the literal. Thus, **emitConstructor** uses the **emitHead** and **emitBody** procedures to generate every rule's constructor method.

After generating JVM instructions for the constructor, the logic compiler generates the instructions for each rule's apply method. Similar to the Logic-to-Java Compiler, the logic compiler of the Logic-to-Bytecode Compiler uses the knowledge about the structure of the rule head to create the apply method. For each rule, it unfolds the unification steps that the generic unification algorithm would follow and incorporates the corresponding Java bytecodes in the apply method. The **matchConstant**, **matchVariable**, and **matchTuple** procedures still use the same guidelines as the

Logic-to-Java Compiler to predict the unification steps for a constant, variable, or tuple with each possible type of subgoal term. The procedure **matchConstant** generates JVM instructions to unify a constant appearing in the head term. The procedure **matchVariable** generates JVM instructions to unify a variable appearing in the term of the head literal. It generates different sets of instructions based on the variable's occurrence in the term. Since the logic compiler is aware of the complete structure of a rule at compile time, it is able to distinguish the first occurrence of a variable. **matchVariable** uses this knowledge, combined with the fact that the first occurrence of a variable in a rule head will always be unbound, to skip the variable dereferencing and the occurs check operations for the first occurrence of a variable. Similarly, since the logic compiler can also distinguish subsequent occurrences of a variable, **matchVariable** uses the knowledge that such an occurrence may be bound to a term and generates instructions to dereference the variable and perform the occurs check.

The procedure **matchTuple** is used to generate JVM instructions to unify tuples appearing in the term of the head literal. For each tuple, it first generates a set of instructions which verify that the corresponding subgoal term is also a **Tuple** and both the tuples have matching functors and the same number of parameters. Note that for the rule head, **matchTuple** does not generate instructions to match the functor and number of parameters with those in the subgoal term. These instructions are avoided due to the rule index refinement, explained in Section 3.4.2, incorporated in the Logic-to-Bytecode Compiler. Also, **matchTuple** does not explicitly generate unification instructions for each parameter of a tuple. Instead, for each parameter it invokes **emitUnify**, which examines the type of that parameter and generates appropriate unification instructions. The following listing shows the mnemonics generated by the logic compiler for the apply method of the head term $p(a, q(X, Y), X)$ of a rule.

Method **boolean** apply(Subgoal, Trail)

```

0:  aload_1
1:  getfield      <Field Subgoal.term>      // Extract the subgoal term
4:  astore_3                               // Store term in second variable
5:  aload_2                               //
6:  invokevirtual <Method Trail.snapshot>    // Take a snapshot of the trail
9:  astore 4
11: aload_3                               // Load subgoal term and
12: checkcast     <class Tuple>              // cast it to a tuple
15: astore 7                               // Store it in local variable 7
17: aload 7
19: getfield      <Field Tuple.body>          // Load tuple's body reference
22: iconst_0                               // Load first parameter
23: aaload
24: astore 6
26: aload 6                                ////////////// Dereferencing
28: getfield      <Field Term.termRef>        // loop to dereference
31: aload 6                                // the first parameter
33: if_acmpeq 46                               //
36: aload 6                                //
38: getfield      <Field Term.termRef>        //
41: astore 6                                //
43: goto 26                                //////////////

46: aload 6
48: aload_0
49: getfield      <Field Constant 'a'>        // Retrive head term – Constant 'a'
52: if_acmpeq 91                               // Compare it with dereferenced
55: aload 6                                // first parameter.
57: instanceof    <Class Variable>           // Verify that the dereferenced
60: ifeq 83                               // parameter is a Variable
63: aload 6

```



```

131: instanceof    <class Variable>           // is a Variable
134: ifeq          177
137: aload         6
139: aload_0
140: getfield        <Field term>              // Perform an occurs check to determine
143: invokestatic    <Method occurrenceCheck(Term,Term)> // if the subgoal variable
146: ifne           169                       // does not occur in head term q(X,Y)
149: aload         6
151: checkcast        <Class Variable>          // Occurs check fails ,
154: astore         5                       // so cast the subgoal term to Variable
156: aload         5
158: aload_0
159: getfield        <Field term>              // Retrieve head term q(X,Y)
162: aload_2
163: invokevirtual    <Method Variable.bind(Term,Trail)> // Bind variable to head term q(X,Y)
166: goto           261
169: aload_2
170: aload         4
172: invokevirtual    <Method Trail.backtrack(TrailNode)> // Occurs check succeeds, backtrack
175: iconst_0
176: ireturn
177: aload         6
179: instanceof        <Class Tuple>           // Determine if second parameter is Tuple
182: ifeq          253
185: aload         6
187: checkcast        <Class Tuple>
190: astore         8
192: aload         8
194: getfield        <Field Tuple.functor>      // Extract the tuple's functor
197: aload_0
198: getfield        <Field functor>           // Retrieve head term's functor 'q'
201: if_acmpeq       212

```

```

204: aload_2
205: aload      4
207: invokevirtual <Method Trail.backtrack(TrailNode)> // Functors don't match, backtrack
210: iconst_0
211: ireturn                                // Functor's match &
212: aload      8                            // both have two parameters
214: getfield    <Field Tuple.body[]>        //
217: iconst_0                                // Retrieve the tuple's first parameter
218: aaload      //
219: astore      6
221: aload_0
222: getfield    <Field Variable>            // Retrieve head term – Variable 'X'
225: aload      6
227: aload_2                                // First occurrence of variable 'X'
228: invokevirtual <Method Variable.bind(Term,Trail)> // Bind variable 'X'
231: aload      8                            // to first parameter
233: getfield    <Field Tuple.body[]>
236: iconst_1                                // Retrieve tuple's second parameter
237: aaload
238: astore      6
240: aload_0
241: getfield    <Field Variable>            // Retrieve head term – Variable 'Y'
244: aload      6
246: aload_2                                // First occurrence of the variable 'Y'
247: invokevirtual <Method Variable.bind(Term,Trail)> // Bind variable 'Y' to second parameter
250: goto        261
253: aload_2
254: aload      4
256: invokevirtual <Method Trail.backtrack(TrailNode)>
259: iconst_0
260: ireturn
261: aload      7

```

```

263: getfield      <Field Tuple.body[]>      // Retrieve tuple's third parameter
266: iconst_2
267: aaload
268: astore      6
270: aload       6
272: aload_0
273: getfield      <Field Variable>          // Retrieve variable 'X'
276: aload_2              // Second occurrence of variable 'X'
277: invokestatic  <Method unify(Term,Term,Trail)> // so unify variable 'X' & third parameter
280: ifne          291              // after performing all the checks
283: aload_2
284: aload        4
286: invokevirtual <Method Trail.backtrack(TrailNode)> // Unification fails, so backtrack
289: iconst_0
290: ireturn

291: iconst_0
292: istore       7
294: iload       7              // The unification head term
296: aload_0              // and the subgoal term succeeds.
297: getfield      <Field body:Subgoal>      // Push the resulting instantiated
300: arraylength              // body literals on the goal stack.
301: if_icmpge    324          //
304: getstatic    <Field Prover.goalStack>  //
307: aload_0              //
308: getfield      <Field body:[]>          //
311: iload       7              //
313: aaload              //
314: invokevirtual <Method goalstack.push(Subgoal)>
317: pop              //
318: iinc         7, 1          //
321: goto        294          //

```

```

324: aload_1
325: aload_0
326: putfield      <Field Subgoal.parent>      // Set current rule instance
329: iconst_1      <Return true>              // as subgoal's parent
330: ireturn        // return success

```

Figure 4.6: Bytecodes generated by the Logic-to-Bytecode Compiler for the apply method of head term $p(a, q(X, Y), X)$. These instructions are equivalent to bytecodes generated by compiling the apply method shown in Figure 3.13 using a standard Java compiler.

4.2.2 System Components

The Logic-to-Bytecode Compiler has the same underlying framework as that of the Logic-to-Java Compiler. The logic compiler of the Logic-to-Bytecode Compiler differs in its technique of generating a rule's Java class file. It directly generates a class for a rule by writing Java bytecodes representing the Java code to the Java class file. Figure 4.7 shows the control flow between the components of the Logic-to-Bytecode Compiler. The input to the preprocessor is a set of input clauses. In its parsing phase, the preprocessor analyzes these clauses and generates as output instances of **Clause**, one for each clause. In the compilation phase, the logic compiler takes the **Clause** instances as input, analyzes each clause, and generates rules. Then, for each rule the logic compiler generates a Java class file containing bytecodes representing the JVM instructions to build the rule and inference instructions to apply the rule. These Java class files form the **Theory** which is then used by the inference engine.

4.3 Refining Compiled Logic

Our design of the Logic-to-Bytecode Compiler's logic compiler is motivated by the need to exploit the similarities between the stack-based architecture of the JVM and the logical deduction domain. To this end, our first step is to have the logic

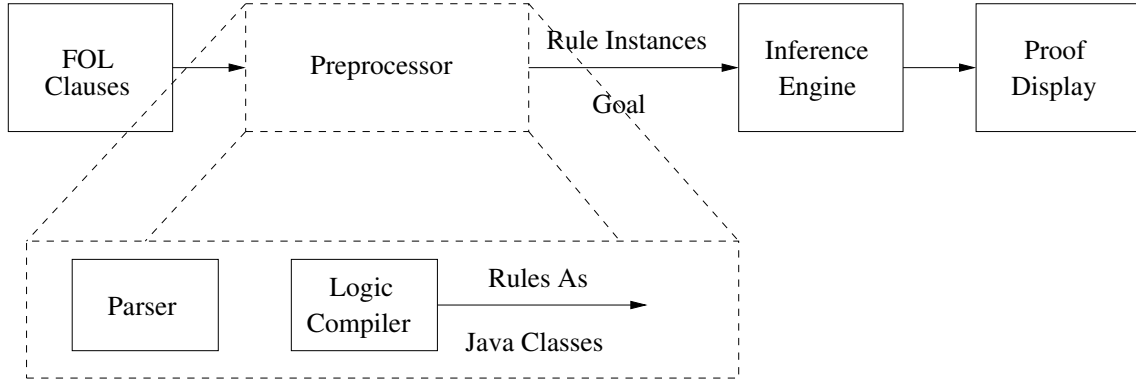


Figure 4.7. The Logic-to-Bytecode Compiler System Components

compiler directly write bytecodes to class files. The code listing of Figure 4.6 generated by the Logic-to-Bytecode Compiler’s logic compiler is not refined to exploit the similarities, and is essentially equivalent to the code generated by the Logic-to-Java Compiler with a standard Java compiler. This section discusses various limitations within this code and the refinements we designed to overcome these limitations.

4.3.1 Bytecode Refinements

Our first set of refinements are the bytecode refinements. These refinements are characterized by the fact that they each provide savings of only a few bytes at a time.

4.3.1.1 *Efficient variable allocation.* The JVM stores the local variables for each method in the local variables array in the method’s stack frame. These variables are addressed by an index, with the first local variable stored at index zero. The JVM permits 64k local variables to be declared in a single method. The JVM instruction set includes generalized two-byte instructions to access any of these 64k variables. However, the instruction set also provides single-byte instructions to access the local variables of a method stored in locations 0 through 3 in the local variables array.

```

0:  aload_2           // Load parameter Trail from variable 2
1:  invokevirtual <Method Trail.snapshot> // Take a snapshot
4:  astore_4           // Store it in local variable 4
6:  aload_1           // Load subgoal from variable 1
7:  getfield          <Field Subgoal.Term> // Extract subgoal term
10: astore_3          // Store term reference in variable 3
11: aload_3

```

Figure 4.8: List of swapping instructions to illustrate the efficient variable allocation refinement implemented by the Logic-to-Bytecode Compiler.

The code listing of Figure 4.6 shows the bytecodes generated for a sample apply method that takes two parameters **Subgoal** and **Trail**. Consider the set of instructions labeled 0 through 19 in the Figure 4.6. These instructions indicate that with a standard Java compiler local variables 1, 2, and 3 store references to **Subgoal** instance, **Subgoal**'s **Term** and the **Trail**, respectively. The array location zero is occupied by a reference to the object that executes the method. Neither of these references are used frequently throughout the method. On the other hand, the instructions labeled 17, 19, 261 indicate that reference to the Tuple's body is frequently pushed and popped off the operand stack. Similarly, the instructions labeled 24, 26, 98, 100, indicate that local variable 6 is frequently used to store and load results of a dereference operation. In this sequence of bytecodes the two most frequently used variables are always accessed with the 2-byte instructions, since array positions 1 through 3 are occupied by the two parameters and the reference to the subgoal's term. The logic compiler of the Logic-to-Bytecode Compiler is designed to make effective use of the single-byte instructions provided to access the local variables 0 through 3. Examining the code listing generated by a standard Java compiler to apply a rule, we observed that the most frequently used variables are usually the reference to the subgoal term's body and the variable used to store the result of a dereference operation. We incorporated this knowledge into the logic compiler so that it generates the bytecodes for each rule's apply method in such a way that these frequently used variables are stored in

```

18: aload_3                // Load parameter from variable 3
19: checkcast    <Class Tuple> // Cast it to Tuple class
22: dup                // Make a copy on the stack
23: astore 8            //
25: getfield    <Field Tuple.body[]> // Reference to tuple's body

```

Figure 4.9. Successive store and load instructions replaced by dup and store instructions.

the first four array locations and are therefore accessible by the single-byte instructions. Figure 4.8 shows the swapping instructions for the method of Figure 4.6. These swapping instructions free-up the local variables 1 and 2 which the logic compiler uses to access the two most frequently used variables: reference to the subgoal's body and the result of the dereference operation.

4.3.1.2 Store followed by fetch. In a rule's Java class there are a number of occasions when an intermediate result is stored in a local variable and is immediately pushed on the stack for the next operation. For example, consider the instructions labeled 15, 17 or 24, 26 shown in Figure 4.6. These instructions store the data on the operand stack in a local variable and immediately fetch it in the next step. Such a store-fetch instruction pair can be replaced by a dup-store instruction pair, which retains a copy of the object on the stack, thereby saving on the fetch instruction. Since the fetch is a three-byte instruction, replacing it with the two-byte dup instruction saves one byte. While a single byte saving may not seem significant, the store-fetch instruction pairs occurs frequently in the apply method of each rule compiled by standard Java compilers. Thus, saving a single byte for each replacement of store-fetch pair with dup-store results in compact code. Figure 4.9 shows the modified bytecode sequence generated by Logic-to-Bytecode Compiler for the instruction pair 15, 17 of Figure 4.6.


```

19 getfield <Field Term. body[]> // Load reference to a Tuple's body
22 dup // Make a copy of this reference,
23 iconst_0 // and retain original on the stack
24 aaload // Use the copy to access the first parameter
.
.
.
62 dup // Make a second copy of the tuple's body reference
63 iconst_1 // Use it to access the second tuple's parameter
64 aaload
.
.
.
168 iconst_2 // Finally use the original copy to access
169 aaload // the third and final parameter of the tuple

```

Figure 4.10: Replacing duplicate data load instructions with load-and-dup pair of instructions to eliminate duplicate data load instructions.

4.3.1.3 *Eliminating duplicate loads.* Consider the set of instructions labeled 17 through 22, 91 through 96, and 261 through 266 from Figure 4.6. These instructions load a reference to a `Tuple`'s body every time the tuple's parameters need to be accessed. The reference to a tuple's body is never modified in an apply method. Moreover, these instructions operate on data not stored in the first four locations of the local variables array. As a result, each load of the tuple's reference requires a two-byte instruction. If the compiler is aware of such operations, it can avoid the two loads by simply duplicating the variable on the stack via a single-byte `dup` instruction when it is first loaded, thereby saving one byte. The Logic-to-Bytecode Compiler is designed to replace the duplicate load instructions by a load-dup pair. It applies this refinement while generating instructions for a rule's apply method. Figure 4.10 shows the refined bytecodes generated by Logic-to-Bytecode Compiler to replace the multiple instructions to load tuple reference.

4.3.2 Dereference Loop Refinements

Loops and control instructions are one of the promising sources of optimization in any program. The Logic-to-Bytecode Compiler applies some refinements to the loops used extensively during ME extension. One example is the while loop that contains instructions to dereference terms. Since the logic compiler does not have knowledge about the types of subgoal terms, it generates the dereference loop to dereference each subgoal term encountered. This makes the dereference loop one of the most frequently executed set of instructions during extension. To understand the refinements the logic compiler implements for the dereference loop, let us consider a **while** loop implemented in Java shown in Figure 4.11(a). Figure 4.11(b) shows the mnemonics for the bytecodes that a standard Java compiler generates for this while loop. An interesting point to note in this sequence is the placement of the conditional jump instruction `if_icmplt`, which represents the test condition $i < 100$. Instructions generated by a standard Java compiler place the instruction for the test condition $i < 100$ after the loop body. In our attempt to improve the efficiency of this loop we find that it is possible to rearrange the placement of the dereference loop test condition in six different ways, based on the contents of the operand stack. Each placement of the loop test condition affects the performance of the dereference loop in a different way. Figure 4.12 shows these implementations. The mnemonic sequence shown in Figure 4.12(Loop Std) is generated by the standard Java compiler and starting with Figure 4.12(Loop 1) are our five alternative implementations of the dereference loop.

The mnemonic sequence for the dereference loop generated by a standard Java compiler, shown in Figure 4.12(Loop Std), is characterized by the placement of the conditional jump instruction `if_acmpne <offset>`. This instruction compares the two objects currently on the operand stack. If the two objects are equal, the execution proceeds to the next instruction. Otherwise, the control jumps back to the offset

```

void whileLoop(){
    int i = 0;
    while( i < 100 ){
        i++;
    }
}

```

(a) Java code for a while loop that iterates from 1 to 100

```

0    iconst_0  // Push the number zero on the stack
1    istore_1  // Pop number zero from the stack into local
variable 1
2    goto 8    // Jump to offset 8
5    iinc 1 1  // Increment local variable 1 by 1
8    iload_1   // Push data from local variable 1
9    bipush 100 // Push the value 100
11   if_icmplt 5 // Check if local variable is less than 100
      // and jump to 5 to continue

```

(b) Mnemonic sequence of a while loop that iterates from 1 to 100

Figure 4.11: Java code of a while loop and its mnemonics generated by a standard Java Compiler.

label_b and again executes the loop. There are various drawbacks in this particular sequence of mnemonics in the context of the dereference operation. First, the execution of the loop begins with a jump instruction **goto**, which requires the JVM to execute a jump even before it executes the body of the loop. Second, before executing the conditional jump instruction **if_acmpne**, the loop loads the term twice from the local variable: first to dereference the variable and then again to use it for the comparison operation. We make the following refinements to eliminate these drawbacks.

For the second drawback, the duplicate loading of the variable, we modify the compilation technique of the Logic-to-Bytecode Compiler to replace the **load-dereference-load-compare** sequence of mnemonics by a **load-dup-dereference-compare** sequence of mnemonics as shown in Figure 4.12(Loop 1). Now, when the execution reaches *label_b*, instead of performing two load instructions, it makes a copy of the

<pre> astore term // term available on stack goto label_b // Store term in local variable term label_a: aload term // Jump to instruction at label_b getfield term.termRef // Load term from local variable astore term // Dereference the term // Store the new term in local variable term label_b: aload term // Load term from local variable getfield term.termRef // Dereference the term aload term // Load term from local variable if_acmpne label_a // Compare the two terms // Jump to label_b if they are not equal </pre> <p style="text-align: center;">Loop Std</p>	<pre> astore term // term available on stack goto label_b // Store term in local variable term label_a: aload term // Jump to instruction at label_b getfield term.termRef // Load term from local variable astore term // Dereference the term // Store the new term in local variable term label_b: aload term // Load term from local variable dup // Copy the term on the stack getfield termRef // Dereference the stack top if_acmpne label_a // Compare the two terms // Jump to label_b if they are not equal </pre> <p style="text-align: center;">Loop 1</p>
<pre> astore term // term available on stack label_a: aload term // Store data in local variable term dup // Load data from local variable rm getfield term.termRef // Make a copy of the term on stack if_aeq label_b // Dereference the term // If the two terms are equal jump to label_b aload term // Otherwise, load term from local variable get term.termRef // Dereference the term astore term // Store the dereferenced term goto label_a // Jump to label_b and repeat the loop label_b: </pre> <p style="text-align: center;">Loop 2</p>	<pre> goto label_b // term available on stack label_a: getfield term.termRef // Jump to label_b label_b: dup // Dereference the stack top dup // Make a copy of the term getfield term.termRef // Make another copy of the term if_acmpneq label_a // Dereference the stack top // Jump to label_a if the two terms are not equal dup // Make a copy of the term astore // Store the term locally for future reference </pre> <p style="text-align: center;">Loop 3</p>
<pre> label_a: dup // term available on stack dup // Duplicate the term on stack getfield term.termRef // Make a second copy on the stack if_acmeq label_b // Dereference the stack top // Jump to label_b if the terms are equal getfield term.termRef // Insert the dereferenced term goto label_a // below the second term from top // Jump to label_a to repeat label_b: ... </pre> <p style="text-align: center;">Loop 4</p>	<pre> label_a: dup // term available on stack getfield term.termRef // Duplicate the term on stack dup_x1 // Dereference the variable // Make a copy of the dereferenced term // Insert the dereferenced term // below the second term from top if_acmpneq label_a // Jump to label_a if the terms are not equal </pre> <p style="text-align: center;">Loop 5</p>

Figure 4.12: Dereference loop implementations studied to achieve efficient execution of dereferencing operation.

term on the stack and accesses the term's `termRef` field to prepare for the comparison. To overcome the first drawback of the while loop, we modify the Logic-to-Bytecode Compiler to create a second implementation of the dereference loop as shown in Figure 4.12 (Loop 2). It generates instructions to replace the `goto – if_acmpne` instruction pair with `if_aeq` to eliminate the unconditional jump executed due to `goto`. Even though the sequence of instructions of Figure 4.12(Loop 2) eliminates the extra jump

instruction, the loop begins with a store and load pair of instructions. To overcome this drawback, we further modify the Logic-to-Bytecode Compiler to create the third implementation of the dereference loop as shown in Figure 4.12(Loop 3).

The Loop 3 implementation replaces the store-load instruction pair with a dup-store pair. This placement allows us to include a dup instruction that makes a copy of the term to be dereferenced on the stack and manipulate it on the stack without storing the intermediate results. Only the final result is stored once the dereferencing completes, thus minimizing the load and store instructions. The execution of the third implementation of the dereference loop begins by first loading the term from a local variable, dereferencing it using the operand stack without storing intermediate results locally, and ends by storing the result in a local variable. Thus, the instructions surrounding the dereference loop are load and store.

In the fourth implementation, shown in Figure 4.12(Loop 4), we modified the Logic-to-Bytecode Compiler such that it does not generate the load and store instructions surrounding the dereference loop. Instead, it generates the instructions preceding the dereference loop so that they leave a copy of the term under consideration on the operand stack. The dereference loop dereferences this term using only the operand stack, without storing intermediate results locally. Finally, instead of storing the dereferenced result locally, this implementation of the loop leaves it on the stack, making it available for the next set of instructions in the apply method. This eliminates the load and store surrounding the dereference loop.

The fifth implementation, shown in Figure 4.12(Loop 5), is the refined version of the fourth implementation. It uses a `dup_x1` instruction, which duplicates the dereferenced term on the stack and inserts the duplicate before the existing term on the stack. For example, if the two terms on the stack are X and Y , where Y is the term obtained by dereferencing X , the `dup_x1` would make a copy of Y on the stack and insert this copy below the second term, X , on the stack. The last two

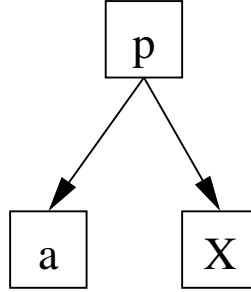


Figure 4.13: Tree representation of a tuple $p(a, X)$, where p is the root, and a and X are leaf nodes.

implementations of the dereference loop are designed to support the stack-based term decomposition refinement, explained next.

4.3.3 Stack-Based Term Decomposition

The JVM is stack-oriented, with most operations taking one or more operands from the stack or pushing results back onto the stack. While the refinements described so far, such as bytecode and dereference loop, enable the Logic-to-Bytecode Compiler to generate compact code, they do not exploit the stack-based architecture of the JVM. A first-order logic term can be visualized as a tree. For instance, constants and variables can be viewed as trees with only a root node and no child nodes. A tuple can be viewed as a tree whose root is the functor and whose child nodes represent the tuple's parameters. Figure 4.13 shows the tree representation of the tuple $p(a, X)$, with p as the root and constant a and variable X as child nodes. Accordingly, the unification of two terms can be modeled as a concurrent depth-first traversal of the trees representing the two terms. At each step in the traversal, unification of the corresponding terms in the tree is performed. For example, as shown in Figure 4.14 the unification of terms $p(a, X)$ and $p(Y, Z)$ involves traversing the trees for these two terms simultaneously. In the first step, the traversals accesses the two functors and attempts to match them. Since they match, the traversals proceeds to the left most

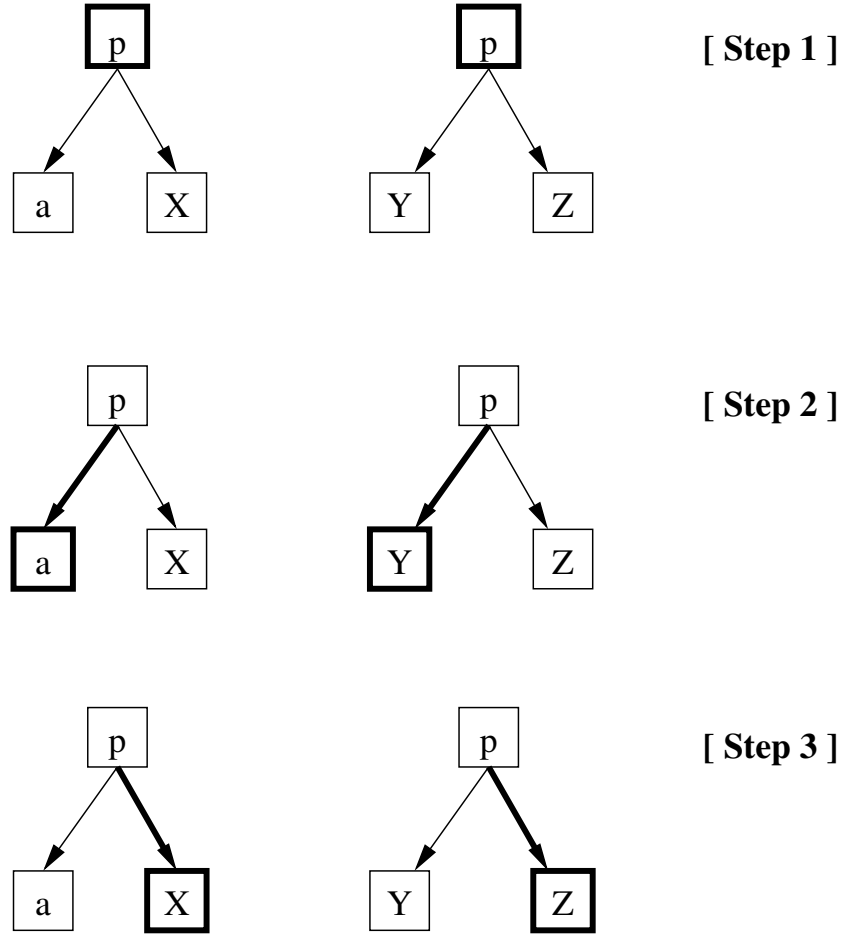


Figure 4.14: Concurrent traversal of two trees representing two terms $p(a, X)$ and $p(Y, Z)$. Nodes marked in bold are the terms being unified at each step.

node in each tree, a and Y respectively. Their unification succeeds. In the last step, the traversals attempt to unify the two remaining terms X and Z . This unification also succeeds. In a similar manner, the unification of a head term with a subgoal term, attempted during an ME extension operation, can be modeled as a concurrent depth-first traversal of the trees representing the two terms.

The structure of a rule's head term is known at compile time. Hence its depth-first traversal is also known at compile time. For example, traversing the head term $p(a, b, s(W))$ in a depth-first manner is equivalent to accessing the **Term** instance for p , followed by accessing a , b , and $s(W)$ in that order. The logic compiler translates the

depth-first traversal steps of a head term into instructions that retrieve each element of the head term in the depth-first order and places them in the rule's apply method.

In contrast, the structure of a subgoal term is not known in advance. Hence, the instructions generated by the logic compiler, which must take into account all possible types of subgoal terms, are not tailored to perform a depth-first traversal of the subgoal term efficiently. For example, consider the instructions from Figure 4.6 labeled 11 through 90, executed by the JVM for the unification of a head term $p(a, q(X, Y), X)$ with any subgoal term. It first stores the subgoal term as a tuple in a local variable. The JVM immediately loads the tuple's body reference on the operand stack to retrieve the tuple's first parameter. This parameter could be any one of constant, variable, or a tuple. So the JVM first dereferences it. It then stores the dereferenced term in another local variable. This dereferenced term is immediately reloaded onto the stack. The JVM then loads the constant a on the stack, and compares the two instances. If this comparison fails, the JVM again reloads the dereferenced term on the stack to determine if its a **Variable**. If it identifies that the term is a **Variable**, the JVM reloads the constant a on the stack and binds the variable to the term a .

Performing a stack-based depth-first traversal of the subgoal term can eliminate the repeated load-store instructions. The same set of instructions could be: store the subgoal term as a tuple locally in the apply method. Load the tuple's body reference on the operand stack and make a copy of the reference and use it to retrieve the first parameter using this reference. Dereference the first parameter. Make a copy of the dereferenced term on the stack. Load the constant a and compare it with the dereferenced term available on the stack. If they are identical, the two term unify. If they do not match, then make a second copy of the dereferenced term on the stack and use it to determine if is a **Variable**. If the JVM identifies the term as a variable, then load the constant a and bind variable, using the dereferenced term available on the operand stack, to the constant a to complete the unification.

The ability to generate bytecodes allows the logic compiler to use the operand stack, provided by the JVM for each method, to implement a stack-based depth-first traversal of the subgoal term at compile-time. In addition, the rich set of stack-manipulating instructions provided by the JVM are made available for the logic compiler to implement an efficient traversal of the subgoal term. This technique of laying out the abstract steps at compile-time that perform a stack-based depth-first traversal of a subgoal term at runtime is called stack-based term decomposition.

The logic compiler of the Logic-to-Bytecode Compiler is designed to generate instructions for each rule that unify a head term with a subgoal term by performing a concurrent depth-first traversal of the two terms. It translates the depth-first traversal steps of a head term into instructions that retrieve each element of the head term in the depth-first order. The instructions to perform a stack-based term decomposition of the subgoal term make use of the JVM operand stack and its stack-manipulating instructions. The main caveat to the stack-based term decomposition using an operand stack is, the state of the operand stack should be properly restored after the traversal completes. Restoring the state of the stack is easy when the subgoal term successfully unifies with the head term. However, at any stage in the traversal, if the unification between the corresponding terms fails, all the terms on the operand stack should be popped off to restore the stack to its state prior to the unification. To handle such cases, the logic compiler uses the compile-time knowledge of the structure of the head term to accurately estimate the number of elements on the operand stack at each step in the traversal and generates instructions to pop those elements. Specifically, it uses the depth of each node in the tree for the head term to predict the number of references on operand stack. For example, consider the instructions generated for the head term $p(a, q(X, Y), X)$. Figure 4.15 shows the tree for this head term. If the unification of any of the terms a , $q(X, Y)$, or X with the corresponding subgoal term fails, then pop one term. Since the depth of these terms is one, the logic compiler

predicts that there will be one reference on the stack, the reference to a tuple. Similarly, to handle the case where the unification of either X or Y could fail, the logic compiler generates instructions to pop two terms off the operand stack, since X and Y are at depth two.

To understand the unification of two terms using stack-based term decomposition, consider the unification of the head term $p(a, q(X, Y), X)$ with the subgoal $p(W, q(Z, Z), s(W))$. Figures 4.15, 4.16, and 4.17 illustrate the unification. The first column of these figure shows the instructions generated to unify the head term $p(a, q(X, Y), X)$ with any subgoal. The second and third columns of the Figures 4.15 and 4.16 show the tree representations of the head term and the subgoal term respectively at each unification step. Each leaf node in these trees represents a variable or a constant. Each non-leaf node represents a functor of a tuple, having branches pointing to the nodes representing each of its parameters. Variables occurring more than once in the same term are represented by a single node with two or more branches pointing to it. The nodes in the head term and subgoal term trees marked in bold are the nodes currently visited and represent the pair of terms being unified. The instructions to unify them appear in the first column alongside the two trees and the contents of the operand stack at the start of the unification step appear in the fourth column. We walk through each step of the unification illustrated in the figure.

- (1) The first step verifies that the head term and the subgoal term have the same functor and same number of parameters. The use of the rule index to obtain a rule whose key matches the subgoal's key eliminates the need to explicitly do these verifications. This step results in a reference to an array containing the parameters of the subgoal tuple p being pushed onto the operand stack.
- (2) The traversal proceeds to the first parameter in the subgoal and the constant a in the head term. A copy of the array reference currently on the stack top is created. This reference is popped off the stack and used to retrieve the first

parameter of the subgoal tuple p , which is pushed onto the operand stack. This term is dereferenced and compared with the constant a from the head. Since they do not match, the type of the dereferenced term is verified to be a variable. Since W is a variable, it is bound to the constant a , causing W to be popped off the operand stack. At the end of this step, the operand stack contains the array reference to the parameters of subgoal tuple p .

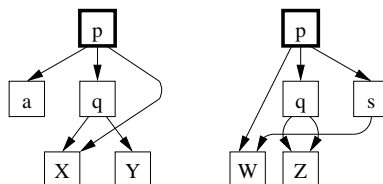
- (3) A copy of the array reference on the stack top is created and then popped off the stack to retrieve the second parameter $q(Z, Z)$. This parameter is pushed onto the operand stack and dereferenced. The dereferenced term is identified as a tuple and a reference to the tuple $q(Z, Z)$ is pushed on the stack. It is verified that $q(Z, Z)$ has two parameters, since the next term from the head to be unified, $q(X, Y)$, is also a tuple with two parameters. Next, the functor q from $q(Z, Z)$ is pushed onto the stack and popped off to compare with the functor q from the head term. Since the two functors match, step 3 concludes with the array reference to the parameters of the subgoal tuple p and the reference to the tuple $q(Z, Z)$ left on the stack.
- (4) This step begins by popping the reference to the tuple $q(Z, Z)$ off the stack to retrieve a reference to the array containing the two tuple parameters Z and Z and placing it on the stack. A copy of this array reference is created on the operand stack and popped off to retrieve the first parameter Z . This parameter is pushed on onto the operand stack to be unified with the first parameter X of tuple $q(X, Y)$ from the head term. Since this is the first occurrence of X in the head term, X is bound to the subgoal parameter Z without performing dereferencing and an occurs check. The unification step pops Z off the stack, leaving two values on the operand stack – the bottom one is the array reference for the parameters of subgoal $p(W, q(Z, Z), s(W))$ and the top one is the array reference for the parameters of tuple $q(Z, Z)$.

Operand Stack

```
Method boolean apply(Subgoal, Trail)
0 aload_2
1 dup
2 astore 5
4 invokevirtual #82 <Method TrailNode snapshot()>
7 astore 6
9 aload_1
10 dup
11 astore 4
13 getfield #85 <Field Term t>
```

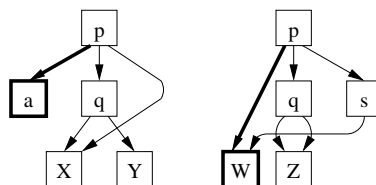
```
// Push subgoal tuple array
16 checkcast #52 <Class Tuple>
19 getfield #88 <Field Term body[]>
```

[Step 1]

 $p(W, q(Z, Z), s(W))$

[Step 2]

```
// Match subterm a
22 dup
23 iconst_0
24 aload
25 dup
26 getfield #91 <Field Term next>
29 dup_x1
30 if_acmpne 25
33 dup
34 astore_1
35 aload_0
36 getfield #31 <Field Term sub_2>
39 if_acmpeq 62
42 aload_1
43 instanceof #37 <Class Variable>
46 ifeq 232
49 aload_1
50 checkcast #37 <Class Variable>
53 aload_0
54 getfield #31 <Field Term sub_2>
57 aload 5
59 invokevirtual #95 <Method void bind(Term, Trail)>
```

 $[W, q(Z, Z), s(W)]$

[Step 3]

```
// Match subterm q(X,Y)
62 dup
63 iconst_1
64 aload
65 dup
66 getfield #91 <Field Term next>
69 dup_x1
70 if_acmpne 65
73 dup
74 astore_1
75 aload_0
76 getfield #57 <Field Term sub_3>
79 if_acmpeq 168
82 aload_1
83 instanceof #37 <Class Variable>
86 ifeq 116
89 aload_1
90 aload_0
91 getfield #57 <Field Term sub_3>
94 invokevirtual #101 <Method boolean occurrenceCheck(Term, Term)>
97 ifne 232
100 aload_1
101 checkcast #37 <Class Variable>
104 aload_0
105 getfield #57 <Field Term sub_3>
108 aload 5
110 invokevirtual #95 <Method void bind(Term, Term)>
113 goto 168
```

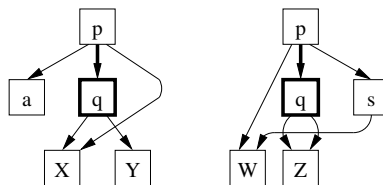
 $[W, q(Z, Z), s(W)]$

Figure 4.15: The bytecodes and the state of the operand stack that compare the functors p , bind W to a during the unification of the two terms $p(a, q(X, Y), X)$ and $p(W, q(Z, Z), s(W))$. The first column of these figure shows the bytecode mnemonics generated for the head term. The second and third columns show the tree representations of the head term and the subgoal term respectively at each unification step. The nodes marked in bold represent the pair of terms being unified.

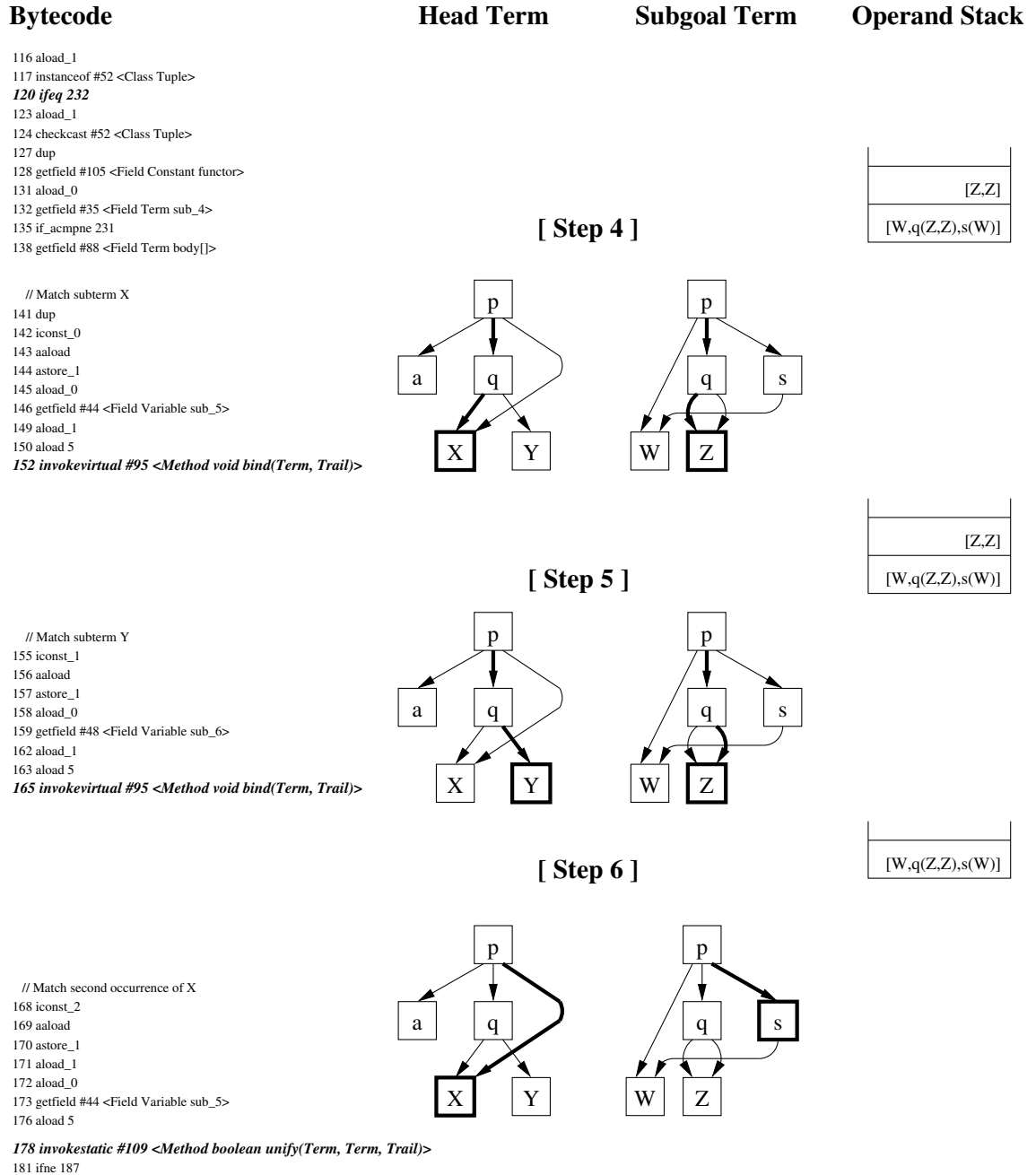


Figure 4.16: Stage in the unification of the two terms $p(a, q(X, Y), X)$ and $p(W, q(Z, Z), s(W))$ that illustrate the state of the operand stack when the bytecodes unify $q(X, Y)$ and $q(Z, Z)$ and bind X to $s(W)$. The first column of these figure shows the bytecode mnemonics generated for the head term. The second and third columns show the tree representations of the head term and the subgoal term respectively at each unification step. The nodes marked in bold represent the pair of terms being unified.

- (5) To unify the second parameter of tuple $q(Z, Z)$, which is also the last, with the second parameter Y of tuple $q(X, Y)$ from the head term, the array reference at the stack top is popped off the stack and used to retrieve the second parameter Z . Again, since it is the first occurrence of Y in the head term, Y is bound to the subgoal parameter Z , without performing dereferencing and an occurs check. At the end of this step, the stack contains the array reference for the parameters of subgoal $p(W, q(Z, Z), s(W))$.
- (6) This step unifies the last parameter X of the head term with the last parameter $s(W)$ of the subgoal term. The array reference at the top of the stack is popped off the stack and used to retrieve the third parameter $s(W)$. Since this is the second occurrence of X in the head term, it is verified that X does not occur in $s(W)$ before binding X to $s(W)$. This step successfully completes the unification between the head term $p(a, q(X, Y), X)$ and the subgoal term $p(W, q(Z, Z), s(W))$, with the operand stack empty.
- (7) The last step considers the result of the unification. If the unification succeeds, the instructions from offset 184 through 230, shown in Figure 4.17, execute. They push the body of the rule onto the goal stack and return success.
- (8) The logic compiler generates a special set of instructions to handle the case where the unification can fail. In order to restore the state of the operand stack at any stage in the unification, the logic compiler adds pop instructions at the end of each apply method. The number of pops generated is equal to the the depth of the tree representation of the head term. Instead of placing one or more pop instructions in each portion of the apply method that handles the failed unification of a term, the logic compiler simply places a jump instruction that shifts the execution to the pop instructions placed

Bytecode	Head Term	Subgoal Term	Operand Stack
<i>// Success</i>			
184 goto 233	//////////	//	
187 iconst_0		//	
188 istore 7		//	
190 goto 213		//	
193 getstatic #115 <Field java.util.ArrayList gs>		//	
196 aload_0			
197 getfield #74 <Field Subgoal body[]>		// When unification succeeds	
200 iload 7		// push the instantiated subgoals	
202 aaload			
203 invokevirtual #121 <Method boolean add(java.lang.Object)>		// on the goal stack	
206 pop		// and return success	
207 iload 7			
209 iconst_1			
210 iadd		//	
211 istore 7		//	
213 iload 7		//	
215 aload_0		//	
216 getfield #74 <Field Subgoal body[]>		//	
219 arraylength			
220 if_icmplt 193		//	
223 aload 4		//	
225 aload_0		//	
226 putfield #125 <Field Rule r>		//	
229 iconst_1			
230 ireturn	//////////	//	
<i>// Failure</i>	//////////	//	
231 pop		// When unification fails	
232 pop		// pop exact number of references	
233 aload 5		// on the operand stack	
235 aload 6			
237 invokevirtual #129 <Method void unwind(TrailNode)>		// and return failure	
240 iconst_0			
241 ireturn	//////////	//	

Figure 4.17: Instructions generated to handle success or failure of the unification. Instructions labeled 184 through 230 push the instantiated subgoals on the goal stack and return success. The instructions labeled 231 through 241 are the pop instructions generated to restore the state of the operand stack when the unification fails. The number of pop instructions generated, two, is equal to the the depth of the tree representation of a rule's head term.

at end. The address of the jump instruction ensures that the number of pop instructions executed restore the operand stack to its state before the failure.

This technique leads to an overall decrease in the number of instructions executed to apply a rule. While stack-based term decomposition utilizes the operand stack for the depth-first traversal of the subgoal term, it uses the stack in a slightly different manner from most stack-based depth-first traversals. Usually, stack-based depth-first traversals operate by pushing all the child nodes of a certain non-leaf node onto the stack before processing them. In contrast, stack-based term decomposition

pushes a reference to a non-leaf node on the stack. It then repeatedly uses this reference to access each of its children one at a time instead of pushing all of them onto the stack. This approach is chosen in order to accommodate the JVM's implementation of the operand stack in register-centric architectures. In register-centric architectures the JVM's operand stack is maintained in the main memory. This implementation results in excess memory transactions to push and pop data between local variables and the operand stack and again between the operand stack and the CPU registers. Our technique of loading the child nodes one at a time minimizes the memory transactions, while still exploiting the stack-based nature of the JVM for unification.

CHAPTER FIVE

Evaluation

This chapter describes the evaluation of our three reasoning systems. Our evaluation is based on the results obtained by testing each reasoning system with two suites of problems. The first suite is made up of problems that we designed to test the behavior of our reasoning systems under various conditions. The second suite consists of problems from the TPTP problem library (Sutcliffe, Suttner, and Yemenis 1994). The TPTP is a large collection of first-order logic problems which serve as a common suite for evaluation and comparison of reasoning systems.

5.1 Test Problem Suite

- 1) $match(bb, bb, bb)$
- 2) $\neg match(bb, Y, Z) \vee match(bb, s(X, Y), s(X, Z))$
- 3) $\neg match(X, s(1, Y), Z) \vee match(s(0, X), Y, Z)$
- 4) $\neg match(X, s(0, Y), Z) \vee match(s(0, X), Y, Z)$

(a) Class 1 input clause set

$$\begin{aligned} \text{Goal G11: } & \neg \text{match}(s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, bb))))))))) \\ & Y, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, bb))))))))) \\ \text{Goal G12: } & \neg \text{match}(s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, bb))))))))) \\ & Y, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(0, bb))))))))) \\ \text{Goal G13: } & \neg \text{match}(s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, s(0, bb))))))))) \\ & Y, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, s(1, s(0, bb))))))))) \end{aligned}$$

(b) Class 1 goal set used in the evaluation.

Figure 5.1: The class 1 clause set and goals designed to generate long-skinny proof trees. The tree is skinny because the proof is found on a single branch of the tree.

Our test suite is designed to test the effectiveness of the proof process and the refinements implemented in our reasoning systems. It consists of five classes of

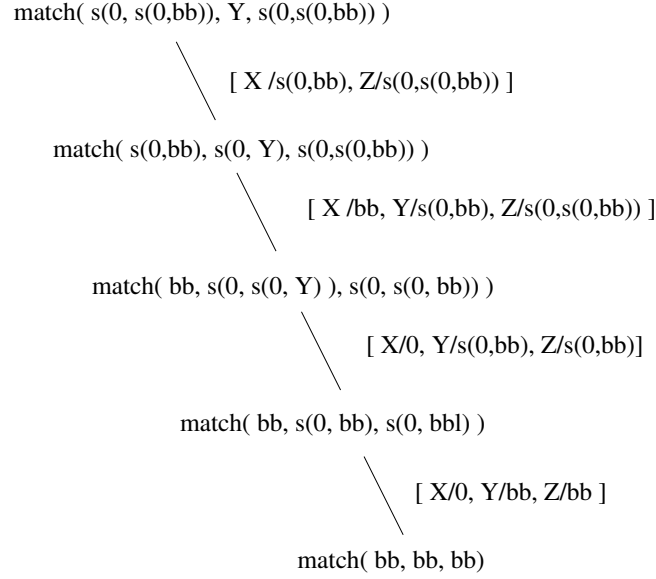


Figure 5.2: An example of a long-skinny proof tree derived for the goal $match(s(0, s(0, bb)), Y, s(0, s(0, bb)))$ from the Class 1 clause set of Figure 5.1(a).

problems. Each class of problems has an input clause set and three goals to be proved. Our aim is to record the execution time of each reasoning system as it proves the goals from all five classes of problems.

5.1.1 Class 1: Long-Skinny Proof Trees

Our first class of problems is designed to observe the performance of the iterative-deepening strategy, a combination of breadth and depth-first search strategies, used in our reasoning systems to find the proof for a subgoal. Specifically, the first class of problems is designed such that the proof is found by traversing deep down on a single branch of a proof tree. To achieve this, the input clause set for this class of problems consists of simple clauses, each having at the most two literals. Figure 5.1(a) shows the input clause set used in our evaluation. Consider the proof tree for $match(s(0, s(0, bb)), Y, s(0, s(0, bb)))$, shown in Figure 5.2. Since each clause has at the most two literals, each ME extension step generates a single subgoal, causing every node in the proof tree to have exactly one branch and making the proof

- 1) $guess(bb, Z)$
- 2) $check(bb, bb)$
- 3) $\neg check(Y, Z) \vee check(s(X, Y), s(X, Z))$
- 4) $\neg guess(X, s(0, Y)) \vee \neg guess(X, s(0, Y)) \vee \neg guess(X, s(0, Y)) \vee guess(s(0, X), Y)$
- 5) $\neg check(Y, Z) \vee \neg guess(X, Y) \vee \neg check(Y, Z) \vee \neg guess(X, Y) \vee$
 $\neg check(Y, Z) \vee \neg guess(X, Y) \vee \vee match(X, Y, Z)$

(a) Class 2 input clause set designed to generate fat-bushy proof trees

- Goal G21: $\neg match(s(0, s(0, s(0, bb))), Y, s(0, s(0, s(0, bb))))$
 Goal G22: $\neg match(s(0, s(0, s(0, s(0, bb))))), Y, s(0, s(0, s(0, s(0, bb))))))$
 Goal G23: $\neg match(s(0, s(0, s(0, s(0, s(0, bb))))), Y, s(0, s(0, s(0, s(0, s(0, bb))))))$

(b) Class 2 goal set used in the evaluation.

Figure 5.3. The Class 2 Clause Set And Goals

tree skinny. Finding the proof therefore requires the proof procedure to traverse the entire depth of the single branch of this tree. We use the goals G11, G12, and G13 shown in Figure 5.1(b) for all our evaluations.

5.1.2 Class 2: Fat-Bushy Proof Trees

Class 2 problems are also designed to test the performance of the iterative-deepening search strategy. They test the cases when the proof is found at a shallow depth in the proof tree, but only after traversing the entire breadth of the proof tree. To generate such proof trees, the input clause set of Class 2 problems, shown in Figure 5.3(a), consists of clauses which generate several subgoals at each ME extension step. For example, when a subgoal term $\neg guess(s(0, bb), Y)$ unifies with the head term $guess(s(0, X), Y)$ from the fourth clause, it generates three subgoals $\neg guess(bb, s(0, Y))$, $\neg guess(bb, s(0, Y))$, and $\neg guess(bb, s(0, Y))$. This characteristic contributes to the breadth of the proof tree. Figure 5.4 shows one such proof tree for the goal $match(s(0, s(0, bb)), Y, s(0, s(0, bb)))$, based on the input clause set of Figure 5.3(a). This proof tree is fat and bushy, because it has a shallow depth, the

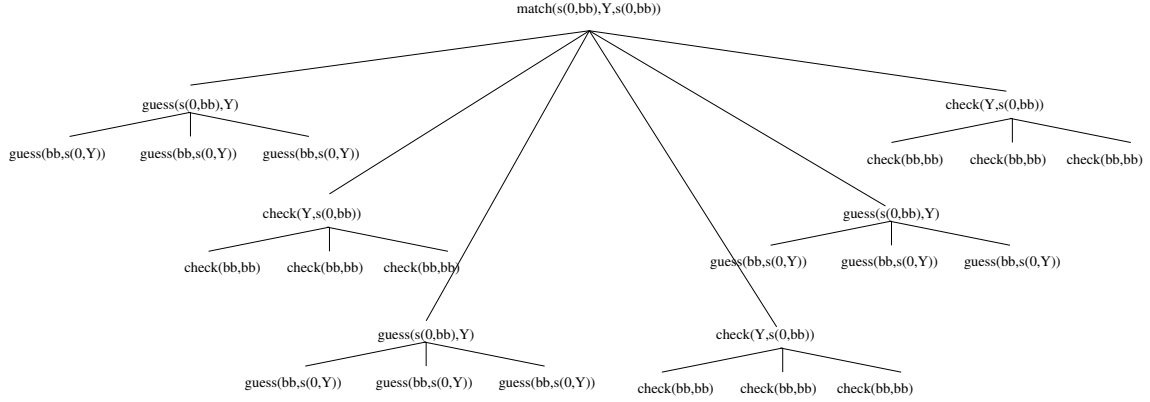


Figure 5.4: A short-bushy proof tree derived for the goal $guess(s(0, X), Y)$ from the Class 2 clause set of Figure 5.3(a). This tree is fat and bushy, because it has a shallow depth, the root has 8 children, and all the other nodes except the leaf nodes have at least one child.

root has 8 children, and all the other nodes except the leaf nodes have at least one child. All our evaluations use the input clause set of Figure 5.3(a) to prove the goals G21, G22, and G23 shown in Figure 5.3(b).

5.1.3 Class 3: Long Variable Bindings

Class 3 problems are geared towards observing the performance of the dereference loop. To this end, the Class 3 clauses, shown in Figure 5.5(a), contain terms such that during dereferencing, the dereference loop goes through a large number of iterations. Figure 5.6 shows a proof tree to illustrate this concept. Consider the unification step which binds subgoal variable $X2$ to the term bb from the literal $match(bb, s(X, Y), s(X, Z))$ clause of (14). Before binding variable $X2$ to the term bb , the variable $X2$ is dereferenced. Dereferencing E yields variable $X1$, which itself is bound to variable X . Dereferencing variable X yields the constant bb . Thus, dereferencing $X2$ yields the constant bb which is identical to the constant bb from the clause (14). Thus, dereferencing $X2$ requires four iterations of the dereference loop. Proving the three goals G31, G32, and G33, defined for the clause set of Figure 5.5(a), entails several dereferencing operations allowing us to observe the effect on the performance.

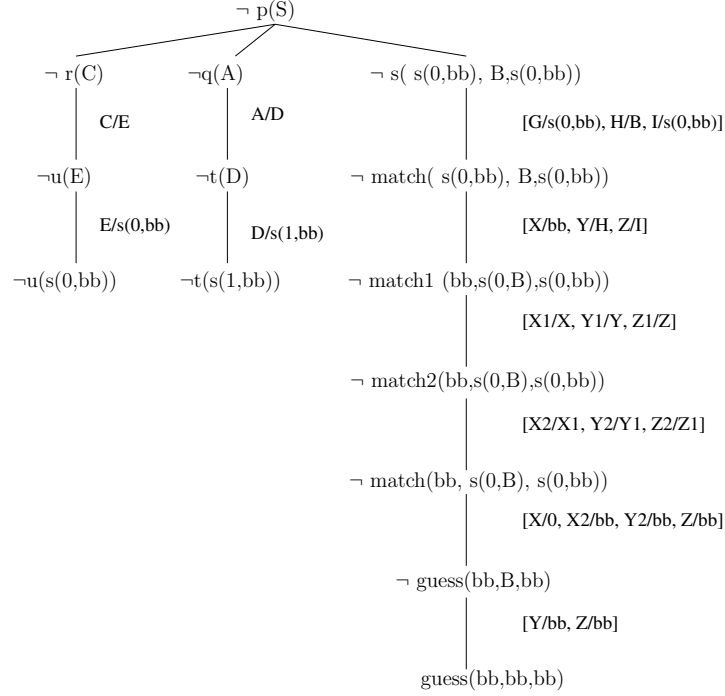


Figure 5.6: Proof tree derived from the Class 3 problem set of Figure 5.5(a) to illustrate the presence of long chains of variable bindings.

$\neg p(e(E E, aa(B B, cc(D D))), f(F F, l(a(B B), C C)), l(a), m(b))$. The stack-based term decomposition of this subgoal term results in considerable use of the operand stack. Consider the stage in the unification when the JVM attempts to unify the term DD with a term from a rule head. To access this term, a reference to the tuple p is placed on the stack before decomposing the tuple into its functor p and parameters. Now, since each of the parameters is also a tuple, a reference to the tuple e is pushed onto the operand stack. This process is repeated for the tuples aa , cc , and finally the term DD . In all, performing the stack-based decomposition of the subgoal term DD involves storing five tuple references on the operand stack. Figure 5.8(b) shows the three goals G41, G42, and G43, defined for the clause set of Figure 5.8(a), which we use for our evaluations.

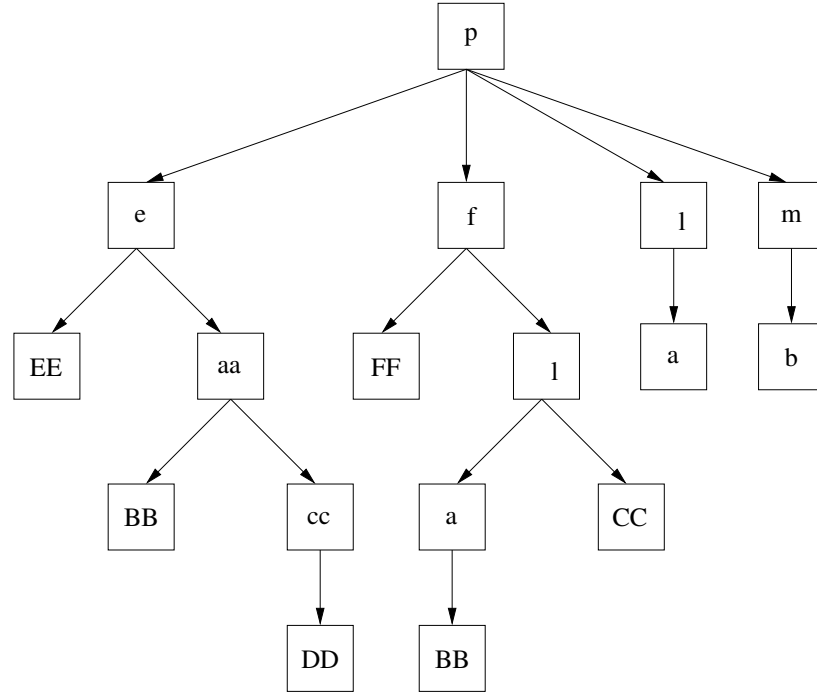


Figure 5.7: Tree representation of a subgoal term $\neg p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), l(a), m(b))$ used to illustrate a complex term. Decomposing such a complex term on the JVM operand stack, pushes as many as five term references on the stack. Such terms are used to evaluate the performance of stack-based term decomposition.

5.1.5 Class 5: Propositional Logic

Class 5 problems consist of clauses in propositional logic. Every clause in propositional logic is a declarative sentence which is either true or false but never both. Hence, these clause do not contain variables. Unlike the class 3 problem set, which generated long chains of variable bindings, the class 5 clauses do not generate any bindings due to the absence of variables in the clause set. As a result, the class 5 problems are useful to observe the performance of our system when the unification operation does not generate any variable bindings. Unlike the clause sets belonging to other classes of problems, we do not generate clauses for Class 5 clause sets. Instead, we use three problems from the TPTP problem library (Sutcliffe, Suttner, and Yemenis 1994). The first problem is problem number *PUZ014-1.p* from the puzzles

- 1) $p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), l(a), m(b))$
- 2) $p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), s(X), t(Y, Y)) \vee$
 $\neg p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), X, Y)$
- 3) $p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), U, V, r(s(U), t(V, V))) \vee$
 $\neg p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), U, V)$
- 4) $p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), U, V, x(T)) \vee$
 $\neg p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), U, V, T)$
- 5) $\neg p(e(EE, aa(BB, cc(DD))), f(FF, l(a(BB), CC)), U, V, T)$

(a) Class 4 input clause set containing literals with complex terms.

- Goal G41: $\neg p(c(CC, l(LL, n(O))), dd(aa(BB, cc(DD))), e(EE, aa(BB, cc(DD))),$
 $f(FF, l(a(BB), CC)), s(s(s(s(s(Q))))), T, x(x(Z)))$
- Goal G42: $\neg p(c(CC, l(LL, n(O))), dd(aa(BB, cc(DD))), e(EE, aa(BB, cc(DD))),$
 $f(FF, l(a(BB), CC)), s(s(s(s(s(Q))))), T, x(x(Z)))$
- Goal G43: $\neg p(c(CC, l(LL, n(O))), dd(aa(BB, cc(DD))), e(EE, aa(BB, cc(DD))),$
 $f(FF, l(a(BB), CC)), s(s(s(s(s(s(Q))))))), T, x(x(Z)))$

(b) Class 4 goal set used in the evaluation.

Figure 5.8. The Class 4 Clause Set And Goals

problem domain (Lusk and Overbeek 1985). The other two problems are *SYN003-1.006.p* and *SYN004-1.007.p* from the syntactic problem domain (Plaisted 1982). For all the tests, we record the execution time to solve each of these three problems.

5.1.6 Test Environment

We execute our tests on an 800 MHz, Pentium III, dual-processor system with one gigabyte of RAM. The system is shared between multiple users. The Java version used for the evaluations is 1.4.0. During an evaluation we record the execution time taken by reasoning system under consideration for each of the 15 goals consisting of three goals from each of the five classes of problems in the test suite. Each recorded execution time is the average over 20 executions for that goal. This is done to account for the variations in the execution times due to other processes that utilize CPU cycles and memory.

5.2 *Configuring the Logic-to-Bytecode Compiler System*

The logic compiler designed for the Logic-to-Bytecode Compiler system implements four refinements; rule caching, bytecode refinements, dereference loop refinements, and stack-based term decomposition. We test the effect of each of these refinements individually on the performance of the Logic-to-Bytecode Compiler system. The results of these tests are used to create a configuration of the Logic-to-Bytecode Compiler system, having some combination of these refinements that gives the best performance.

5.2.1 *Rule Caching*

The Logic-to-Bytecode Compiler system maintains a cache of **Rule** instances created during the course of a proof procedure. Its inference engine always searches the cache for the required **Rule** instance, instantiating a new instance only if it is not in the cache. This reduces the overhead on Java's runtime environment and should benefit the performance of the inference engine. It should be noted that the rule caching refinement is included in all three reasoning systems. Since it provides equal performance benefits in all three systems, we restrict a discussion of its benefits to the Logic-to-Bytecode Compiler system. We run a set of tests to observe the effect of rule caching on the performance of the Logic-to-Bytecode Compiler system. Table 5.1 lists the execution time in seconds of the Logic-to-Bytecode Compiler system to prove the 15 goals without and with rule caching. Each execution time for a goal is the average over 20 executions for that goal. The table is divided horizontally into five parts, with one part for each class of problems. Each part consists of three rows, with one row for each goal in that class of problems. The first column indicates the problem class. The second column contains an identifier for a goal. The third and fourth columns are the execution times in seconds of the Logic-to-Bytecode Compiler system to prove the goal, without and with rule caching respectively. For each goal,

Table 5.1: Execution times of Logic-to-Bytecode Compiler system in seconds to prove the test suite goals, without and with rule caching.

Problem Class	Goal Identifier	Without Caching	With Caching
Skinny	G11	21.83	10.887
	G12	48.281	24.985
	G13	108.472	57.904
Bushy	G21	6.088	2.218
	G22	35.752	13.770
	G23	392.293	148.850
Long	G31	16.552	8.018
	G32	41.866	21.318
	G33	114.436	61.079
Complex	G41	7.583	3.905
	G42	12.692	6.929
	G43	20.336	11.768
Prop	G51	17.127	6.03
	G52	0.216	0.087
	G53	0.317	0.134

the lesser execution time is marked in bold. An execution time marked bold in the fourth column indicates a performance benefit due to rule caching. For example, the entry for goal G11 in column four is in bold, since the Logic-to-Bytecode Compiler system proves the goal in 10.887 seconds with rule caching and in 21.83 seconds without rule caching. A comparison of the execution times in the third and fourth columns shows that the Logic-to-Bytecode Compiler proves all the goals in lesser time when rule caching is incorporated. Hence, we conclude that the rule caching refinement benefits the performance of the Logic-to-Bytecode Compiler system.

The performance improvement due to rule caching is also illustrated with the graph shown in Figure 5.9. The graph has a total of fifteen data points, one for each goal in our test suite. Each data point is a solution for a goal, averaged over 20 executions. The X-coordinate for a data point is the execution time in seconds taken by the Logic-to-Bytecode Compiler system to prove the goal without rule caching.

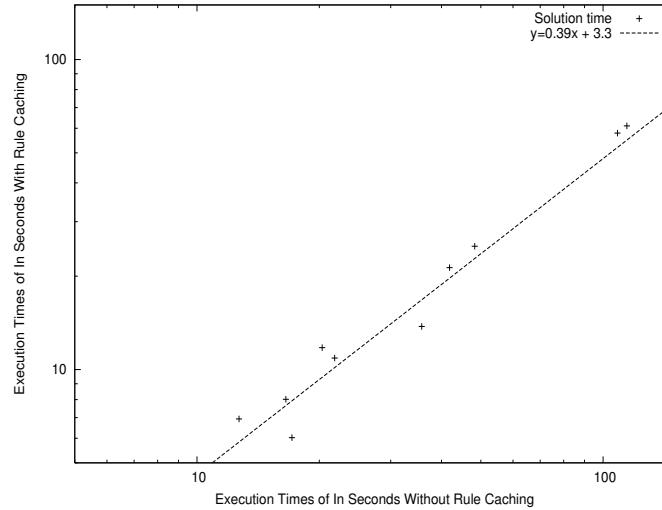


Figure 5.9: Performance improvement of the Logic-to-Bytecode Compiler system when rule caching is incorporated. The Y-axis shows the Logic-to-Bytecode Compiler system's execution times in seconds with rule caching to prove goals in the test suite while the X-axis shows its execution times in seconds without rule caching to prove the same goals. Both axes use a logarithmic scale.

The Y-coordinate for a data point is the execution time in seconds taken by the Logic-to-Bytecode Compiler system to prove the goal with rule caching. We draw a line of best-fit that best represents the data in the graph. The equation of this line is $y = 0.39x + 3.3$. From this equation, we can conclude that for each goal, the execution time of the Logic-to-Bytecode Compiler system with rule caching is approximately 0.4 times the execution time for the same goal without rule caching, confirming that rule caching yields a performance improvement in the Logic-to-Bytecode Compiler system.

5.2.2 Bytecode Refinements

Bytecode refinements that can be done in the Logic-to-Bytecode Compiler system are of two types - elimination of store-fetch and duplicate loads. We evaluate the collective effect of these refinements on the Logic-to-Bytecode Compiler system's performance. For this evaluation we configure the Logic-to-Bytecode Compiler sys-

Table 5.2: Execution times of the Logic-to-Bytecode Compiler system in seconds to prove the test suite goals, without and with bytecode refinements.

Problem Class	Goal Identifier	Without Refinement	With Refinement
Skinny	G11	10.887	10.848
	G12	24.985	24.912
	G13	57.904	57.419
Bushy	G21	2.218	2.211
	G22	13.770	13.748
	G23	148.850	147.631
Long	G31	8.018	8.058
	G32	21.318	21.331
	G33	61.079	61.149
Complex	G41	3.905	3.911
	G42	6.929	7.017
	G43	11.768	11.772
Prop	G51	6.030	6.032
	G52	0.087	0.086
	G53	0.134	0.135

tem with both the bytecode refinements and use it to prove the 15 goals from our test suite. We record the execution time in seconds taken to prove each goal. The execution time recorded for a goal is the average over 20 executions for that goal. Table 5.2 is a listing of the execution times for each goal without and with bytecode refinements. In both cases, rule caching is incorporated in the Logic-to-Bytecode Compiler system. Similar to Table 5.1, this table has five horizontal groups, one for each class of problems. Each row corresponds to the execution times for a goal. The first two columns are for the problem class and the goal identifier. The third and fourth columns contain the execution time in seconds for a goal, without and with bytecode refinements respectively. The lesser execution times are marked in bold. Observing the execution times with bytecode refinements in column four of Table 5.2 we notice a small difference in the execution times. For example, the Logic-to-Bytecode Compiler system needs 10.887 seconds to prove goal G11 without bytecode refinements, and 10.848

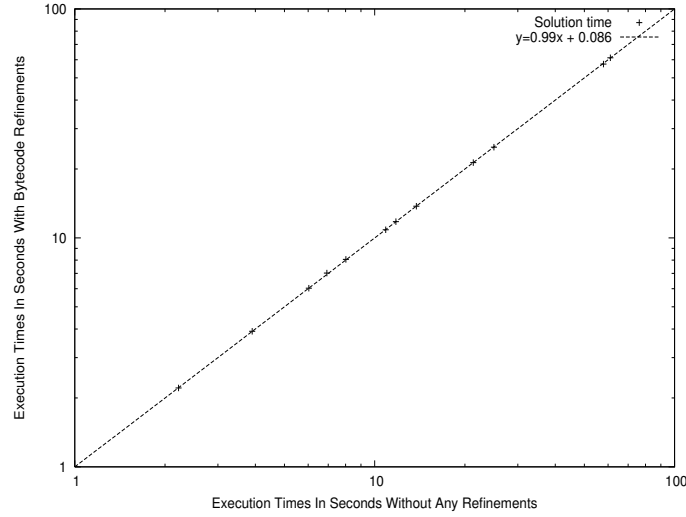


Figure 5.10: The effect of using bytecode refinements on the performance of the Logic-to-Bytecode Compiler system. The Y-axis shows the Logic-to-Bytecode Compiler system's execution times in seconds with bytecode refinements to prove goals in the test suite on a logarithmic scale. The X-axis shows execution times of the Logic-to-Bytecode Compiler in seconds without bytecode refinements for the same goals, also on a logarithmic scale.

seconds to prove it with bytecode refinements - a small benefit of only 0.039 seconds from incorporating bytecode refinements. Some other entries in the table, such as for goal G22 and G23 also exhibit the same characteristic. We find that for some goals, incorporating bytecode refinements leads to an increase in the time taken by the Logic-to-Bytecode Compiler system to prove the goal. For example, the Logic-to-Bytecode Compiler system takes 21.318 seconds to prove goal G32 without bytecode refinements, but needs 21.331 seconds to prove it with bytecode refinements, clearly illustrating a performance degradation caused by incorporating the refinements.

We plot the execution times of Table 5.2 in the graph of Figure 5.10. The X-axis of this graph corresponds to the execution time in seconds of the Logic-to-Bytecode Compiler system without bytecode refinements and the Y-axis corresponds to the execution time in seconds with bytecode refinements. The graph has a total of fifteen data points, where each data point represents the solution to a goal from our test

Table 5.3: Listing of the execution times in seconds to prove the test suite goals, when the Logic-to-Bytecode Compiler system uses each of the six versions of the dereference loop.

Problem Class	Goal Identifier	Standard Version	Loop 1	Loop 2	Loop 3	Loop 4	Loop 5
Skinny	G11	10.887	10.838	10.821	10.972	10.958	10.804
	G12	24.985	24.705	24.732	24.964	24.911	24.706
	G13	57.904	57.821	57.86	57.645	57.479	57.831
Bushy	G21	2.218	2.208	2.210	2.219	2.215	2.211
	G22	13.770	13.785	13.759	13.817	13.836	13.724
	G23	148.850	147.960	147.235	148.532	148.787	147.353
Long	G31	8.018	8.039	8.007	7.977	7.993	8.014
	G32	21.318	21.375	21.309	21.246	21.298	21.288
	G33	61.079	61.299	60.949	61.070	60.900	60.881
Complex	G41	3.905	3.916	3.899	3.912	3.898	3.901
	G42	6.929	6.952	6.959	7.02	6.998	6.96
	G43	11.768	11.748	11.75	11.875	11.827	11.75
Prop	G51	6.03	6.023	6.192	6.191	6.204	6.2
	G52	0.087	0.086	0.085	0.085	0.086	0.084
	G53	0.134	0.134	0.134	0.134	0.134	0.134

suite, averaged over 20 executions. We draw a line of best-fit, that passes through most of the data points in the graph. The equation of this line is $y = 0.99x + 0.086$. From this equation we can see that if the Logic-to-Bytecode Compiler system needs 10 seconds to prove a goal without the bytecode refinements, it will need 9.98 seconds to prove the same goal with the bytecode refinements – not a noticeable decrease in execution time with the bytecode refinements.

Although our test results indicate that bytecode refinements do not provide a noticeable performance improvement and in some cases even degrade the Logic-to-Bytecode Compiler system’s performance, we do not discard them. We retain them to examine the effect on the performance of the Logic-to-Bytecode Compiler system when bytecode refinements are combined with the other refinements.

5.2.3 *Dereference Loop Refinements*

The dereference loop refinements focus on the implementation of the while loop used to dereference terms during unification. We create five different versions of this loop, each containing refinements intended to reduce the time taken by the Logic-to-Bytecode Compiler system to execute the loop. We test these five versions separately in order to find the most efficient one. The system must prove each of the 15 goals in our test suite for six versions of the dereference loop - the five refined versions we created and the dereference loop generated by a standard Java compiler. Rule caching is incorporated into the Logic-to-Bytecode Compiler system for all the tests. We record the execution time in seconds taken by the Logic-to-Bytecode Compiler system to prove each goal, with each version of the loop. The execution time recorded for a goal is the average over 20 executions for that goal. Table 5.3 lists the execution times in seconds for each goal and for all six versions of the dereference loop. Each row of the table lists the problem class and the goal identifier in the first and second columns respectively. The remaining columns list the execution times for each loop version, starting with the loop generated by the standard Java compiler and then each of the five loop versions we created. In each row, the lowest execution time is marked in bold. On comparing the execution times for the six loop implementations, it is clear that we don't have a clear winner – each loop implementation performs well for a few goals, but there is no single implementation that causes the Logic-to-Bytecode Compiler system to prove all or even a majority of the goals faster than any other implementation. Even so, we do not discard this refinement.

5.2.4 *Stack-based Term Decomposition*

As in the tests we have discussed so far, we use the 15 goals from our test suite to test the effect of stack-based term decomposition on the performance of the Logic-to-Bytecode Compiler system. Once again, the Logic-to-Bytecode Compiler

Table 5.4: Listing of the execution times in seconds of the Logic-to-Bytecode Compiler system to prove the test suite goals, with stack-based term decomposition and without it.

Problem Class	Goal Identifier	Without Stack-based Decomposition	With Stack-based Decomposition
Skinny	G11	10.887	10.9
	G12	24.985	24.941
	G13	57.904	57.906
Bushy	G21	2.218	2.223
	G22	13.770	13.81
	G23	148.850	148.958
Long	G31	8.018	8.153
	G32	21.318	21.67
	G33	61.079	62.188
Complex	G41	3.905	3.888
	G42	6.929	6.934
	G43	11.768	11.708
Prop	G51	6.03	6.152
	G52	0.087	0.087
	G53	0.134	0.136

system has rule caching incorporated. It proves the 15 goals without stack-based term decomposition and with stack-based term decomposition incorporated and we record all the execution times. The execution time recorded for each goal is the average over 20 executions for that goal. Table 5.4 lists these execution times in seconds for the fifteen goals. The layout of the table is similar to Table 5.1 with five horizontal sections, one for each class of problems. Each row contains the execution time in seconds for a goal, without the stack-based term decomposition and with it. In each row, the lesser execution time is marked in bold. The execution times in the third and fourth columns clearly show the effect of stack-based term decomposition on the Logic-to-Bytecode Compiler system. The majority of values in the third column marked in bold mean that the execution time without stack-based term decomposition is less than the execution time with it, meaning that the

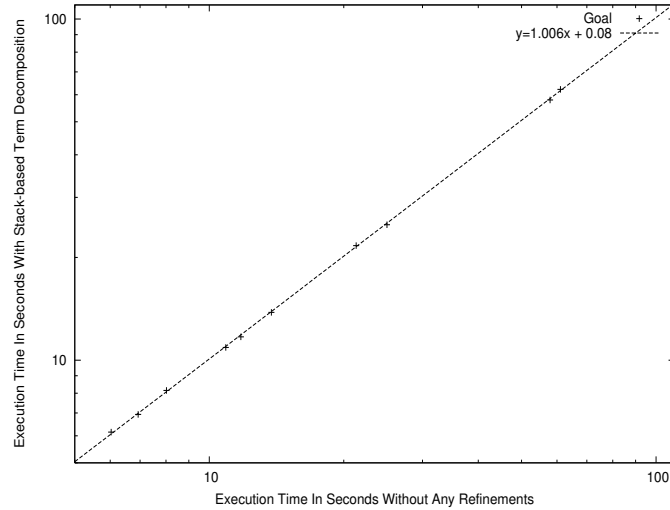


Figure 5.11: Effect on the performance of the Logic-to-Bytecode Compiler system by using the stack-based term decomposition refinement. The Y-axis shows the Logic-to-Bytecode Compiler system's execution times in seconds with the refinement, to prove goals in the test suite. The X-axis shows execution times of the Logic-to-Bytecode Compiler system in seconds without the refinement for the same goals. Both axes use a logarithmic scale.

inclusion of the stack-based term decomposition refinement leads to a performance degradation.

5.2.5 Efficient Logic-to-Bytecode Compiler Configuration

So far from our tests we know that bytecode refinements and stack-based term decomposition do not yield substantial improvement in the Logic-to-Bytecode Compiler system's performance and in some cases cause the performance to degrade. We also know that there is no single implementation of the dereference loop that performs better than the rest. These are outcomes obtained by testing the Logic-to-Bytecode Compiler system separately with each refinement. We conduct experiments to see if combining one or more of these refinements yields better performance. To test each combination of refinements, we use the 15 goals from our test suite and record the execution time of the Logic-to-Bytecode Compiler system to prove each goal. From our experiments, we find that with a combination of rule caching, bytecode refinements,

Table 5.5: A listing of the execution times in seconds of the Logic-to-Bytecode Compiler system to prove the test suite goals. The Logic-to-Bytecode Compiler system’s execution times when it uses each individual refinement and when it uses the combination of all the refinements.

Problem Class	Goal Identifier	With Rule Caching	With Bytecode Refinements	With Stack-based Decomposition	With Refinements Combined
Skinny	G11	10.887	10.848	10.900	10.752
	G12	24.985	24.912	24.941	24.571
	G13	57.904	57.419	57.906	56.748
Bushy	G21	2.218	2.211	2.223	2.178
	G22	13.770	13.748	13.81	13.551
	G23	148.850	147.631	148.958	145.747
Long	G31	8.018	8.058	8.153	7.948
	G32	21.318	20.331	21.67	21.140
	G33	61.079	61.149	62.188	60.57
Complex	G41	3.905	3.911	3.888	3.866
	G42	6.929	7.017	6.934	6.937
	G43	11.768	11.772	11.708	11.740
Prop	G51	6.03	6.032	6.152	5.978
	G52	0.087	0.086	0.087	0.087
	G53	0.134	0.135	0.136	0.134

the fifth implementation of the dereference loop and stack-based term decomposition, the Logic-to-Bytecode Compiler system proves the 15 test suite goals faster with than any other combination. The results of our test for this combination of refinements is shown in Table 5.5. Each row of this table lists the problem class and the goal identifier in the first and second columns respectively. The remaining columns list the execution times of the Logic-to-Bytecode Compiler system with each individual refinement and with the combination of refinements that executes the fastest. Looking at the data in the table, it is clear that with the combination of refinements, the Logic-to-Bytecode Compiler system proves goals faster than almost all of the refinements taken individually. In order to understand this counter-intuitive result, consider the following scenario: During the unification process, the logic compiler dereferences a

subgoal before attempting to unify it. When a subgoal term is dereferenced, the result of the dereferencing is stored into local variables from the stack and is immediately reloaded to perform stack-based decomposition of that term. This requires generating instructions that store the dereferencing results in the local variables and reloading them onto the stack at the start of unification for the subgoal term. Such extra instructions to reload values that were on the stack cause the unification, and consequently, the entire proof to take longer to execute. However, combining the fifth implementation of the dereference loop with bytecode refinements causes the dereferencing result to be retained on the stack. As the result is already on the stack, stack-based term decomposition does not generate instructions to load it. Since the dereferencing results are retained on the stack for every dereferenced subgoal term, it leads to an overall reduction in the number of instructions. This in turn reduces the time taken for unification, resulting in a reduced time to prove the goal. For all our remaining tests with the Logic-to-Bytecode Compiler system, we use the combination of rule caching, the fifth version of the dereference loop, bytecode refinements and stack-based term decomposition.

5.3 *Interpreted versus Compiled Execution*

Our next set of tests compare the performance of the Java Logic Interpreter system with the Logic-to-Java Compiler system. We first record the execution time of the Java Logic Interpreter system to prove the 15 goals from our test suite. Next, we record the execution time of the Logic-to-Java Compiler system to prove the same set of goals. The execution time recorded for each goal is the average over 20 executions for that goal. The Table 5.3 shows these execution times. The table contains one row for each goal. Each row has four columns. The column indicates the problem class, the second column contains the goal identifier and last two columns contain the execution time in seconds for the Java Logic Interpreter system and the Logic-

Table 5.6: Listing of the execution times in seconds for the Java Logic Interpreter system and the Logic-to-Java Compiler system to prove the test suite goals.

Problem Class	Goal Identifier	Interpreted Execution	Compiled Execution
Skinny	G11	20.227	10.793
	G12	46.183	24.766
	G13	106.68	57.646
Bushy	G21	12.502	2.211
	G22	30.999	13.787
	G23	231.58	147.863
Long	G31	11.391	8.032
	G32	29.846	21.283
	G33	84.805	60.981
Complex	G41	195.163	3.878
	G42	239.347	6.904
	G43	265.003	11.786
Prop	G51	7.249	6.904
	G52	0.113	0.087
	G53	0.18	0.133

to-Java Compiler system. In each row, the lesser execution time is marked in bold. All the values in column 4 are marked in bold, indicating that the Logic-to-Java Compiler system needs less time than the Java Logic Interpreter system to prove all the goals. For example, for goal G11 the Java Logic Interpreter system needs 20.227 seconds, whereas the Logic-to-Java Compiler system needs only 10.793 seconds to prove the same goal. The large reduction in execution times with the Logic-to-Java Compiler system clearly indicates that its compiled approach is more efficient than the interpreted approach of the Java Logic Interpreter system.

The superior performance of the Logic-to-Java Compiler system is also illustrated with the graph shown in Figure 5.12. This graph has 15 data points, each representing the solution to a goal from our test suite, averaged over 20 executions. The X-coordinate for a data point is the execution time in seconds taken by the Java Logic Interpreter system to prove the goal. The Y-coordinate for a data point is the

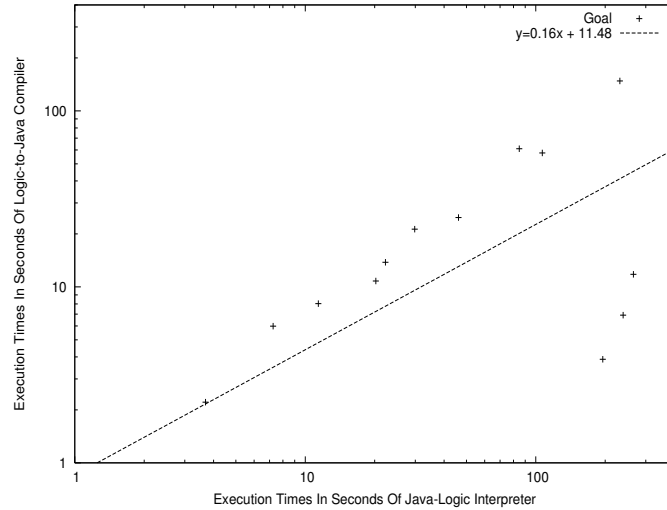


Figure 5.12: Comparison of the performance of the Logic-to-Java Compiler system and the Java Logic Interpreter system. The Y-axis shows the Logic-to-Java Compiler system's execution times in seconds to prove goals in the test suite, on a logarithmic scale. The X-axis shows execution times of the Java Logic Interpreter system in seconds for the same goals, also on a logarithmic scale.

execution time in seconds taken by the Logic-to-Java Compiler system to prove the goal. We draw a best-fit line that passes through most of the data points in the graph. The equation of this line is $y = 0.16x + 11.48$. The equation and the graph show that the compiled execution performs better than the interpreted execution of the Java Logic Interpreter.

5.4 Standard versus Specialized Compiler

Our final test compares the performance of the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system. Similar to all the previous tests, we use the 15 goals from our test suite. We first record the execution time of the Logic-to-Java Compiler system to prove each of the 15 goals. Next, we record the execution time of the Logic-to-Bytecode Compiler system for the same set of goals. The execution time recorded for each goal is the average over 20 executions for that goal.

Table 5.7: Listing of the execution times in seconds for the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system to prove the test suite goals.

Problem Class	Goal Identifier	Java Compiler	Bytecode Compiler
Skinny	G11	10.893	10.752
	G12	24.766	24.571
	G13	57.646	56.748
Bushy	G21	2.221	2.178
	G22	13.787	13.551
	G23	147.863	145.747
Long	G31	8.032	7.948
	G32	21.283	21.140
	G33	60.981	60.57
Complex	G41	3.878	3.866
	G42	6.904	6.937
	G43	11.786	11.740
Prop	G51	5.980	5.978
	G52	0.087	0.087
	G53	0.133	0.134

Table 5.7 contains the results for this test. This table has an identical format to Table 5.3, except that the two rightmost columns now list the execution times for the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system respectively. A glance at the two rightmost columns tells us that the Logic-to-Bytecode Compiler system has lower execution times than the Logic-to-Java Compiler system for almost all the goals. We also see that the difference between the execution times is not substantial. For example, for goal G11 the execution times of the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system differ only by 0.141 seconds. However, the existence of a difference between the execution times is sufficient to validate our hypothesis that a specialized logic compiler, rather than a standard Java compiler, enables us to incorporate refinements that improve the performance.

The graph of Figure 5.13 is a plot for the data in Table 5.7. The X-axis corresponds to the execution time in seconds of the Logic-to-Java Compiler system and the

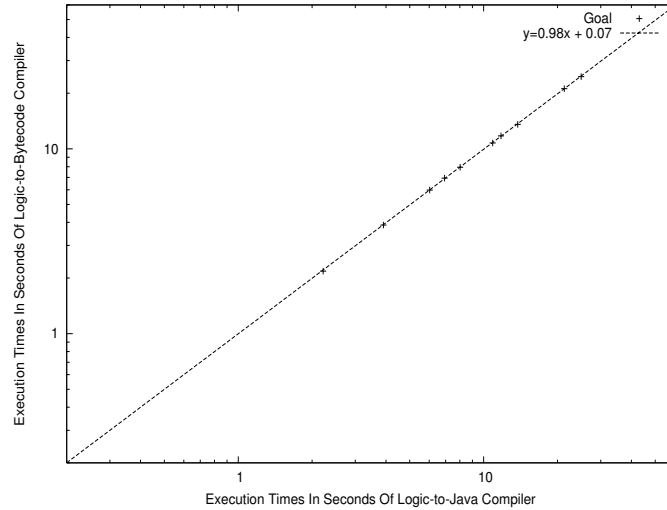


Figure 5.13: Comparison of the performance of the Logic-to-Bytecode Compiler system and the Logic-to-Java Compiler system. The Y-axis shows the Logic-to-Bytecode Compiler system’s execution times in seconds to prove goals in the test suite. The X-axis shows execution times of the Logic-to-Java Compiler system in seconds for the same goals. Both axes use a logarithmic scale.

Y-axis corresponds to the execution time of the Logic-to-Bytecode Compiler system. Similar to the graph of Figure 5.12, this graph also has a total of fifteen data points, where each data point represents the solution to a goal from our test suite, averaged over 20 executions. We draw a line of best-fit that passes through most of the data points in the graph. The equation of this line is $y = 0.98x + 0.07$.

5.5 Experiment With TPTP Problem Suite

In addition to our test suite, we test the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system with several problems described in the TPTP problem library (Sutcliffe, Suttner, and Yemenis 1994). The TPTP problem library is a library of problems for Automated Theorem Proving (ATP) systems. The library contains problems from various domains such as logic, mathematics, geometry, computer science, and engineering. These problems serve as a common suite for evaluation and comparison of ATP systems.

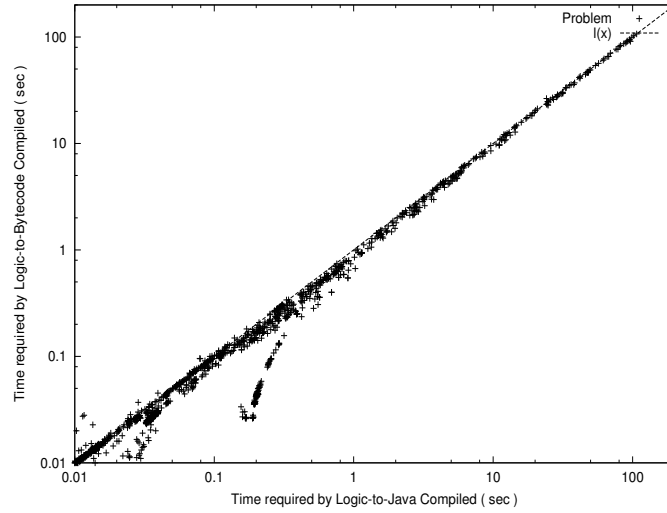


Figure 5.14: The performance of the Logic-to-Bytecode Compiler system compared to the Logic-to-Java Compiler system when both solve the same set of problems from the TPTP library. The Y-axis shows the Logic-to-Bytecode Compiler system's execution times in seconds to solve the problems. The X-axis shows execution times of the Logic-to-Java Compiler system in seconds for the same problems. The data points are plotted on a logarithmic scale.

For each problem in the TPTP library, we assign the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system exactly one minute in which to solve a problem. We record the execution times for only those problems that both the Logic-to-Java Compiler system and the Logic-to-Bytecode Compiler system solved in one minute. The graph of Figure 5.14 shows a plot of these execution times in seconds, with the Logic-to-Java Compiler system's execution times represented on the X-axis and the Logic-to-Bytecode Compiler system's on the Y-axis. In our analysis of this graph, we ignore problems solved in under 0.1 seconds by both the systems since values under 0.1 seconds are too small to compare. Each data point in the graph is the solution time in seconds for proving a goal, averaged over 20 executions for that goal.

Analyzing the graph, we observe that most data points appear slightly below the diagonal $y = x$. This indicates that the performance of the Logic-to-Bytecode

Compiler system is slightly better than the Logic-to-Java Compiler system. In addition, the small vertical cluster of data points between $x = 0.1$ and $x = 0.2$ indicates that a particular set of problems from the TPTP library respond especially well with the Logic-to-Bytecode Compiler system. Overall, the graph supports our hypothesis that a specialized logic compiler, instead of a standard Java compiler will enable us to incorporate refinements into the bytecodes generated, thereby yielding better performance.

CHAPTER SIX

Conclusion

Automated reasoning systems are powerful programs capable of solving complex problems from a wide variety of domains. However, the high computational overhead incurred by such systems inhibits their widespread use. Java, with its platform independence and support for parallelization, seems particularly suited to build reasoning systems that overcome this computational overhead. Our thesis is motivated by the drawbacks in existing Java-based automated reasoning systems. Our aim is to develop an automated reasoning system in Java for full first-order logic that exploits the similarities between Java's architecture and logical deduction techniques. The emphasis throughout is on designing an efficient system that takes advantage of Java's support for building architecture-independent applications, without sacrificing the completeness of the proof procedure.

Our first reasoning system, the Java Logic Interpreter, focuses on creating a Java representation for logical formulae and the implementation of an inference procedure based on Model Elimination. The Java representation is designed to efficiently perform the two ME operations, extension and reduction. The Java Logic Interpreter's proof process is realized as a search procedure based on backtracking. Completeness is guaranteed by performing a depth-first, iterative-deepening search. The Java Logic Interpreter is characterized by its generic inference procedure which analyzes the structure of each input clause at each inference step. This technique is similar to the execution of interpreted programming languages, making the proof process of the Java Logic Interpreter inherently slow.

The design of our second reasoning system, the Logic-to-Java Compiler, is based on the hypothesis that execution of compiled logic is faster than interpreted execution. The architecture of the Logic-to-Java Compiler includes a logic compiler that

has three features designed to collectively speed up the proof process. First, the logic compiler creates rules for each input clause to efficiently perform ME extension, without sacrificing completeness. Second, the creation of rules enables the logic compiler to generate rule-specific inference procedures in advance, by utilizing the compile-time knowledge of the structure of the rule. Third, at runtime, the proof process of the Logic-to-Java Compiler executes these precompiled rule-specific inference routines at each inference step, instead of interpreting the structure of input clauses.

The analysis of the performance of the Java Logic Interpreter versus the Logic-to-Java Compiler shows a large reduction in the time for proving goals with the Logic-to-Java Compiler. This reduction is primarily due to the time saved by eliminating the need to analyze the structure of each rule at runtime, made possible by the execution of compiled inference instructions for each rule. The superior performance of the Logic-to-Java Compiler system also confirms the hypothesis that execution of compiled logic is faster than interpreted execution.

The Logic-to-Java Compiler uses a standard Java compiler to compile the rule-specific inference procedures into executable methods. A general-purpose Java compiler cannot customize Java classes to exploit the similarities between Java's architecture and the unification procedure. The design of our third system, the Logic-to-Bytecode Compiler, is based on the hypothesis that a specialized compiler, rather than a standard compiler, provides ability to incorporate refinements that improve the performance. The Logic-to-Bytecode Compiler has a custom logic compiler which translates the Java representation of each rule directly into a Java class. Its knowledge of the structure of the each rule, along with the ability to directly generate Java classes enables it to incorporate the bytecode refinements, dereference loop refinements, and stack-based term decomposition. These refinements map the stack-based logical inference process of the Logic-to-Bytecode Compiler to the execution principles of Java's runtime environment.

The analysis of the Logic-to-Java Compiler versus the Logic-to-Bytecode Compiler reveals that the Logic-to-Bytecode Compiler needs less time than the Logic-to-Java Compiler system to prove goals, though the difference between the execution times is small. The small difference can be primarily attributed to the implementation of the stack-based JVM on a register-centric processor. Implementing the JVM's operand stack and local variables array on a register-centric processor introduces a hardware abstraction layer that maps the local variables array and the operand stack to CPU registers. Thus, moving data between the stack and local variables translates to moving data between registers (Hsieh, Gyllenhaal, and mei W. Hwu 1996). The refinements incorporated in the Logic-to-Bytecode Compiler are designed to retain results on the operand stack, thereby minimizing the data transfer instructions between the stack and local variables. The JVM can only use a limited number of registers for the mapping of the operand stack. If the number of elements on the operand stack directly map to the number of CPU registers, the data transfer between local variables and the stack is efficient. As the number of elements on the operand stack increases, there are not enough registers to store all the elements on the stack. This in turn introduces extra instructions to manipulate those elements of the operand stack which cannot be mapped to registers. These additional operations, necessitated by the use of CPU registers to implement the operand stack, offset the effects of refinements geared towards making effective use of the operand stack.

6.1 *Future Work*

Our thesis does not explore all the possible refinements that can be made to improve the Logic-to-Bytecode Compiler's performance. Rather, our objective is to motivate the use of specialized compilers by demonstrating the performance gains that can be achieved by using them. The small decrease in the time taken by the Logic-to-Bytecode Compiler to prove goals, as compared to the Logic-to-Java Compiler, is

sufficient to demonstrate this. In future, a different approach can be taken in designing refinements that take into consideration the optimizations implemented by the JVM to compensate for the inadequate support provided by the underlying architectures for its stack-based operational principle. Another area worth exploring is to observe the performance of the Logic-to-Bytecode Compiler on a Java processor (McGhan and O'Connor 1998). A Java processor is an execution model that implements the JVM in silicon to directly execute Java bytecodes. Java processors are tailored to the JVM, providing hardware support for features such as stack processing, multi-threading, and garbage collection. Our reasoning system could potentially perform much better with such an execution model tailored specifically to the JVM than with register-centric architectures, making the Java processor a model worth testing our Logic-to-Bytecode Compiler with.

An important topic barely touched upon in this thesis, is the introduction of parallelization in the logical reasoning process using Java. Logical deduction in first-order logic offers various form of parallelism, such as and-parallelism, or-parallelism, and unification parallelism, depending on which operations are transformed into parallel operations (Gupta, Pontelli, Ali, Carlsson, and Hermenegildo 2001). The Java platform supports several paradigms for high-performance parallel and distributed computing (Getov, Hummel, and Mintchev 1998). Its portable programming language provides extensive support to perform high-performance network parallel computing (Ferrari 1998). Because of its platform independence and uniform interface for parallel computing, the Java platform provides an attractive environment in which to explore the use of parallelization in logical reasoning using Java.

Developing the three reasoning systems and studying their performance demonstrates that Java is suited to developing automated reasoning systems. Even though Java programs typically exhibit more overhead than natively compiled programs, Java provides excellent features to compensate for this overhead. The performance gain

achieved by the refinements incorporated in the Logic-to-Bytecode Compiler indicates that the major impediment to the use of Java for computationally intensive applications like automated deduction – the performance – is not intrinsic to Java and can be solved using various refinement techniques.

Can automated reasoning systems be implemented in Java? Is Java suitable for the computationally intensive domain of logical reasoning? Does Java provide support to overcome the computational overhead inherent to automated reasoning systems? This thesis provides answers to some of these questions. We sincerely hope the discussion presented may contribute to others who ask the same questions.

BIBLIOGRAPHY

- (1999). *NetProlog: a logic programming System for the Java Virtual Machine*.
- Aho, A. V. and J. D. Ullman (1977). *Principles of Compiler Design*. Addison-Wesley.
- Astrachan., O. L. (1992). METEOR: Exploring model elimination theorem proving. Technical Report Technical report DUKE-TR-1992-22.
- Astrachan, O. L. and M. E. Stickel (1992). Caching and lemmaizing in model elimination theorem provers. In *Conference on Automated Deduction*, pp. 224–238.
- Baader, F. and W. Snyder (2001). Unification theory. In J. Robinson and A. Voronkov (Eds.), *Handbook of Automated Reasoning*, Volume I, pp. 447–533. Elsevier Science Publishers.
- Baumgartner, P. and U. Furbach (1994). PROTEIN: A PROver with a theory extension INTERface. In *Conference on Automated Deduction*, pp. 769–773.
- Bundy, A. (1999). A survey of automated deduction. *Lecture Notes in Computer Science 1600*, 153–173.
- Cartwright, R. and J. McCarthy (1979). First order programming logic. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 68–80. ACM Press.
- David A. Plaisted, Y. Z. (2000). *The Efficiency of Theorem Proving Strategies: A Comparative and Asymptotic Analysis*. Friedrich Vieweg and Sohn.
- Ferrari, A. (1998). Jpvm: network parallel computing in Java. *Concurrency: Practice and Experience 10*(11–13), 985–992.
- Fikes, R., J. Jenkins, and G. Frank (2003). Jtp: A system architecture and component library for hybrid reasoning. In *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*, pp. 1–6.
- Getov, V., S. F. Hummel, and S. Mintchev (1998). High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience 10*(11–13), 863–872.
- Gupta, G., E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo (2001). Parallel execution of prolog programs: a survey. *Programming Languages and Systems 23*(4), 472–602.

- Hsieh, C.-H. A., J. C. Gyllenhaal, and W. mei W. Hwu (1996). Java bytecode to native code translation: the caffeine prototype and preliminary results. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, Washington, DC, USA, pp. 90–99. IEEE Computer Society.
- Ibens, O. (1997). The setheo system (system description). In *WLP*, pp. 0–.
- Kaci, H. A. (1991). *Warren's Abstract Machine: A Tutorial Reconstruction*. Mass.: MIT Press.
- Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.* 27(1), 97–109.
- Kowalski, R. (1986). *Logic for problem-solving*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co.
- Letz, R., J. Schumann, S. Bayerl, and W. Bibel (1992). Setheo: a high-performance theorem prover. *J. Autom. Reason.* 8(2), 183–212.
- Letz, R. and G. Stenz (2001). Model elimination and connection tableau procedures. pp. 2015–2112.
- Lindholm, T. and F. Yellin (1999). *The Java Virtual Machine Specification*. Addison-Wesley Professional.
- Loveland *et al.* (1974). An implementation of the model elimination proof procedure. *Journal of the ACM (JACM)* 21(1), 124–139.
- Loveland, D. W. (1968). Mechanical theorem-proving by model elimination. *Journal of the ACM (JACM)* 15(2), 236–251.
- Lusk, E. and R. Overbeek (1985). Non-Horn Problems. *Journal of Automated Reasoning* 1(1), 103–114.
- McGhan, H. and M. O'Connor (1998). Picojava: A direct execution engine for java bytecode. *Computer* 31(10), 22–30.
- Philippe Charles and Dave Shields and Vadim Zaliva. The jikes research virtual machine project.
- Plaisted, D. (1982). A Simplified Problem Reduction Format. *Artificial Intelligence* 18, 227–261.
- Robinson, J. A. (1965a). A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)* 12(1), 23–41.
- Robinson, J. A. (1965b). A machine-oriented logic based on the resolution principle. *J. ACM* 12(1), 23–41.

- Robinson, J. A. (1983). Automatic deduction with hyper-resolution. In J. Siekmann and G. Wrightson (Eds.), *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pp. 416–423. Berlin, Heidelberg: Springer.
- Schumann, J. and R. Letz (1990). PARTHEO: A high-performance parallel theorem prover. In *Conference on Automated Deduction*, pp. 40–56.
- Shinghal, R. (1992). *Formal Concepts in Artificial Intelligence: Fundamentals*, Chapter 3, pp. 34. Chapman and Hall Computing.
- Stickel, M. E. (1986). A Prolog technology theorem prover: Implementation by an extended Prolog compiler. In J. H. Siekmann (Ed.), *Proceedings of the Eighth International Conference on Automated Deduction*, Volume 230, Berlin, pp. 573–587. Springer-Verlag.
- Sutcliffe, G., C. Suttner, and T. Yemenis (1994). The TPTP problem library. In A. Bundy (Ed.), *Proc. 12th Conference on Automated Deduction CADE, Nancy/France*, pp. 252–266. Springer-Verlag.
- van Caneghem, M. and D. H. D. Warren (1986). *Logic programming and its applications*. Ablex Publishing Corp.
- Wos, L., G. A. Robinson, and D. F. Carson (1965). Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM* 12(4), 536–541.