PARALLEL UNIT RESULTING RESOLUTION

Diplomarbeit

am Fachbereich Informatik
der Universität des Saarlandes
von

Christoph Meyer

Saarbrücken, 1996

Erklärung

Ich erkläre die vorliegende Arbeit selbständig im Sinne der Diplomprüfungsordnung erstellt und ausschließlich die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Saarbrücken, 13. Februar 1996

Christoph Meyer

Betreuer: Priv. Doz. Dr. Hans Jürgen Ohlbach
Dr. Peter Graf

Für meine Eltern Karin und René Meyer.

Danksagung

Mein größter Dank gilt meinem Betreuer Peter Graf, der sich stets Zeit nahm für meine Fragen und Probleme und mir durch seine kritischen Kommentare und Verbesserungsvorschläge sehr geholfen hat.

Ebenso danke ich Hans-Jürgen Ohlbach für seine Anregungen und viele fruchtbare Diskussionen, die erheblich zur Entstehung dieser Arbeit beigetragen haben.

Ich danke den Mitarbeiterinnen und Mitarbeitern des Max-Planck-Instituts für die angenehme Atmosphäre und für die große Hilfsbereitschaft bei fachlichen Problemen. Besonders erwähnen möchte ich Christoph Weidenbach und Peter Barth.

Ich danke Michael Christen, Boris Kraft, Jan-Georg Smaus und allen anderen Kommilitonen am Max-Planck-Institut für die Zusammenarbeit und für die schöne gemeinsam verbrachte Zeit.

Saarbrücken, Februar 1996 Christoph Meyer

Preface

The term "Parallel unit resulting resolution" refers to a modified unit resulting resolution rule working on sets of substitutions. We modified the inference rule in order to investigate term indexing and to exploit parallelism in automated reasoning. Term indexing supports the construction of efficient automated reasoning systems by providing rapid access to first-order predicate calculus terms with specific properties. The theoretical background and the implementation of a theorem prover called Purr which implements parallel unit resulting resolution as well as experiments with Purr are presented in this thesis. The author addresses the reader interested in new indexing techniques and in distributed theorem proving. The reader is assumed to have detailed knowledge of automated reasoning and logic.

Contents

List of Figures V

1. Introduction **1**

1.1 Aim of the Work 1

1.2 Parallel Unit Resulting Resolution 2

1.3 Structure of the Work 3

2. Preliminaries **5**

2.1 Rules 5

2.2 First-Order Logic 6

2.3 Graphs and Trees 11

2.4 Algorithms 12

3. UR-Resolution **13**

3.1 Unit Resulting Resolution 13

3.2 Clause Graph 15

3.3 UR-Resolution on Substitution Sets 21

4. Indexing **31**

4.1 Classification of Indexing Techniques 31

4.2 Substitution Tree Indexing 33

4.3 Indexing Operations 38

4.3.1 Subsumption 40

4.3.2 Union 42

4.3.3 Multi-Merge 42

4.3.4 Selection 44

5. Parallelism **47**

5.1 Notions of Parallelism 47

5.1.1 Parallelism in Logic 47

5.1.2 Parallelism in Practice 48

5.2 Parallel Programming Systems 52

5.2.1 Overview 52

5.2.2 PVM 54

6. The Prover 57
6.1 Indexing Algorithms . . . 57
6.1.1 Subsumption . . . 57
6.1.2 Union . . . 59
6.1.3 Multi-Merge . . . 59
6.1.4 Selection . . . 62
6.2 The Implementation . . . 63
6.2.1 Overview . . . 63
6.2.2 Preprocessing . . . 64
6.2.3 Reasoning Phase . . . 66
6.2.7 Postprocessing . . . 75
6.2.8 Options in Purr . . . 77
6.3 Techniques . . . 78
6.3.1 Contexts . . . 78
6.3.2 Indexing and Process Communication . . . 81

7. Experiments 85
7.1 Indexing . . . 85
7.2 Parallelism . . . 92

8. Conclusion 95

Bibliography . . . 97
Index . . . 101

List of Figures

1. Introduction 1

1.1 A Clause Graph 2

2. Preliminaries 5

2.1 Presentation of Algorithms 12

3. UR-Resolution 13

3.1 Undirected Clause Graph 15

3.2 Directed Clause Graph 17

3.3 Directed Clause Graph 17

3.4 Labeled Directed Clause Graph with Test and Send Substitutions 19

3.5 Test Unification and Send Instantiation 23

3.6 Initial State of the Search Sets 27

3.7 `RESOLVE` applied on link $(v_5, (\tau_3, \sigma_3), v_4)$ with receiver literal v_3 27

3.8 `RESOLVE` applied on link $(v_5, (\tau_4, \sigma_4), v_6)$ with receiver literal v_4 28

4. Indexing 31

4.1 Substitution tree 33

4.2 Subsumption in Weighted Substitution Trees 37

4.3 A Selective Substitution Tree 38

4.4 An Algorithm for the Transition Rule `RESOLVE` 38

4.5 Subsumption as an $n{:}m$ indexing task 41

4.6 A Sequence of Selective Substitution Trees 46

5. Parallelism 47

5.1 A Parallel Programming Model 49

5.2 Shared-Memory Model 49

5.3 Message Passing Model 50

5.4 Communication Patterns 51

6. The Prover 57

6.1 Algorithm for `subsume` 58

6.2 Algorithm for `union` 59

6.3 Algorithm for `multi-merge` 60

6.4 Multi-Merge with Three Substitution Trees 61

6.5 Algorithm for `selection` 62

6.6 Algorithm for `partition` 63
6.7 The Master Process 63
6.8 The Preprocessing 64
6.9 Create Clause Graph 65
6.10 Create all links leading to vertices in V_{in} 65
6.11 Delete tautologies 66
6.12 Delete clauses with pure literals 66
6.13 Sequential Reasoning in PURR 67
6.14 UR-Resolution on Link $(v, (\tau, \sigma), w)$ 69
6.15 Optimized UR-Resolution on Link $(v, (\tau, \sigma), w)$ 70
6.16 Optimized UR-Resolution on Literal Associated with v 70
6.17 Termination on Clause C 71
6.18 Termination on Units with Predicate P 72
6.19 Start and Control of the Parallel System 73
6.20 The Postprocessing 75
6.21 Recursive Proof Generation 76
6.22 Two Contexts C_1 and C_2 with Bindings 79
6.23 The Multi-Merge Operation with Three Substitution Trees 79
6.24 Test Unification and Send Instantiation 80
6.25 Process Communication 81
6.26 Transformation of a Substitution Tree 82
6.27 Transformation of a Substitution 82
6.28 Transformation of a Term 83
7. Experiments 85
7.1 Clause Graph of the Problem IC-JL⇒IC-4 87
7.2 Experiments with the Implicational Propositional Calculus 88
7.3 Statistics of the Resolution Process in PURR$_{Literal}$-1 on Problem IC-JL⇒IC-4 . . 89
7.4 Experiments with the Equivalential Calculus 90
7.5 Experiments with the R Calculus 91
7.6 Experiments with PURR$_{Literal}$-1 on the Two-Valued Sentential Calculus 91
7.7 Clause Set in Group Theory 93
8. Conclusion 95

1

Introduction

Parallel unit resulting resolution is based on a modified unit resulting resolution rule working on sets of substitutions. The ur-resolvents are represented by substitutions. We attach the substitution sets to vertices and links of a clause graph [Eis91, Kow75] which is created on the original clause set in order to associate the substitutions with the according literals and clauses. The links of the clause graph determine possible applications of the modified inference rule. New substitutions are created and exchanged among the substitution sets in the clause graph. The graph structure itself does not change. Our modified ur-resolution scheme can be supported efficiently by indexing techniques and, moreover, can be applied to different nuclei concurrently. This concept has been implemented in our distributed theorem prover called PURR. The original idea is based on the PhD thesis of Ulrich Aßmann [Aßm92] and on the work of Antoniou and Ohlbach [AO83].

1.1 Aim of the Work

One of the most important aspects of an automated reasoning system is the employed logic calculus. The inference rules of a calculus often are implemented in order to examine the properties of the calculus or to find solutions of challenging problems. In contrast, we shall employ the inference rule of unit resulting resolution for the investigation of *indexing techniques* and for the exploitation of *parallelism* in automated reasoning. Both indexing and parallelism can significantly improve the *efficiency* of a theorem prover.

Indexing. The maintenance of large databases is supported by indexing which provides fast access to stored data. In automated reasoning we employ databases that contain first-order terms. Typical queries to such term indexes are: Given a database D containing terms (literals) and a query term t, find all terms in D that are unifiable with, instances of, variants of, or more general than t. Thus indexing can be used to support the search of partners for resolution or subsumption.

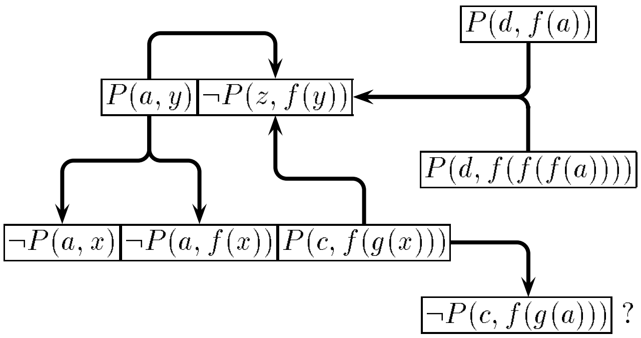


Figure 1.1: A Clause Graph

Parallelism. In general, a program working on a problem which is composed of several 'independent' parts can be divided into concurrent processes with each process working on one part of the problem. There are cases in which the performance of such a program can be improved with increasing concurrency. The possible *speedup* is limited by the degree of *dependence* inherently to the problem and by the parallel machine employed. In automated reasoning we can discover a large variety of such dependencies. They really complicate the investigation of possible improvements. Concurrent processes working on dependent parts of a problem have to solve these dependencies with *communication*. The more processes work on a problem, the more communication usually is required. In sum, the main task during the design of a distributed parallel system is to find a reasonable balance between the degree of parallelism and the amount of communication overhead.

1.2 Parallel Unit Resulting Resolution

We extend the unit resulting resolution rule to work on sets of substitutions. To this end we will introduce a so-called *clause graph*. The nodes of the clause graph correspond to literals in the clause set. Two literals are connected by a link if the literals are complementary and unifiable. In other words, connected literals are possible ur-resolution partners. Note that the clause graph does not change during the reasoning phase. An example of a clause graph is depicted in Figure 1.1. Consider for example the two literals $\neg P(z, f(y))$ and $P(c, f(g(x)))$. They are connected because they have opposite sign and they are unifiable by the unifying substitution $\sigma = \{z \mapsto c, y \mapsto g(x)\}$.

New ur-resolvents are represented by substitutions which are collected in sets of substitutions. These substitution sets are attached to nodes and links in the clause graph. We will show that substitution sets can be represented by indexes in a natural manner. In PURR indexes become the fundamental data structure instead of the usual clauses and literals. New ur-resolvents are exchanged among substitution sets in the form of indexes. Reasoning-based operations like subsumption and the computation of simultaneous unifiers are extended to set operations based on indexing techniques. Moreover, the unit resulting resolution rule is concurrently applied to different nuclei.

1.3 Structure of the Work

The first two Chapters 1 and 2 provide a general overview and introduce the main notions. In Chapter 3 we present a modified unit-resulting resolution rule on sets of substitutions. The main operations in the modified inference rule are presented as indexing methods in Chapter 4. The notions of parallelism in the field of logic and under practical issues are introduced in Chapter 5. The next Chapter 6 addresses implementational apsects of this work by presenting the central algorithms of Purr. A sequence of experiments with Purr is discussed in Chapter 7. The work is finished with a conclusion in Chapter 8.

Chapter 1: Introduction. The purpose of this chapter is to motivate the two main goals: The investigation of *indexing techniques* and the exploitation of *parallelism* in automated theorem proving. We briefly describe our modified ur-resolution scheme as the basis to study advanced indexing methods and parallelism in a theorem prover.

Chapter 2: Preliminaries. In the preliminary Chapter 2 we will focus on four different subjects. First, we present *rules*. They will be used throughout this work for defining functions in an elegant and simple way. In the second section the standard notions for *first-order logic* are introduced. As the standard notations for logic are used, readers familiar with this topic may skip the second section. Since the employed indexing technique relies on tree-like structures, we state some notions describing *graphs* and *trees* in the third section. Finally, we describe the way *algorithms* are presented in the fourth section.

Chapter 3: UR-Resolution. In the first section of Chapter 3 we briefly discuss *unit-resulting resolution* and reveal possible modifications in order to obtain a resolution scheme whose implementation can be supported by indexing methods and parallelism. Then we introduce a data structure called *clause graph* containing information about possible applications of ur-resolution. Finally, a modified ur-resolution rule working on *sets of substitutions* is introduced in the third section. The substitution sets are attached to vertices and links of the clause graph. The modified rule creates and exchanges substitution sets among the sets in the graph. The required reasoning-based operations can be implemented efficiently by indexing methods. The rule also can be applied to substitution sets in parallel.

Chapter 4: Indexing. First, we present an indexing technique called *substitution tree indexing* which can represent sets of idempotent substitutions. In the second section the four reasoning-based operations required by the modified ur-resolution rule are presented as indexing operations on substitution trees. In particular, we discuss the *subsumption* and *union* operations of two substitution sets, the *multi-merge* operation of arbitrary many substitution sets, and the *selection* operation of "lightest" substitutions.

Chapter 5: Parallelism. In addition to the presentation of the usual notions of parallelism in *logic* and *practice*, we also compare a selection of *parallel programming systems*. We conclude that the *Parallel Virtual Machine* (PVM) library currently seems to be the most convenient library to support the implementation of parallelism in our theorem prover.

Since we mainly discuss the notions of parallelism and well-known parallel programming libraries, readers familiar with these topics might want to skip this chapter.

Chapter 6: The Prover. In the first section detailed algorithms for the four indexing operations subsumption, union, multi-merge, and selection are presented. The implementational aspects of PURR are discussed in the second section. The system mainly performs three phases: The *preprocessing* for the creation and optimization of the clause graph, the *reasoning phase* involving the distributed processing of ur-resolution, and, finally, the *postprocessing* for the generation of a proof protocol. In the last section two important aspects of the implementation are presented: First, variables are maintained in *contexts* in order to represent variable bindings. Second, a *transformation* of substitution trees into a process-independent form.

Chapter 7: Experiments. In this chapter the results of experiments with PURR are presented. We compare the proof times obtained with different settings of PURR to the proof times of the sequential theorem prover OTTER. In the first section we use problem sets addressing the *indexing* methods in PURR. Experiments involving *parallelism* are presented in Section 2.

Chapter 8: Conclusion. We conclude that in many experiments PURR's advanced indexing operations achieve high inference rates and are able to handle large sets of inferences. Moreover, indexing supports the communication of concurrent processes which exchange sets of inferences. We discuss our experiences with parallelism in our theorem prover and point out the importance of decentralized distributed processing and of flexible control of granularity of a distributed theorem prover.

2

Preliminaries

In this chapter we introduce the notions of four different subjects. First, we present *rules.* They will be used throughout this work for defining functions in an elegant and simple way. In the second section the standard notions for *first-order logic* are introduced. This introduction is quite short, we merely present the notations needed in this thesis. As the standard notations for logic are used, readers familiar with this topic may skip the second section. Since a large part of the indexing techniques rely on tree-like structures, we state some notions describing *graphs* and *trees* in the third section. Finally, we describe the way *algorithms* are presented. The whole chapter is mainly based on the PhD thesis of Peter Graf [Gra96].

2.1 Rules

In this text most definitions are based on sequences of rules. Each rule consists of three parts. The first part contains a *pattern* and is written at the left side of the assignment. The second part occurs at the right side of the assignment and contains the resulting *value.* The third part is preceded by the keyword "if" and contains a *condition* under which the rule may be applied. Usually, a definition consists of more than a single rule, as illustrated by the following schema:

$$\begin{array}{rcl} \mathit{pattern}_1 & := & \mathit{value}_1 \text{ if } \mathit{condition}_1 \\ \vdots & & \vdots \\ \mathit{pattern}_n & := & \mathit{value}_n \text{ if } \mathit{condition}_n \end{array}$$

The rules are read top down. The ith rule is selected if all previous rules could not be applied, if the pattern represents the expression at issue, and if the condition is fulfilled. If a rule does not include a condition, the condition is evaluated to **true** by default. Note that a condition occurring in a rule that could not be satisfied is assumed to occur negated in all rules below.

In a rule that considers terms, x represents any variable, a represents any constant, f represents any non-constant function symbol, and t represents any term. Consider the following example:

$$\begin{array}{rcl} \mathsf{is\_a\_constant}(a) & := & \mathsf{true} \\ \mathsf{is\_a\_constant}(t) & := & \mathsf{false} \end{array}$$

On one hand, we see that the value of $\mathsf{is\_a\_constant}(b)$ is true because b is a constant and the symbol a in the rule represents any constant. On the other hand, $\mathsf{is\_a\_constant}(x)$ is false because x is a variable and therefore the first rule could not be applied.

2.2 First-Order Logic

2.2.1 Signature

The standard notions for first-order logic are used.

Definition 2.2.1 (Signature)
A signature $\Sigma := (\mathbf{V}, \mathbf{F}, \mathbf{P})$ consists of the following disjoint sets:

- $\mathbf{V}$ is a countable infinite set of variable symbols.
- $\mathbf{F}$ is a countable infinite set of function symbols. It is divided into the sets of n–place function symbols $\mathbf{F}_n$ $(n \in \mathbb{N}_0)$.
- $\mathbf{P}$ is a finite set of predicate symbols divided into the sets of n–place predicate symbols $\mathbf{P}_n$.

We will name variables x, y, z. We use the symbols f, g, h for functions and a, b, c for constants (0–place function symbols). Predicates are represented by P, Q, R.

Definition 2.2.2 (Special Symbols)
The following special symbols are available:

- The logical connectives $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall, \exists$
- The auxiliary symbols "(", ")", ", "

2.2.2 Terms, Literals, and Clauses

In first-order logic, constants and variables are used to denote objects. Predicates express properties of or relations between objects. Functions describe operations to be performed on objects. Constants, variables, and functions can be composed into terms, thus allowing arbitrarily complex object descriptions.

Definition 2.2.3 (Terms)
The set of *terms* $\mathbf{T}$ is the least set with $\mathbf{V} \subseteq \mathbf{T}$ and $f(t_1, \ldots, t_n) \in \mathbf{T}$ if $f \in \mathbf{F}_n$ and $t_1, \ldots, t_n \in \mathbf{T}$. The set of variables $\mathsf{VAR}(t)$ occurring in a term t is defined as

$$\begin{aligned} \mathsf{VAR}(x) &:= \{x\} \\ \mathsf{VAR}(a) &:= \emptyset \\ \mathsf{VAR}(f(t_1, \ldots, t_n)) &:= \bigcup_{1 \le i \le n} \mathsf{VAR}(t_i) \end{aligned}$$

A term t with $\mathsf{VAR}(t) = \emptyset$ is called *ground*. Additionally, the function top denotes the *top symbol* of a term.

$$\begin{aligned} \mathsf{top}(x) &:= x \\ \mathsf{top}(a) &:= a \\ \mathsf{top}(f(t_1, \ldots, t_n)) &:= f \end{aligned}$$

A term t is called *linear* if all variables of t occur exactly once in the term. The *depth* of a term is defined as

$$\begin{aligned} \mathsf{depth}(x) &:= 0 \\ \mathsf{depth}(a) &:= 0 \\ \mathsf{depth}(f(t_1, \ldots, t_n)) &:= 1 + \mathsf{max}\{\mathsf{depth}(t_1), \ldots, \mathsf{depth}(t_n)\} \end{aligned}$$

The *arity* of a term is defined as

$$\begin{aligned} \mathsf{arity}(x) &:= 0 \\ \mathsf{arity}(a) &:= 0 \\ \mathsf{arity}(f(t_1, \ldots, t_n)) &:= n \end{aligned}$$

A typical notational variant used in theorem proving is the clause form. A set of clauses represents a formula and each clause consists of a collection of literals.

Definition 2.2.4 (Atoms and Literals)
$P(t_1, \ldots, t_n)$ is an *atom* if $P \in \mathbf{P}_n$ and $t_1, \ldots, t_n \in \mathbf{T}$. Atoms and their negations are *literals*. A literal is called *negative* if it consists of an atom and a negation symbol. Otherwise it is called *positive*. Two literals $P(s_1, \ldots, s_n)$ and $\neg P(t_1, \ldots, t_n)$ are called *complementary*.

Definition 2.2.5 (Clauses)
A *clause* is a finite set of literals. The set is interpreted as the disjunction of the literals, with the whole clause being universally quantified over all variables occurring in it. A *unit* clause contains only one literal.

2.2.3 Positions

Throughout this work sequences are written in square brackets with [] denoting the empty sequence.

Definition 2.2.6 (Concatenation of Sequences)
The concatenation of sequences is performed by the function | with

$$\begin{aligned}[X]|[\,] &:= [X]\\ [X]|[i,Y] &:= [X,i]|[Y]\end{aligned}$$

For reasons of simplicity we often omit some of the square brackets and write X and $[X|i,Y]$ instead of $[X]$ and $[X]|[i,Y]$, respectively. In this thesis most sequences contain natural numbers separated by commas.

Definition 2.2.7 (Positions in a Term)
A *position* in a term is a finite sequence of natural numbers. The subterm of a term t *at position* p is denoted by t/p and defined as follows:

$$\begin{aligned}t\,/\,[\,] &:= t\\ f(t_1,\ldots,t_n)\,/\,[i|p] &:= t_i\,/\,p\end{aligned}$$

The set of *positions of the term* $t = f(t_1,\ldots,t_n)$ is defined by

$$\begin{aligned}\mathsf{O}(x) &:= \{[\,]\}\\ \mathsf{O}(a) &:= \{[\,]\}\\ \mathsf{O}(f(t_1,\ldots,t_n)) &:= \{[\,]\} \cup \bigcup_{p\in\mathsf{O}(t_i)} \{[i|p]\}\end{aligned}$$

For example, the term $g(a)$ occurs at the positions [1,1] and [2] in $f(g(g(a)),g(a))$. Accordingly, we have $f(g(g(a)),g(a))$ / [1,1] = $g(a)$ and $\mathsf{O}(h(a,g(b),x))$ = {[], [1], [2], [2,1], [3]}.

As positions are sequences of natural numbers, we can use the lexicographical extension of the natural ordering $>$ on natural numbers to sort positions.

Definition 2.2.8 (Total Ordering on Positions)
The lexicographical extension of the natural ordering $>$ on natural numbers is defined as follows:

$$\begin{aligned}p \overset{*}{>} [\,] \quad &\text{if} \quad p \neq [\,]\\ [i|p] \overset{*}{>} [j|q] \quad &\text{if} \quad i > j \;\vee\; (i = j \;\wedge\; p \overset{*}{>} q)\end{aligned}$$

For example, [1,2] $\overset{*}{>}$ [1,1], [2] $\overset{*}{>}$ [1,1], and [1,2,1] $\overset{*}{>}$ [1,2]. Note that $p \overset{*}{>} q$ if in the preorder traversal of the tree that represents a term containing both positions p and q the position q is visited before p.

2.2.4 Substitutions, Unification, and Matching

Definition 2.2.9 (Substitutions)
A *substitution* $\sigma : \mathbf{V} \to \mathbf{T}$ is an endomorphism on the term algebra such that the set $\{x \in \mathbf{V} \mid x\sigma \neq x\}$ is finite. The *domain* of a substitution is defined as

$$\mathsf{DOM}(\sigma) := \{x \in \mathbf{V} \mid x\sigma \neq x\}$$

The *codomain* of a substitution is defined as

$$\mathsf{COD}(\sigma) := \{x\sigma \mid x \in \mathsf{DOM}(\sigma)\}$$

The *image* of a substitution is defined as

$$\mathsf{IM}(\sigma) := \mathsf{VAR}(\mathsf{COD}(\sigma))$$

Since every substitution σ is uniquely determined by its effect on the variables of $\mathsf{DOM}(\sigma)$, it can be represented as a finite set of variable-term pairs $\{x_1 \mapsto x_1\sigma, \ldots, x_n \mapsto x_n\sigma\}$ where $\mathsf{DOM}(\sigma) = \{x_1, \ldots, x_n\}$. For example, the domain of the substitution $\sigma = \{x \mapsto f(a,b), y \mapsto g(z)\}$ is $\mathsf{DOM}(\sigma) = \{x, y\}$ and the codomain is $\mathsf{COD}(\sigma) = \{f(a,b), g(z)\}$. The set of variables introduced by σ is $\mathsf{IM}(\sigma) = \{z\}$.

Definition 2.2.10 (Composition of Substitutions)
Let $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ and $\tau = \{y_1 \mapsto t_1, \ldots, y_m \mapsto t_m\}$ be two substitutions. The *composition* $\sigma\tau$ of two substitutions is defined as

$$x(\sigma\tau) := (x\sigma)\tau$$

for all $x \in \mathbf{V}$. It can be computed as

$$\sigma\tau = \{x_1 \mapsto s_1\tau, \ldots, x_n \mapsto s_n\tau\} \cup \{y_i \mapsto t_i \mid y_i \in \mathsf{DOM}(\tau)\backslash\mathsf{DOM}(\sigma)\}$$

Consider, for instance, the substitutions $\sigma = \{z \mapsto f(x)\}$ and $\tau = \{x \mapsto a, y \mapsto c\}$. We have $\sigma\tau = \{z \mapsto f(a), x \mapsto a, y \mapsto c\}$. Note that the assignment $x \mapsto a$ is part of the composition, although it was applied to the variable x in $\mathsf{IM}(\sigma)$. The join of σ and τ defined below will not contain $x \mapsto a$ anymore.

Definition 2.2.11 (Join of Substitutions)
Let $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ and $\tau = \{y_1 \mapsto t_1, \ldots, y_m \mapsto t_m\}$ be two substitutions. The *join* of the substitutions σ and τ is defined as

$$\sigma \bullet \tau := \{x_1 \mapsto s_1\tau, \ldots, x_n \mapsto s_n\tau\} \cup \{y_i \mapsto t_i \mid y_i \in \mathsf{DOM}(\tau)\backslash\mathsf{IM}(\sigma)\}$$

Obviously, for $\sigma = \{z \mapsto f(x)\}$ and $\tau = \{x \mapsto a, y \mapsto c\}$ we have $\sigma \bullet \tau = \{z \mapsto f(a), y \mapsto c\}$. The join of substitutions is closely related to the composition. The only difference is that, contrary to the composition, assignments that could be applied are not contained in the result of the join. In Section 4.2.1.2 the join of substitutions will be needed to define deletion in substitution trees.

Definition 2.2.12 (Restriction)
Let σ be a substitution and $U \subseteq \mathbf{V}$ a set of variables. The *restriction* $\sigma|_U$ is the substitution with $\mathsf{DOM}(\sigma|_U) \subseteq U$ which agrees with σ on U.

Definition 2.2.13 (Idempotent Substitution)
The substitution σ is called *idempotent* iff $\sigma\sigma = \sigma$.

For idempotent substitutions we have $\mathsf{DOM}(\sigma) \cap \mathsf{IM}(\sigma) = \emptyset$.

Definition 2.2.14 (Variant Terms)
A substitution ρ is called a *renaming* if it is injective on $\mathsf{DOM}(\rho)$ and if the codomain $\mathsf{COD}(\rho)$ only contains variables. Two terms s and t are called *variants* if a renaming ρ exists such that $s\rho = t$.

Definition 2.2.15 (Matcher)
A substitution μ is called a *matcher* from term s to term t if $s\mu = t$. In this case s is called a *generalization* of t and t is called an *instance* of s.

Definition 2.2.16 (Unifiable Terms, Most General Unifier)
Two terms s and t are called *unifiable* if and only if a substitution σ exists such that $s\sigma = t\sigma$. In this case the substitution σ is called a *unifier* of s and t. A unifier σ is called *most general unifier* (mgu) if for every unifier λ of s and t a substitution τ exists such that $\sigma\tau = \lambda$. We define the function $\mathsf{mgu}(s, t)$ to compute the most general unifier of the terms s and t. The extention of $\mathsf{mgu}()$ to compute the most general unifier of atoms is straightforward.

For example, the terms $f(a, y)$ and $f(x, b)$ are unifiable and the mgu is $\{x \mapsto a, y \mapsto b\}$. Note that the most general unifier for two terms is unique up to variable renamings if theories are not involved. Terms may be *non-unifiable* for different reasons. *Clashes* occur when two non-variable symbols occurring at identical positions in the two terms are not identical. A clash is called *direct* if it can be detected without considering partial substitutions. For example, a direct clash is detected when unifying $f(a, x)$ and $f(b, y)$. The detection of indirect clashes requires the consideration of partial substitutions. The unification of $f(x, x)$ and $f(a, b)$ fails because the variable x can only be bound either to the constant a or to the constant b. Failures resulting from *occur-checks* also take partial substitutions into consideration. For example, the occur-check detects the failure when unifying $f(x, x)$ and $f(y, g(y))$ because a unifier would have to contain the binding $y \mapsto g(y)$.

Definition 2.2.17 (Merge of Substitutions)
Let σ and τ be two idempotent substitutions. A *unifier* for σ and τ is a substitution ρ such that $\sigma\rho = \tau\rho$. A unifier ρ of two substitutions is called *most general* if for every unifier λ of σ and τ a substitution μ exists such that $\rho\mu = \lambda$. The substitutions σ and τ are *compatible* if they have a most general unifier ρ. In this case the *merge* of σ and τ is defined as

$$\sigma \times \tau := (\sigma\rho)|_{DOM(\sigma)\cup DOM(\tau)}$$

Merging two substitutions corresponds to calculating the most general common instance. The domain and the codomain of the resulting substitution are computed as follows:

$$\begin{aligned} \mathsf{DOM}(\sigma \times \tau) &= \mathsf{DOM}(\sigma) \cup \mathsf{DOM}(\tau) \\ \mathsf{COD}(\sigma \times \tau) &= (\mathsf{DOM}(\sigma \times \tau))\mathsf{mgu}(\mathsf{DOM}(\sigma \times \tau)\sigma, \mathsf{DOM}(\sigma \times \tau)\tau) \end{aligned}$$

For the two substitutions $\sigma = \{x \mapsto f(a, u), y \mapsto c\}$ and $\tau = \{x \mapsto f(v, b)\}$ we have $\sigma \times \tau = \{x \mapsto f(a, b), y \mapsto c\}$.

Definition 2.2.18 (Merge of Substitution Sets)
Let Σ, Θ be two substitution sets. We extend the merge operator $\times$ to a generic merge operator .

$$\Sigma \times \Theta := \{\rho | \rho = \mu \times \lambda, \mu \in \Sigma, \lambda \in \Theta\}$$

The extention to an arbitrary number of arguments is straightforward.

2.2.5 Normalization

Finally, we introduce the notion of a normalized term $\overline{s}$ for a term s. Normalization renames the variables of terms s and t in such a way that $\overline{s} = \overline{t}$ holds for terms equal modulo variable renaming.

The purpose of normalization is to rename terms and substitutions before they are inserted into a term index in order to enable more sharing of common symbols in the index.

The variables in a term are renamed to so-called *indicator variables*, which are denoted by $*_i$. The set $\mathbf{V}^*$ of indicator variables is a subset of $\mathbf{V}$.

Definition 2.2.19 (Normalization of Terms)
Let $s = f(s_1, \ldots, s_n)$ be a term. The set of first occurrences of variables in s is defined as

$$\mathsf{O}_{first}(s) := \{p \mid p \in \mathsf{O}(s), s/p \in \mathbf{V}, \forall q \in \mathsf{O}(s), p \overset{*}{>} q.\ s/q \neq s/p\}$$

Let $\mathsf{O}_{first}(s) = \{p_1, \ldots, p_m\}$ and $p_j \overset{*}{>} p_i$ for $1 \leq i < j \leq m$. Then the substitution $\sigma = \{s/p_1 \mapsto *_1, \ldots, s/p_m \mapsto *_m\}$ with $*_i \in \mathbf{V}^*$ is called *normalization* and

$$\overline{s} := s\sigma$$

For example, $\overline{f(x)} = \overline{f(y)} = f(*_1)$ and $\overline{h(x,x,y)} = \overline{h(z,z,x)} = h(*_1,\ *_1,\ *_2)$. The next definition extends the normalization of terms to the normalization of substitutions.

Definition 2.2.20 (Normalization of Substitutions)
Let $\sigma = \{x_1 \mapsto t_1,\ \ldots,\ x_n \mapsto t_n\}$ be a substitution and $t = \overline{f_n(t_1, \ldots, t_n)}$ with $f_n \in F_n$. Additionally, let $<$ be a fixed total ordering on variables and $x_1 < \ldots < x_n$. The *normalized substitution* $\overline{\sigma}$ is defined as

$$\overline{\sigma} := \{x_1 \mapsto t/1, \ldots, x_n \mapsto t/n\}$$

For example, if $\sigma = \{x \mapsto f(u,v), y \mapsto f(a,v)\}$ and $x < y$, then $\overline{\sigma} = \{x \mapsto f(*_1, *_2), y \mapsto f(a, *_2)\}$. However, if we chose $y < x$, the normalization of σ would be $\overline{\sigma} = \{x \mapsto f(*_2, *_1), y \mapsto f(a, *_1)\}$.

2.3 Graphs and Trees

Definition 2.3.1 (Graph)
A *graph* $G = (V, E)$ consists of a finite, nonempty set of *vertices* V and a set of *edges* E. If the edges are ordered pairs (v, w) of vertices, then the graph is said to be *directed*; v is called the *tail* and w the *head* of the edge (v, w). If the edges are unordered pairs (sets) of distinct vertices, also denoted by (v, w), then the graph is said to be *undirected.* A directed graph is said to be *labeled* if the edges are ordered triples (v,Label, w) of two vertices and an arbitrary label.

Definition 2.3.2 (Acyclic Graph)
A *path* in a graph is a sequence of edges of the form $(v_1, v_2), \ldots, (v_{n-1}, v_n)$. A path is *simple* if all edges and vertices, except possibly the first and last vertices, are distinct. A *cycle* is a simple path which begins and ends at the same vertex. An *acyclic graph* does not contain cycles.

Definition 2.3.3 (Tree)
A *tree* $T = (V, E)$ is a directed acyclic graph having the following properties:

1. There is exactly one vertex that no edges enter. This vertex is called *root*.
2. Every vertex except the root has one entering edge.
3. There is a unique path from the root to each vertex.

If (v, w) is in E, then v is called the *father* of w, and w is the *son* of v. If there is a path from v to w, then v is an *ancestor* of w and w is a *descendant* of v. A vertex with no sons is called *leaf*. A vertex w and all its descendants are called a *subtree* of v. The vertex w is called the *root* of that subtree. The empty tree is denoted by ε.

We introduce the following notions for a tree $T = (V, E)$: The set of vertices of a tree is denoted by $\mathsf{nodes}(T) := V$. Additionally, $\mathsf{root}(T)$ denotes the root of the tree. Subtrees of the tree are denoted by $\mathsf{sons}(T) := \{v \mid \mathsf{father}(v) = \mathsf{root}(T)\}$.

Definition 2.3.4 (Ordered Tree)
An *ordered tree* is a tree in which the sons of each vertex are ordered. When drawing an ordered tree we assume that the sons of each vertex are ordered from left to right.

2.4 Algorithms

We use a standardized notation for presenting algorithms. An example algorithm for computing the value of $n! = 1 \cdot \ldots \cdot n$ is shown in Figure 2.1. Predefined commands like **if** and **then** are written **boldface**. The scope of **if**, **else**, and **forall** commands is defined by indention. All variables like n and *RESULT* are local. If global variables are needed, a comment will tell this. The result of a function is the value returned by the command **return** in line 8. Formal parameters of an algorithm may be defined together with their type.

```
1  algorithm fac(int n)
2   begin
3      if $n = 1$ then
4         $RESULT = 1$
5      else
6         ⟨ This is a comment. ⟩
7         $RESULT = n * \mathtt{fac}(n - 1)$
8      return $RESULT$
9   end
```

Figure 2.1: Presentation of Algorithms

3

UR-Resolution

The *Unit Resulting Resolution* was introduced by McCharen, Overbeek, and Wos [MOW76a]. We shall employ this inference rule for the investigation of indexing techniques and distributed processing in automated reasoning. To this end we extend the unit resulting resolution rule to work on sets of substitutions. In order to detach the inference rule from the usual notion of clauses, we introduce a clause graph on the initial clause set. The clause graph provides precomputed information about the potential application of the modified unit resulting resolution.

In Chapter 4 we will show that substitution sets can be represented by indexes in a natural manner. In Purr these indexes become the fundamental data structure instead of the usual clauses and literals. Thus in this work an index efficiently represents a substitution set. The main operations like subsumption and the computation of simultaneous unifiers are extended to set operations based on indexing techniques. Moreover, the modified ur-resolution rule is concurrently applied to different nuclei of the clause graph. New sets of conclusions are exchanged among parallel processes in the form of indexes.

In the first section we briefly discuss the unit resulting resolution principle and motivate possible modifications. The clause graph is introduced in the second section. Finally, we discuss our extention of unit resulting resolution to an inference rule working on substitution sets in the third section.

3.1 Unit Resulting Resolution

The ur-resolution rule is applied to a set of m unit clauses $\{K_1\}, \ldots, \{K_m\}$ and a single nucleus $\{L_1, \ldots, L_{m+1}\}$ consisting of $m+1$ literals. If there is a simultaneous complementary unifier μ for all m pairs of literals K_i, L_i, then $\{L_{m+1}\}\mu$ is the unit resulting resolvent. In the general presentation of the rules all pairs of literals K_i, L_i are assumed to be complementary, i.e. to have opposite signs. The computation of the simultaneous unifier ignores the signs of the literals; $|K|$

denotes the atom contained in the literal K.

$$\begin{array}{ll} \{K_1\} & \\ \vdots & \\ \{K_m\} & \\ \{L_1, \ldots, L_m, L_{m+1}\} & \exists\mu.\ \mu = \mathsf{mgu}([|K_1|, \ldots, |K_m|], [|L_1|, \ldots, |L_m|]) \\ \hline \{L_{m+1}\}\mu & \end{array}$$

As an example for unit resulting resolution consider the non-unit clause

$$\{\neg\mathtt{MARRIED}(x, y), \neg\mathtt{MOTHER}(x, z), \mathtt{FATHER}(y, z)\}$$

and the two unit clauses

$$\begin{array}{l} \{\mathtt{MARRIED}(\mathit{lisa}, \mathit{joe})\} \\ \{\neg\mathtt{FATHER}(\mathit{joe}, \mathit{pete})\} \end{array}$$

UR-Resolution yields the clause

$$\{\neg\mathtt{MOTHER}(\mathit{lisa}, \mathit{pete})\}$$

Unit resulting resolution can produce both negative and positive resolvents. The ur-inference rule is refutation complete on the class of unary refutation complete clause sets. *Horn* clauses are an important subset of this class.

Exploiting that ur-resolvents $\{L_{m+1}\}\mu$ are unit clauses, the resolvents can be represented as substitutions μ as long as the according literal L_{m+1} is known. Consider a modified resolution scheme which delays the application of simultaneous unifiers μ_i of former resolution steps until the computation of the current simultaneous unifier μ. Note that unit clauses of the initial clause set have empty substitutions μ_i.

$$\begin{array}{ll} \{K_1\}\mu_1 & \\ \vdots & \\ \{K_m\}\mu_m & \\ \{L_1, \ldots, L_m, L_{m+1}\} & \exists\mu.\ \mu = \mathsf{mgu}([|K_1\mu_1|, \ldots, |K_m\mu_m|], [|L_1|, \ldots, |L_m|]) \\ \hline \{L_{m+1}\}\mu & \end{array}$$

In this resolution scheme, the literals K_i are always literals of the initial clause set. The initial clause set also determines the possible combinations of literals K_i and L_i. If the literals K_i and L_i are complementary and unifiable, then an instance $K_i\mu_i$ might still be unifiable with L_i. If two complementary literals K_i and L_i are not unifiable, then no instance $K_i\mu_i$ will ever be unifiable with L_i.

Pairwise complementary and unifiable literals of the initial clause set can be computed in advance. The unifiability of literals in a clause set is represented by a clause graph. The nodes of the clause graph correspond to the literals of the initial clause set. The links denote unifiability and thus determine the application of ur-resolution. We attach sets of ur-resolvents represented as substitutions to the according nodes in the clause graph. The refutation process thus consists of the creation and the exchange of substitutions via links in a clause graph.

In the next section we develop a convenient graph structure containing information about the unifiability of literals. The graph structure will serve as a base for a modified unit-resulting resolution rule that will solely work on substitutions.

3.2 Clause Graph

3.2.1 Undirected Clause Graph

In a clause set some unit clauses or ur-resolvents are potential ur-resolution partners for a certain nucleus. These relations do not change during subsequent applications of ur-resolution. One means to represent this information explicitly is a *clause graph.* Each vertex of such a clause graph denotes a literal of the clause set. Two vertices are linked if the associated literals are complementary and unifiable. This definition yields a so-called *undirected clause graph.* For the sake of simplicity we also refer to the term "vertex v corresponding to literal L" as "literal v" if the correspondence is non-ambiguous. Consider the following clause set:

$$\begin{array}{l} \{Q(a)\} \\ \{P(a)\} \\ \{\neg Q(x), \neg P(a), P(y)\} \\ \{\neg P(d), \neg P(b)\} \\ \{\neg P(c)\} \end{array}$$

The corresponding undirected clause graph is depicted in Figure 3.1. The vertices, whose corresponding literals form a non-unit clause, are presented in groups. Each pair of complementary and unifiable literals is connected by a link.

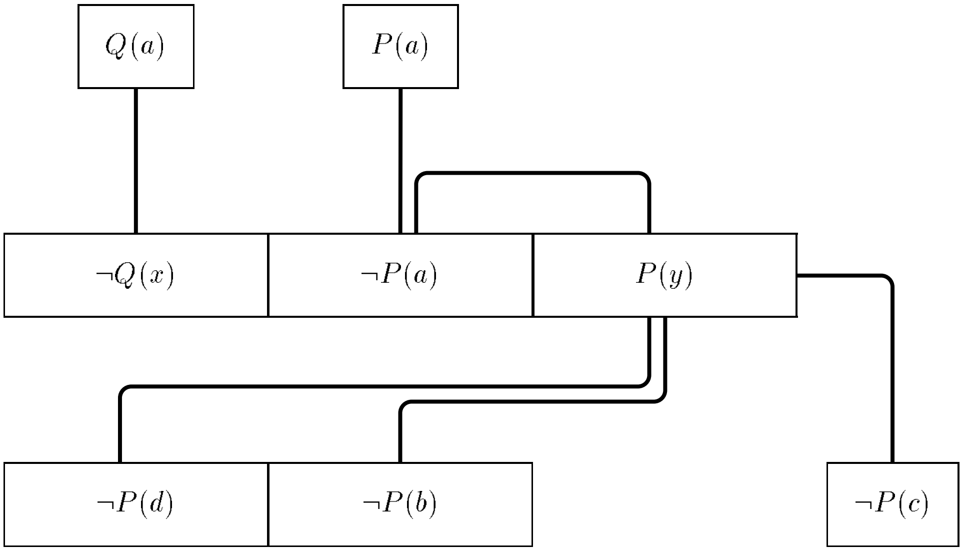


Figure 3.1: Undirected Clause Graph

3.2.2 Directed Clause Graph

An undirected link indicates a possible application of ur-resolution. However, nothing is said about which literals are regarded as electrons and which literals play the role of a literal of the nucleus. Moreover, some links seem to be redundant for the refutation process. For instance, in order to find a refutation of the unit clause $\{\neg P(c)\}$ in Figure 3.1, the clause $\{\neg P(d), \neg P(b)\}$ is not needed.

In order to avoid superfluous links in a clause graph we introduce directed links and a classification of clauses. The direction of a link determines the type of the connected literals. The originating literal is called the *sender literal.* This literal corresponds to an electron which is either a unit clause in the initial clause set or a ur-resolvent of a nucleus. The target literal is referred to as the *receiver literal* which is part of a nucleus.

Chang and Slagle [CS79] introduced a classification of the clause set inventing the *Set-of-Support* (SOS) strategy. Here the clause set is divided into two disjoint subsets. One subset contains the clauses originating from the axioms, where the other subset contains the clauses from the negated theorem. The idea is that the axiom set itself is not unsatisfiable. To find a refutation the clauses that form the theorem are also required. The SOS strategy forbids resolvents inferred by solely using clauses from the axioms.

According to the SOS strategy some clauses in a clause graph are marked as so-called *query clauses.* As a consequence some reasoning possibilities need not to be considered. A query clause usually corresponds to a theorem in the clause set. The creation of a clause graph is a recursive process considering the literals of the query clauses as receiver literals. If an according sender literal belongs to a nucleus, then the remaining literals of the nucleus have to be considered as receiver literals as well.

In addition to the SOS strategy, the sign of the literals might be used as a restriction on directed links. For example, one could only allow links from positive to negative literals. As a consequence every ur-resolvent would be positive. This restriction corresponds to the positive and negative versions of *hyperresolution* introduced by John Alan Robinson [Rob65].

Note that a directed clause graph is a generalization of the undirected approach if all clauses are marked as query clauses. In a directed clause graph with all clauses marked as query clauses, each linked pair of vertices has two links (v, w) and (w, v).

In Figure 3.2 the unit clause $\{\neg P(c)\}$ is marked as a query clause by the question mark. The literal $P(y)$ of the triple clause $\{\neg Q(x), \neg P(a), P(y)\}$ is an appropriate partner for the query clause. The remaining literals $\neg Q(x)$ and $\neg P(a)$ match the two positive unit clauses $Q(a)$ and $P(a)$ and the $P(y)$ itself (indicated by a so-called *internal link*). Such an internal link connects literals of a single clause. The clause $\{\neg P(d), \neg P(b)\}$ is omitted, because there is no possible connection with other clauses. Thus this clause is not needed for the refutation of the query clause $\neg P(c)$.

3.2.3 Labeled Directed Clause Graph

A directed clause graph mirrors ur-resolution possibilities. The direction of links distinguishes connected literals as electron and part of a nucleus. Sender literals are considered as electrons whereas receiver literals belong to nuclei. The representation of ur-resolvents as substitutions also requires additional information about the unifiability of linked literals. To meet these requirements we introduce *labeled links* containing *splitted most general unifiers* of linked literals. As a result we obtain a *labeled directed clause graph.*

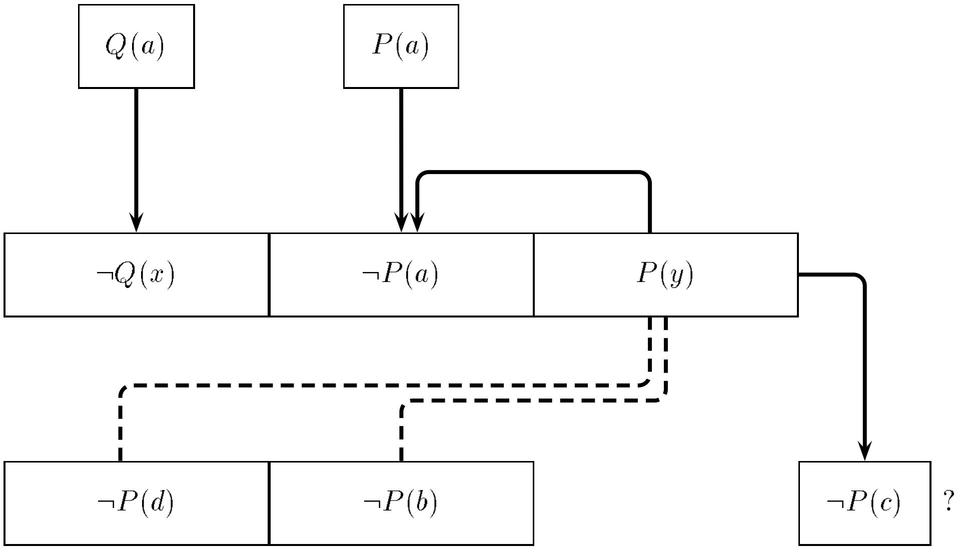


Figure 3.2: Directed Clause Graph

3.2.3.1 Examples

Consider the following fragment of a clause set:

$$\begin{array}{l} \{R(a)\} \\ \{R(b)\} \\ \{\neg R(x), P(x,a)\} \\ \{\neg P(b,y), Q(y)\} \\ \dots \end{array}$$

Assume that the literal $Q(y)$ has been chosen as a sender literal by some query clauses. The corresponding directed clause graph is depicted in Figure 3.3.

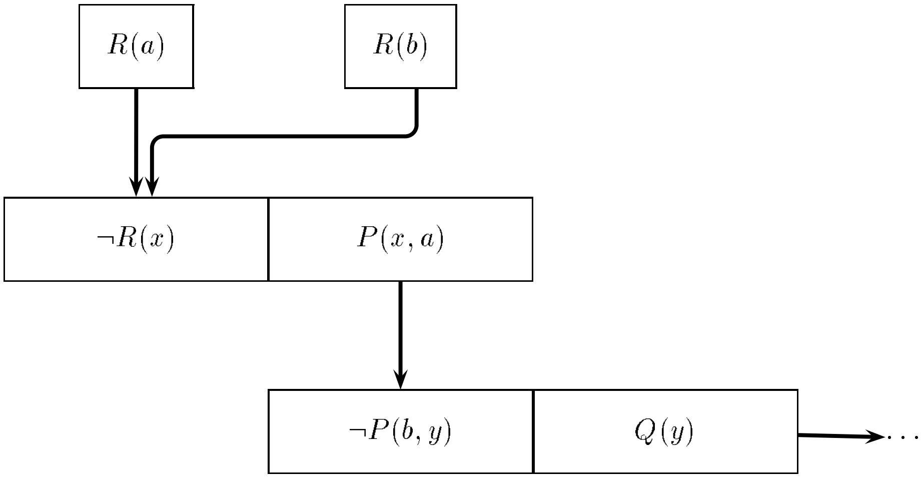


Figure 3.3: Directed Clause Graph

Two ur-resolvents $\{P(a,a)\}$ and $\{P(b,a)\}$ might be inferred by using the clauses $\{R(a)\}$, $\{R(b)\}$, and the nucleus $\{\neg R(x), P(x,a)\}$. The ur-resolvent $\{P(b,a)\}$ can be used as a new electron with the literal $\neg P(b,y)$ of the nucleus $\{\neg P(b,y), Q(y)\}$ to resolve $\{Q(a)\}$. The other ur-resolvent $\{P(a,a)\}$ is not compatible with $\neg P(b,y)$.

In order to check if a specific instance of a sender literal is unifiable with the according receiver literal we consider the most general unifier of the two literals. In the example, the most general unifier of the sender literal $P(x,a)$ and the receiver literal $\neg P(b,y)$ is the substitution $\sigma = \{x \mapsto b, y \mapsto a\}$. The unifiability can be checked if we also consider ur-resolvents represented as substitutions. These substitutions correspond to the simultaneous unifiers of ur-resolution. The ur-resolvent $\{P(a,a)\}$ is represented by the substitution $\mu_1 = \{x \mapsto a\}$. The ur-resolvent $\{P(b,a)\}$ is represented by the substitution $\mu_2 = \{x \mapsto b\}$. Obviously, only the substitution μ_2 is unifiable with the substitution σ.

We observe that only assignments of σ to variables occurring in the sender literal are needed to check if a specific instance of the sender literal is unifiable with the receiver literal. In the example that is the assignment $\{x \mapsto b\}$ of σ which has to be checked for unifiability with the substitutions μ_1 and μ_2.

The assignments to variables occurring in the receiver literal are also needed in order to create a simultaneous unifier in subsequent ur-resolution steps. In the example, the assignment $\{y \mapsto a\}$ of σ is used to instantiate the receiver literal $\neg P(b,y)$.

The next example contains clauses with more complex terms. Consider the clause set:

$\{Q(x, f(y))\}$
$\{P(f(f(x)), x, a)\}$
$\{\neg Q(f(x), x), \neg P(f(x), f(y), a), P(x, f(y), z)\}$
$\{\neg P(f(b), f(b), c)\}$ is a query clause

Note that all clauses are considered variable disjoint. The corresponding clause graph is depicted in Figure 3.4. Each link in the graph is labeled with a so-called *splitted most general unifier* of the connected literals. The unifier is divided into two substitutions τ and σ. The substitution τ is called the *test substitution.* The substitution σ is referred to as the *send substitution.* Note that a test substitution only considers variables of the sender literal whereas a send substitution only contains variables of the receiver literal in its domain.

The most general unifier of the connected literals is splitted in order to support two essential operations: The *test unification* and the *send instantiation.*

- Every instantiation μ of a sender literal has to be tested for unifiability with the connected receiver literal. This test corresponds to the selection of a ur-resolvent as an electron. We also refer to this operation as the *test unification.* The test unifies the instantiation μ and the test substitution τ of the corresponding link.

- The resulting unifier of a successful test unification is applied to the send substitution σ. We refer to this operation as the *send instantiation.* The result of the send instantiation is a substitution μ' which instantiates the receiver literal. As the receiver literal belongs to a nucleus, the according instantiation μ' is used for the computation of simultaneous unifiers together with instantiations of the remaining literals in order to perform a unit resulting resolution step.

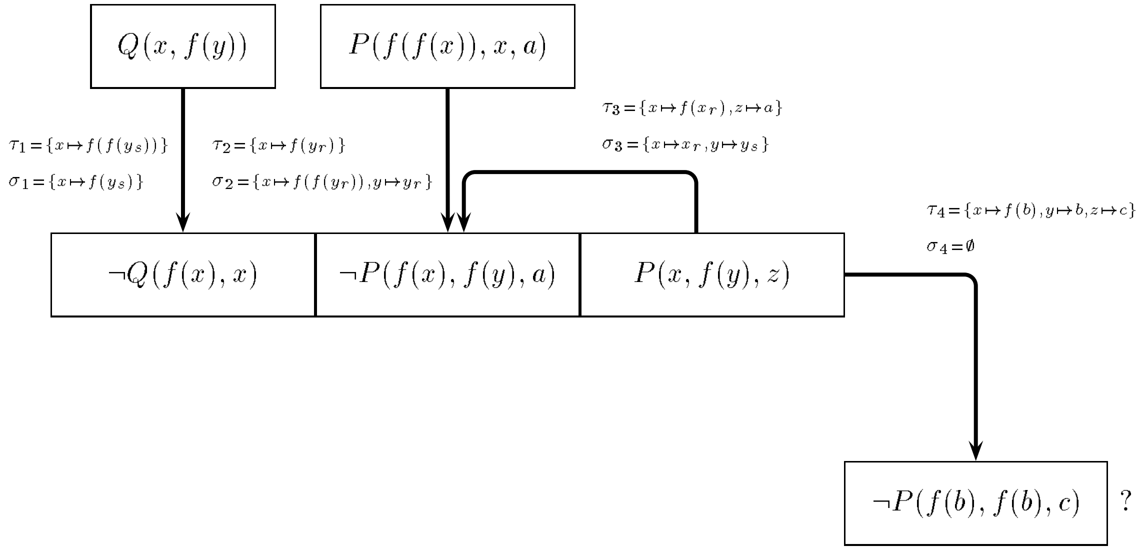

$Q(x, f(y))$
$P(f(f(x)), x, a)$
$\tau_3 = \{x \mapsto f(x_r), z \mapsto a\}$
$\sigma_3 = \{x \mapsto x_r, y \mapsto y_s\}$
$\tau_1 = \{x \mapsto f(f(y_s))\}$
$\sigma_1 = \{x \mapsto f(y_s)\}$
$\tau_2 = \{x \mapsto f(y_r)\}$
$\sigma_2 = \{x \mapsto f(f(y_r)), y \mapsto y_r\}$
$\tau_4 = \{x \mapsto f(b), y \mapsto b, z \mapsto c\}$
$\sigma_4 = \emptyset$
$\neg Q(f(x), x)$
$\neg P(f(x), f(y), a)$
$P(x, f(y), z)$
$\neg P(f(b), f(b), c)$
?

Figure 3.4: Labeled Directed Clause Graph with Test and Send Substitutions

Consider the link labeled with τ_4 and σ_4. Given the test substitution τ_4 every instance of the variable x in the sender literal $\{P(x, f(y), z)\}$ has to be of the form $f(b)$ in order to be unifiable with the query clause $\{\neg P(f(b), f(b), c)\}$. Moreover, instances of y are restricted to the constant b and instances of z are restricted to c. The empty send substitution σ_4 indicates that the receiver literal does not contain any variables. Thus an instantiation μ' will be empty after a successful test unification.

At the link labeled with τ_1 and σ_1 variables are involved in both the sender and the receiver literal. Note that a test substitution τ actually is not required for unit clauses since there are no special instantiations for unit clauses. In this case, the send substitution σ_1 suffices for the correct instantiation of the receiver literal. The same applies to the link labeled with τ_2 and σ_2.

The internal link labeled with τ_3 and σ_3 shows a new, complex situation. In general, variables of literals of one clause are not disjoint, i.e. the variable x occurring in the sender literal $P(x, f(y), z)$ and the receiver literal $\neg P(f(x), f(y), a)$ denotes a single variable. However, variables like x and y have to be distinguished in the test and send substitution if the literals are connected with an internal link. Therefore, variables of the sender literal are indexed by "s" in the codomain of the test and send substitution. Variables of the receiver literal are indexed by "r".

The test substitution τ_3 restricts all instances of x to the term $f(x_r)$, where the variable x_r denotes the variable x in the receiver literal. The corresponding assignment $\{x \mapsto x_r\}$ of σ_3 achieves the correct instantiation of the receiver literal. The variable x_r refers to the same variable in the codomain of τ_3. If x_r is bound to a term t while testing an instance of the sender literal against τ_3, the term t also becomes an instance of the variable x in the receiver literal.

Now, consider the assignment $\{y \mapsto y_s\}$ in σ_3. If an instance of the sender literal instantiates the variable y, then this instance also appears at the variable y in the receiver literal. The domain variable y of $\{y \mapsto y_s\}$ belongs to the receiver literal, whereas the codomain variable y_s belongs to the sender literal.

Note that an additional conversion of variables is needed for the correct instantiation of the send substitutions. This problem arises when variables in the codomain of the send substitution are not instantiated. We shall discuss this issue in more detail in Chapter 6.

3.2.4 Definitions

We finally summarize the previous discussion by two informal definitions. In particular, we define the splitted most general unifier of linked literals and the labeled directed clause graph. The clause graph will be used in our ur-resolution scheme on substitution sets which is discussed in the next section.

Definition 3.2.1 (Splitted Most General Unifier)
Let L_s be a sender literal and let L_r be a receiver literal. The literals are not necessarily variable disjoint. The Splitted Most General Unifier $\texttt{split\_mgu}(L_s, L_r)$ is defined as follows:

$$(\tau, \sigma) = \texttt{split\_mgu}(L_s, L_r) \quad :\Longleftrightarrow \quad \begin{array}{l} |L_s| \text{ and } |L_r| \text{ are unifiable and} \\ |L_s|\tau = |L_r|\sigma \text{ and} \\ (\tau, \sigma) \text{ is most general.} \end{array}$$

Note that this definition does not address the formerly presented variable renaming with indices "s" and "r". However, we will discuss this issue in Chapter 6.

We summarize the main ideas behind the test and the send substitutions as follows:

- The test substitution τ is used to check if a specific instantiation μ of ur-resolvent K is unifiable with a connected receiver literal L.
- The send substitution σ is used as a template for the computation of an instantiation μ' of the receiver literal L considering $K\mu$ as an electron.

Finally, we present the definition of a directed clause graph labeled with test and send substitutions. The later definition of the ur-resolution rule on substitution sets is based on this clause graph. An example of a labeled directed clause graph is depicted in Figure 3.4.

Definition 3.2.2 (Labeled Directed Clause Graph (LDCG))
Let $S = \{C_1, \ldots, C_m\}$ be a non-empty clause set. A non-empty subset of the clauses of S is marked as query clauses. A labeled directed clause graph on S

$$CG_S := (V, E)$$

is a labeled directed graph. The following conditions hold:

- Each literal of the clause set S corresponds to exactly one vertex in V. The function $\mathsf{literal}$ maps each vertex $v \in V$ to exactly one literal $L \in C_i$.
- The function clause maps a vertex $v \in V$ to a set of vertices $\{v, v_1, \ldots, v_n\}$ which correspond to the literals of the according clause of v.
- $\mathsf{query}(S) \subseteq V$ denotes the set of vertices whose corresponding literals are considered as literals of the query clauses in S.

- E is the set of labeled directed edges with $(v, (\tau, \sigma), w) \in E$ if the corresponding literals of v and w are complementary and unifiable with $(\tau, \sigma) = \texttt{split\_mgu}(\mathsf{literal}(v), \mathsf{literal}(w))$. Furthermore, w corresponds to a literal in a query clause or a query clause is reachable via a sequence of labeled directed edges.

A detailed algorithm for the creation and optimization of labeled directed clause graphs will be discussed in Chapter 6.

3.3 UR-Resolution on Substitution Sets

The ur-resolution scheme on substitution sets exploits the labeled directed clause graph. Resolvents are represented as substitutions. Each substitution belongs to a certain literal in the clause set, i.e. a certain vertex in the clause graph. Resolvents are collected in substitution sets which are attached to vertices and links in the clause graph. Substitutions are exchanged between these substitution sets via links in the clause graph. In the next section we introduce a modified version of ur-resolution that works on substitution sets.

3.3.1 Adapting UR-Resolution to Substitution Sets

On page 14 we have introduced ur-resolution. We now present a first modified ur-resolution rule which solely works with substitutions instead of clauses and literals. To this end we create the according LDCG.

We use substitutions to represent instances of literals in the graph. It is important to keep in mind that a single substitution is always associated to a certain literal. In the following ur-resolution rule we associate substitutions μ_i and μ to sender literals in the graph.

The pairs of an electron $K_i\mu_i$ and a literal L_i in the nucleus are replaced by the according splitted unifier (τ_i, σ_i) contained in the graph. The electrons $K_i\mu_i$ are represented by substitutions μ_i which belong to literals K_i in the graph. The function `apply` performs the test unification and the send instantiation which correspond to the pairwise unification of electrons $K_i\mu_i$ and literals L_i in the nucleus and the creation of the according unifier (s. Def. 3.3.2). We currently assume that all test unifications succeed. In other words, the electrons $K_i\mu_i$ and literals L_i are pairwise unifiable. The resulting unifiers have to be unified in order to produce the simultaneous unifier μ which is associated to the literal L_{m+1}.

$$\frac{\begin{array}{ll} \mu_1 & \\ \vdots & \\ \mu_m & \\ (\tau_1, \sigma_1), \ldots, (\tau_m, \sigma_m) & \qquad \exists \mu.\ \mu = \texttt{apply}(\{\mu_1\}, \tau_1, \sigma_1) \times \ldots \times \texttt{apply}(\{\mu_m\}, \tau_m, \sigma_m) \end{array}}{\mu}$$

In practice, we cannot assume that all pairs of literals $K_i\mu_i$ and L_i are compatible. The idea of the next ur-resolution rule is to associate every substitution μ_i and μ to receiver literals in the graph. The substitutions represent the result of already successful test unifications and send instantiations.

If the substitutions μ_i belong to the literals L_i of the nucleus, these substitutions μ_i can directly be unified to a simultaneous unifier μ. Finally, the substitution μ is tested with the splitted unifier (τ_{m+1},σ_{m+1}) of one outgoing link of the literal L_{m+1}. If the test unification succeeds, the resulting substitution belongs to the according receiver literal of the outgoing link. This substitution might be used as a new electron in subsequent applications of the rule.

$$\frac{\begin{array}{ll}\mu_1 & \\ \vdots & \\ \mu_m & \\ (\tau_{m+1},\sigma_{m+1}) & \qquad \exists\mu.\ \mu=\mu_1\times\ldots\times\mu_m\end{array}}{\texttt{apply}(\{\mu\},\tau_{m+1},\sigma_{m+1})}$$

Instead of working with single substitutions μ_i, we could also process substitution sets Σ_i. The modified ur-resolution rule also produces a set of simultaneous unifiers.

$$\frac{\begin{array}{ll}\Sigma_1 & \\ \vdots & \\ \Sigma_m & \\ (\tau_{m+1},\sigma_{m+1}) & \qquad \exists\Sigma\neq\emptyset.\ \Sigma=\Sigma_1\times\ldots\times\Sigma_m\end{array}}{\texttt{apply}(\Sigma,\tau_{m+1},\sigma_{m+1})}$$

In practice, we only consider one substitution set Σ_i of new ur-resolvents. The remaining substitution sets are considered as sets of ur-resolvents that have been unified or merged. Thus a new substitution set Σ_i is merged with the known instances of the remaining literals of the nucleus. Thus our final ur-resolution scheme on substitution sets is

$$\frac{\begin{array}{ll}\Sigma_i, 1\leq i\leq m & \\ \Sigma_1,\ldots,\Sigma_{i-1},\Sigma_{i+1},\ldots,\Sigma_m,(\tau_{m+1},\sigma_{m+1}) & \qquad \exists\Sigma\neq\emptyset.\ \Sigma=\Sigma_1\times\ldots\times\Sigma_m\end{array}}{\texttt{apply}(\Sigma,\tau_{m+1},\sigma_{m+1})}$$

This modified ur-resolution scheme requires substitution sets attached to vertices and links in the clause graph. For example, the substitution set Σ_i in the last rule is associated to the vertex of literal L_i. Later we will show how substitution sets attached to links can be exploited to support subsumption on simultaneous unifiers.

3.3.2 Definitions

3.3.2.1 Sets of Substitution Sets

We define four sets *REC*, *MRG*, *RES*, and *SNT* containing substitution sets that are associated to a clause graph. The first two sets *REC* and *MRG* contain substitution sets that are attached to vertices in a clause graph. The set *REC* contains sets of *received* substitutions. Received substitutions correspond to unifiers of new inferences and the according receiver literals. These substitutions have not been considered for simultaneous unification yet. Substitutions that have been received and simultaneously unified are added to substitution sets in the set *MRG*. These substitutions are called *merged* substitutions.

The other two sets *RES* and *SNT* contain substitution sets that are attached to links in a clause graph. The set *RES* contains sets of *resolved* substitutions for a single link. Resolved substitutions correspond to the common instances of simultaneous unifiers of ur-resolution. In practice, not all of the resolved but often exponentially many substitutions can be considered for subsequent resolution steps. Substitutions that have been considered for subsequent steps are called *sent* substitutions. These substitutions are collected in the substitution sets of the set *SNT*.

Definition 3.3.1 (Sets *REC*, *MRG*, *RES*, *SNT* of Substitution Sets)
Let $CG_S = (V, E)$ be a clause graph. Let $REC = \{\Sigma_{v_1}, \ldots, \Sigma_{v_m}\}, m \geq 0$ be an ordered set of substitution sets Σ_{v_i} and let $MRG = \{\Theta_{v_1}, \ldots, \Theta_{v_m}\}, m \geq 0$ be an ordered set of substitution sets Θ_{v_i}. Each element of *REC* belongs to exactly one vertex $v_i \in V$ of the clause graph CG_S. A substitution set $\Sigma_{v_i} \in REC$ corresponds to the vertex $v_i \in V$ of CG_S. The same correspondence applies to the set *MRG*.

Furthermore, let $RES = \{\Gamma_{v_k,v_l}, \ldots, \Gamma_{v_m,v_n}\}$ be an ordered set of substitution sets Γ_{v_i,v_j} and let $SNT = \{\Delta_{v_k,v_l}, \ldots, \Delta_{v_m,v_n}\}$ be an ordered set of substitution sets Δ_{v_i,v_j}. Each element of *RES* belongs to exactly one link $(v, (\tau, \sigma), w) \in E$ of the clause graph CG_S. A substitution set $\Gamma_{v,w} \in RES$ corresponds to the link $(v, (\tau, \sigma), w) \in E$ of CG_S. The same correspondence applies to the set *SNT*.

3.3.2.2 Transition System PURR

The transition system PURR describes the fundamental algorithm performed by our theorem prover Purr. The transition system defines ur-resolution on substitution sets in a clause graph. The transition rules of PURR use the formerly introduced sets to maintain substitution sets. Substitutions are exchanged between substitution sets of connected literals according to our modified ur-resolution scheme.

We begin with the definition of the auxiliary functions apply and strategy. The function apply combines ur-resolvents with the literal of a nucleus. This operation corresponds to the test unification and the send instantiation. We use the function strategy for the selection of 'best' ur-resolvents. Finally, we present the transition system PURR.

The Combination of UR-Resolvents and a Nucleus. The function apply performs the test unification and the send instantiation. The definition does not consider the technical details of variable renaming. For more details see Chapter 6.

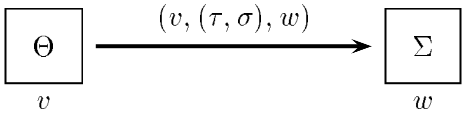


Figure 3.5: Test Unification and Send Instantiation

Definition 3.3.2 (Test Unification and Send Instantiation)
Let $(v, (\tau, \sigma), w)$ be a labeled link. Let Σ and Θ be two substitution sets. The substitutions of Θ belong to the literal of vertex v. The substitutions of Σ belong to the literal of vertex w. We define:

$$\Sigma := \texttt{apply}(\Theta, \tau, \sigma) \text{ if}$$
$$\forall \mu \in \Sigma, \exists \lambda \in \Theta : \rho \text{ is the most general unifier of } \lambda \text{ and } \tau \text{ and}$$
$$\mu \text{ is the renamed instance of } \sigma\rho$$

In Figure 3.5 the literal of vertex v is a sender literal connected by a link $(v, (\tau, \sigma), w)$ to the receiver literal of vertex w. The set Θ contains substitutions which represent instances of the sender literal v. These instances are ur-resolvents. The function `apply` combines the ur-resolvents with the receiver literal w of a nucleus. The set Σ contains substitutions as a result of successful combinations. The substitutions in Σ belong to the receiver literal w.

The combination of ur-resolvents with the receiver literal of a nucleus corresponds to the test unification and the send instantiation. Each substitution λ in Θ is tested for unifiability with the test substitution τ. The most general unifier ρ of a successful test unification is applied to the send substitution σ. The result is renamed to a substitution μ in Σ.

The Selection of UR-Resolvents. The selection of 'best' ur-resolvents is usually required in order to reduce the number of drawn inferences. We also refer to the heuristic of selection as the *strategy*. The function strategy selects substitutions of a substitution set according to a predicate is_wanted. Therefore, the predicate is_wanted actually determines the strategy.

Definition 3.3.3 (Strategy)
Let Σ and Θ be two substitution sets. Let is_wanted be a predicate. We define:

$$\mathsf{strategy}(\Theta) \quad = \quad \{\mu \in \Theta \mid \mathsf{is\_wanted}(\mu)\}$$

The substitutions in $\mathsf{strategy}(\Theta)$ are referred to as *given substitutions*. One example for the predicate is_wanted is to select a certain number of *lightest* substitutions, i.e. substitutions which have the smallest number of symbols. Note that the definition of is_wanted is left open because the strategy itself is not important at this point.

The Transition System `PURR`. The transition system `PURR` consists of the transition rules `RESOLVE`, `TRYTERMINATE`, and `TERMINATE`. The transition rules are applied to the sets *REC*, *MRG*, *RES*, and *SNT* of a clause graph. We define the application of ur-resolution according to one link in a clause graph with the transition rule `RESOLVE`. The transition rule `TRYTERMINATE` maintains the set *MRG* of merged substitutions if the termination test fails. The transition rule `TERMINATE` stops the transition process if the empty clause is detected.

Definition 3.3.4 (Transition System `PURR` without Subsumption)
Let $CG_S = (V, E)$ be a clause graph and let *REC*, *MRG*, *RES*, and *SNT* be sets of substitution sets associated to the clause graph CG_S according to Def. 3.3.1. Initially, the following conditions hold:

1. $RES = \emptyset, SNT = \emptyset$
2. $\forall v \in V : \mathsf{clause}(v) = \{v\}, \forall w \in V : (v, (\tau, \sigma), w) \in E :$
   $\mathtt{apply}(\emptyset, \{\}, \sigma) \in \Sigma_w \wedge \Sigma_w \in REC$
   The substitution sets in *REC* of all literals with incoming links from unit clauses are initialized with the renamed send substitutions σ.
3. $MRG = REC$

Note that the initial state corresponds to the application of ur-resolution with all possible combinations of unit clauses and nuclei. Subsequent ur-reolvents are produced by the transition rule RESOLVE. This rule also uses the set of vertices V and the set of links E of the clause graph CG_S. The symbol $\uplus$ denotes the disjoint set union.

RESOLVE:

$$\frac{\langle \mathit{REC} \uplus \{\Sigma_u\} \uplus \{\Sigma_w\} \quad , \mathit{MRG} \uplus \{\Theta_u\} \quad , \mathit{RES} \uplus \{\Gamma_{v,w}\} \quad , \mathit{SNT} \uplus \{\Delta_{v,w}\}\rangle}{\langle \mathit{REC} \cup \{\Sigma_u\} \cup \{\Sigma_w \cup \Sigma\},\ \mathit{MRG} \cup \{\Theta_u \cup \Sigma_u\},\ \mathit{RES} \cup \{\Gamma_{v,w} \cup \Gamma\},\ \mathit{SNT} \cup \{\Delta_{v,w} \cup \Delta\}\rangle}$$

$$\begin{aligned}
\text{if } & \exists (v, (\tau, \sigma), w) \in E,\ \exists u \in V : \mathsf{clause}(v) = \{u, v, v_1, \ldots, v_m\} \ \wedge \\
& \exists \Theta_{v_1}, \ldots, \Theta_{v_m} \in \mathit{MRG} : \Gamma = \Sigma_u \times \Theta_{v_1} \times \ldots \times \Theta_{v_m},\ \Theta_{v_i} \neq \emptyset \ \wedge \\
& \qquad \Delta = \mathsf{strategy}(\{\Gamma_{v,w} \cup \Gamma\} \backslash \Delta_{v,w}) \ \wedge \\
& \qquad \Sigma = \mathtt{apply}(\Delta, \tau, \sigma)
\end{aligned}$$

The literal u represents the receiver literal of the nucleus $\{u, v, v_1, \ldots, v_m\}$. The literal v is the sender literal of the nucleus connected to the receiver literal w of another clause.

The substitutions in Σ_u are newly received substitutions. The substitution sets Θ_{v_i} of the remaining literals $v_1, \ldots, v_m$ contain substitutions which have already been merged in previous ur-resolution steps. These substitutions have to be merged with Σ_u. Then, the substitutions in Σ_u are copied to the set Θ_v of merged substitutions. The resulting common instances Γ of the simultaneous unification are inserted as resolved substitutions into the according set $\Gamma_{v,w}$.

The strategy selects the 'best' substitutions Δ of all resolved substitutions except for those substitutions $\Delta_{v,w}$ that have already been sent. Finally, the instances Σ of the receiver literal w are inserted into the according set Σ_w of received substitutions.

TRYTERMINATE:

$$\frac{\langle \mathit{REC} \uplus \{\Sigma_v\},\ \mathit{MRG} \uplus \{\Theta_v\} \quad , \mathit{RES},\ \mathit{SNT}\rangle}{\langle \mathit{REC} \cup \{\Sigma_v\},\ \mathit{MRG} \cup \{\Theta_v \cup \Sigma_v\},\ \mathit{RES},\ \mathit{SNT}\rangle}$$

$$\begin{aligned}
\text{if } & \exists v \in V : \mathsf{clause}(v) = \{v, v_1, \ldots, v_m\} \ \wedge \\
& \exists \Theta_{v_1}, \ldots, \Theta_{v_m} \in \mathit{MRG} : \Sigma_v \times \Theta_{v_1} \times \ldots \times \Theta_{v_m} = \emptyset,\ \Theta_{v_i} \neq \emptyset
\end{aligned}$$

The transition rule TRYTERMINATE considers failed termination tests. The literal v represents the receiver literal of a clause $\{v, v_1, \ldots, v_m\}$. The substitutions in Σ_v are newly received substitutions. The substitution sets Θ_{v_i} of the remaining literals $v_1, \ldots, v_m$ contain substitutions which have already been merged in previous termination test. These substitutions have to be merged with Σ_v. Then, the substitutions in Σ_v are copied to the set Θ_v of merged substitutions. The termination fails since there is no simultaneous unifier that instantiates the clause $\{v, v_1, \ldots, v_m\}$ to the empty clause.

TERMINATE:

$$\frac{\langle \mathit{REC} \uplus \{\Sigma_v\},\ \mathit{MRG},\ \mathit{RES},\ \mathit{SNT}\rangle}{\mathit{STOP}}$$

$$\begin{aligned}
\text{if } & \exists v \in V : \mathsf{clause}(v) = \{v, v_1, \ldots, v_m\} \ \wedge \\
& \exists \Theta_{v_1}, \ldots, \Theta_{v_m} \in \mathit{MRG} : \Sigma_v \times \Theta_{v_1} \times \ldots \times \Theta_{v_m} \neq \emptyset,\ \Theta_{v_i} \neq \emptyset
\end{aligned}$$

The transition rule TERMINATE stops the search process if all literals of a single clause are instantiated by simultaneously unifiable substitutions.

3.3.2.3 Example

Consider the following example of a clause set.

$$\begin{array}{l}
\{Q(x, f(y))\} \\
\{P(f(f(x)), x, a)\} \\
\{\neg Q(f(x), x), \neg P(f(x), f(y), a), P(x, f(y), z)\} \\
\{\neg P(f(b), f(b), c)\} \text{ is a query clause}
\end{array}$$

The example has been presented on page 18. The according clause graph is depicted in Figure 3.4. The refutation of the query clause $\{\neg P(f(b), f(b), c)\}$ consists of two ur-resolution steps. The ur-resolvent (5) refutes the denied theorem.

$$\begin{array}{ll}
(1)\ \{Q(x, f(y))\} & \\
(2)\ \{P(f(f(z)), z, a)\} & \\
(3)\ \{\neg Q(f(u), u), \neg P(f(u), f(v), a), P(u, f(v), w)\} & \mu = \{u \mapsto f(f(v))\} \\
\hline
(4)\ \{P(f(f(v)), f(v), w)\} &
\end{array}$$

$$\begin{array}{ll}
(1)\ \{Q(x, f(y))\} & \\
(4)\ \{P(f(f(v)), f(v), w)\} & \\
(3)\ \{\neg Q(f(x_1), x_1), \neg P(f(x_1), f(x_2), a), P(x_1, f(x_2), x_3)\} & \mu = \{x_1 \mapsto f(x_2)\} \\
\hline
(5)\ \{P(f(x_2), f(x_2), x_3)\} &
\end{array}$$

The same refutation can be computed on the clause graph with the transition system PURR. Figure 3.6 depicts the initial state of the sets *REC*, *MRG*, *RES*, and *SNT*. The substitution sets of *REC* and *MRG* are attached to vertices v_3, v_4, and v_6. The internal link and the link to the theorem are labeled with the according substitution sets of *RES* and *SNT*. Note that parts of the sets always remain empty and therefore have been omitted.

The sets in *REC* and *MRG* of the vertex v_3 contain the substitution $\mu_1 = \{x \mapsto f(y_1)\}$. This substitution corresponds to the combination of the literals v_1 and v_3. The variable in the codomain of μ_1 denoted with y_1 belongs to the literal v_1. Note that μ_1 corresponds to the send substitution σ_1 modulo renaming.

The substitution $\mu_2 = \{x \mapsto f(f(y))\}$ corresponds to the combination of v_2 and v_4. Obviously, the variable y belongs to the same variable in the nucleus. Note that the assignment $\{y \mapsto y_r\}$ of the send substitution σ_2 has been omitted since y_r implicitly corresponds to the variable y of the nucleus.

Figure 3.7 shows the situation after performing the transition rule RESOLVE on the internal link $(v_5, (\tau_3, \sigma_3), v_4)$ and the receiver literal v_3. The substitution μ_1 is unified with all substitutions that have been received and merged at the remaining incoming links, i.e. it is unified with μ_2 of literal v_4. The common instance $\mu_3 = \{x \mapsto f(f(y))\}$ is stored in the set of resolved substitutions of the current link. Then the only substitution μ_3 in *RES* is selected as given and copied to the set of sent substitutions in *SNT*. The function apply computes the according instantiation $\mu_4 = \{x \mapsto f(y_5), y \mapsto y_5\}$ which is passed to *REC* of vertex v_4. Obviously, the variable y_5 belongs to the literal v_5. This transition corresponds to the first ur-resolution step.

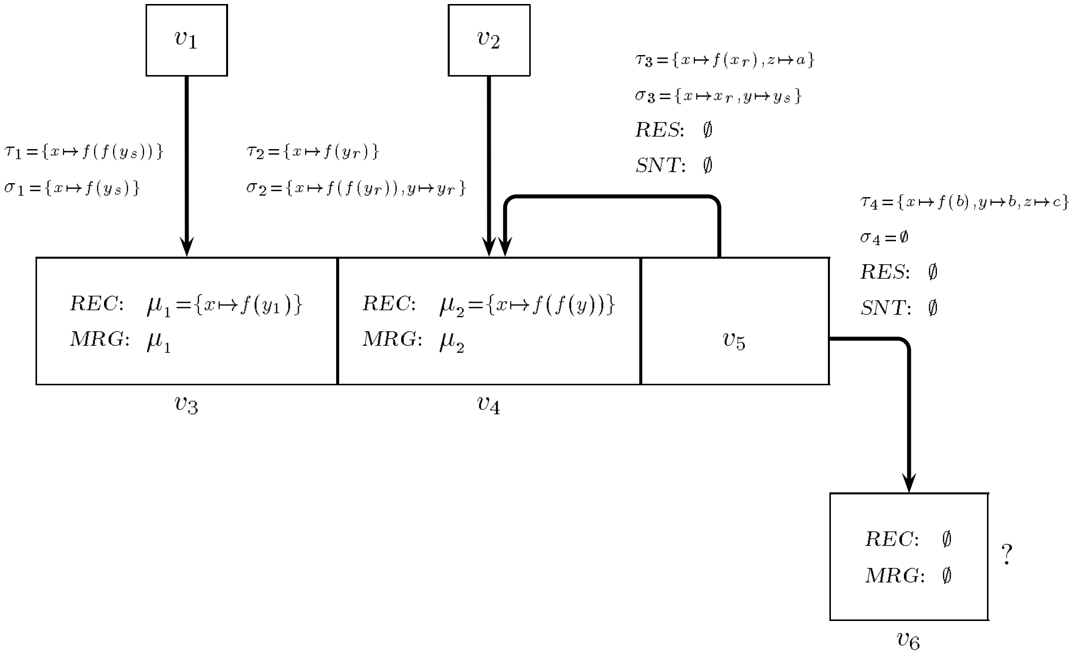

v_1
v_2
$\tau_3 = \{x \mapsto f(x_r), z \mapsto a\}$
$\sigma_3 = \{x \mapsto x_r, y \mapsto y_s\}$
RES: $\emptyset$
SNT: $\emptyset$
$\tau_1 = \{x \mapsto f(f(y_s))\}$
$\sigma_1 = \{x \mapsto f(y_s)\}$
$\tau_2 = \{x \mapsto f(y_r)\}$
$\sigma_2 = \{x \mapsto f(f(y_r)), y \mapsto y_r\}$
$\tau_4 = \{x \mapsto f(b), y \mapsto b, z \mapsto c\}$
$\sigma_4 = \emptyset$
RES: $\emptyset$
SNT: $\emptyset$
REC: $\mu_1 = \{x \mapsto f(y_1)\}$
MRG: μ_1
REC: $\mu_2 = \{x \mapsto f(f(y))\}$
MRG: μ_2
v_5
v_3
v_4
REC: $\emptyset$
MRG: $\emptyset$
?
v_6

Figure 3.6: Initial State of the Search Sets

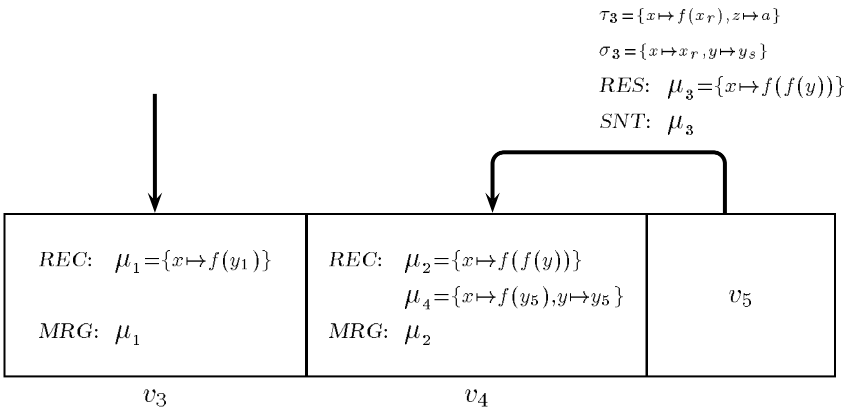

$\tau_3 = \{x \mapsto f(x_r), z \mapsto a\}$
$\sigma_3 = \{x \mapsto x_r, y \mapsto y_s\}$
RES: $\mu_3 = \{x \mapsto f(f(y))\}$
SNT: μ_3
REC: $\mu_1 = \{x \mapsto f(y_1)\}$
MRG: μ_1
REC: $\mu_2 = \{x \mapsto f(f(y))\}$
$\mu_4 = \{x \mapsto f(y_5), y \mapsto y_5\}$
MRG: μ_2
v_5
v_3
v_4

Figure 3.7: `RESOLVE` applied on link $(v_5, (\tau_3, \sigma_3), v_4)$ with receiver literal v_3

The second ur-resolution step is performed by `RESOLVE` on the link $(v_5, (\tau_4, \sigma_4), v_6)$ with the receiver literal v_4. See Figure 3.8 for the current state. The substitutions μ_2 and μ_4 of v_4 are unified with the merged substitution μ_1 of v_3. The resulting common instances are $\mu_5 = \{x \mapsto f(f(y))\}$ and $\mu_6 = \{x \mapsto f(y_5), y \mapsto y_5\}$. We select μ_6 as given. The according instantiation $\mu_7 = \emptyset$ is sent to *REC* of v_6. The substitution μ_7 is empty due to the empty send substitution σ_4.

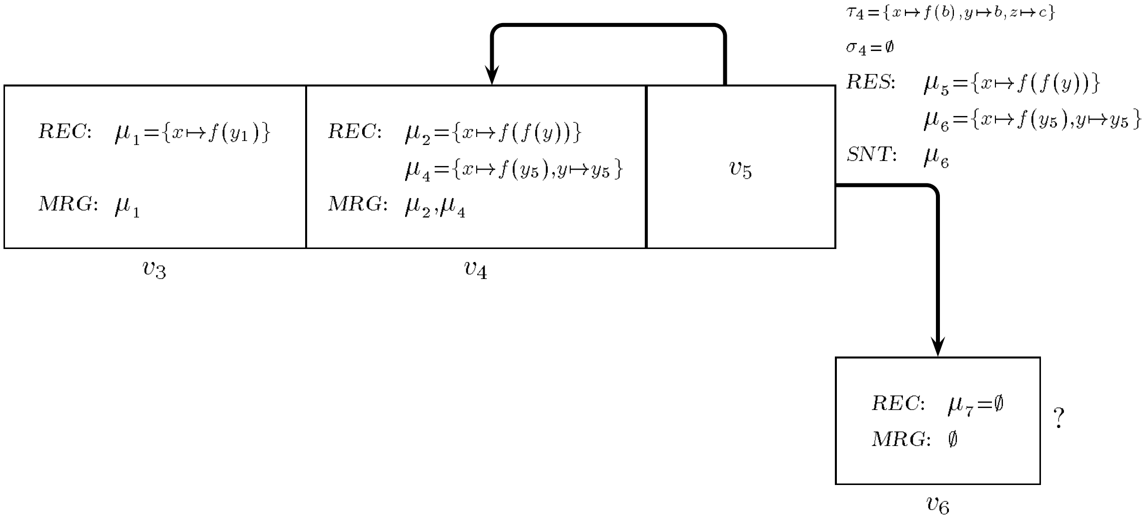

$\tau_4=\{x\mapsto f(b),y\mapsto b,z\mapsto c\}$
$\sigma_4=\emptyset$
RES: $\mu_5=\{x\mapsto f(f(y))\}$
$\mu_6=\{x\mapsto f(y_5),y\mapsto y_5\}$
SNT: μ_6
REC: $\mu_1=\{x\mapsto f(y_1)\}$
MRG: μ_1
REC: $\mu_2=\{x\mapsto f(f(y))\}$
$\mu_4=\{x\mapsto f(y_5),y\mapsto y_5\}$
MRG: μ_2,μ_4
v_5
v_3
v_4
REC: $\mu_7=\emptyset$
MRG: $\emptyset$
?
v_6

Figure 3.8: `RESOLVE` applied on link $(v_5, (\tau_4, \sigma_4), v_6)$ with receiver literal v_4

Finally, the transition rule `TERMINATE` is applied on the receiver literal v_6. Obviously, `TERMINATE` succeeds because v_6 belongs to a unit clause, i.e. the unification test is not required. In general, a unit query clause is solved if at least one substitution has been received.

3.3.3 A Transition System with Subsumption

We define a transition system `SUBSUMPTION` which performs subsumption on two substitutions sets. Substitutions in one set which are subsumed by substitutions in the other set are removed.

Definition 3.3.5 (Transition System `SUBSUMPTION`)
The rule `SUBSUMPTION` is defined on two substitution sets. The rule performs subsumption on both sets by removing all instances of one set in the other set and vice versa.

$$\texttt{SUBSUMPTION:}\frac{\langle \Sigma \uplus \Sigma_{ins},\ \Theta \uplus \Theta_{ins} \rangle}{\langle \Sigma, \Theta \rangle} \quad \text{if} \quad \begin{array}{l} \forall \mu \in \Sigma_{ins},\ \exists \lambda \in \Theta : \lambda \text{ is a generalization of } \mu \wedge \\ \forall \mu \in \Theta_{ins},\ \exists \lambda \in \Sigma : \lambda \text{ is a generalization of } \mu \end{array}$$

The transition system `PURR` with subsumption points out where subsumption may take place in the reasoning process. We distinguish *input* and *output* subsumption: Subsumption might be performed with received substitutions and merged substitutions of one literal. This subsumption test is referred to as input subsumption. Resolved substitutions might be tested for subsumption with earlier resolved substitutions of one literal. This subsumption test is called output subsumption. In general, subsumption is considered as one of the most important reduction methods. Thus subsumption is an integral part of our approach. We present the final transition system `PURR` which includes input and output subsumption.

Definition 3.3.6 (Transition System PURR with Subsumption)
Let $CG_S = (V, E)$ be a clause graph and let *REC*, *MRG*, *RES*, and *SNT* be sets of substitution sets associated to the clause graph CG_S according to Def. 3.3.1. The initial state is defined as in Def. 3.3.4. The definition of the three transition rules is based on Def. 3.3.4 of the transition system PURR without subsumption. Here, we merely add the subsumption test to the rules.

RESOLVE:

$$\frac{\langle \mathit{REC} \uplus \{\Sigma_u\} \uplus \{\Sigma_w\},\ \mathit{MRG} \uplus \{\Theta_u\},\ \mathit{RES} \uplus \{\Gamma_{v,w}\},\ \mathit{SNT} \uplus \{\Delta_{v,w}\}\rangle}{\langle \mathit{REC} \cup \{\Sigma_u\} \cup \{\Sigma_w \cup \Sigma\},\ \mathit{MRG} \cup \{\Theta'_u \cup \Sigma'_u\},\ \mathit{RES} \cup \{\Gamma'_{v,w} \cup \Gamma'\},\ \mathit{SNT} \cup \{\Delta_{v,w} \cup \Delta\}\rangle}$$

if $\exists (v, (\tau, \sigma), w) \in E,\ \exists u \in V : \mathsf{clause}(v) = \{u, v, v_1, \ldots, v_m\}\ \wedge$

$$\begin{aligned}
\exists \Theta_{v_1}, \ldots, \Theta_{v_m} \in \mathit{MRG} : \langle \Sigma_u, \Theta_u\rangle \downarrow_{\texttt{SUBSUMPTION}} &= \langle \Sigma'_u, \Theta'_u\rangle\ \wedge \\
\Gamma &= \Sigma'_u \times \Theta_{v_1} \times \ldots \times \Theta_{v_m},\ \Theta_{v_i} \neq \emptyset\ \wedge \\
\langle \Gamma, \Gamma_{v,w}\rangle \downarrow_{\texttt{SUBSUMPTION}} &= \langle \Gamma', \Gamma'_{v,w}\rangle\ \wedge \\
\Delta &= \mathsf{strategy}(\{\Gamma'_{v,w} \cup \Gamma'\} \backslash \Delta_{v,w})\ \wedge \\
\Sigma &= \texttt{apply}(\Delta, \tau, \sigma)
\end{aligned}$$

The subsumption test with the set of received substitutions Σ_u and the set of merged substitutions Θ_u corresponds to the input subsumption. The subsumption test with the set of new ur-resolvents Γ and the set of previously resolved substitutions $\Gamma_{v,w}$ corresponds to the output subsumption.

TRYTERMINATE:

$$\frac{\langle \mathit{REC} \uplus \{\Sigma_v\},\ \mathit{MRG} \uplus \{\Theta_v\},\ \mathit{RES},\ \mathit{SNT}\rangle}{\langle \mathit{REC} \cup \{\Sigma_v\},\ \mathit{MRG} \cup \{\Theta'_v \cup \Sigma'_v\},\ \mathit{RES},\ \mathit{SNT}\rangle}$$

if $\exists v \in V : \mathsf{clause}(v) = \{v, v_1, \ldots, v_m\} \wedge \langle \Sigma_v, \Theta_v\rangle \downarrow_{\texttt{SUBSUMPTION}} = \langle \Sigma'_v, \Theta'_v\rangle\ \wedge$
$\exists \Theta_{v_1}, \ldots, \Theta_{v_m} \in \mathit{MRG} : \Sigma'_v \times \Theta_{v_1} \times \ldots \times \Theta_{v_m} = \emptyset,\ \Theta_{v_i} \neq \emptyset$

The input subsumption with the set of received substitutions Σ_v and the set of merged substitutions Θ_v can also reduce the amount of substitutions which have to be tested for termination.

TERMINATE:

$$\frac{\langle \mathit{REC} \uplus \{\Sigma_v\},\ \mathit{MRG} \uplus \{\Theta_v\},\ \mathit{RES},\ \mathit{SNT}\rangle}{\mathit{STOP}}$$

if $\exists v \in V : \mathsf{clause}(v) = \{v, v_1, \ldots, v_m\} \wedge \langle \Sigma_v, \Theta_v\rangle \downarrow_{\texttt{SUBSUMPTION}} = \langle \Sigma'_v, \Theta'_v\rangle\ \wedge$
$\exists \Theta_{v_1}, \ldots, \Theta_{v_m} \in \mathit{MRG} : \Sigma'_v \times \Theta_{v_1} \times \ldots \times \Theta_{v_m} \neq \emptyset,\ \Theta_{v_i} \neq \emptyset$

The ur-resolution refutation process was represented as the computation and distribution of substitution sets in a clause graph. Notions like clauses or literals are not needed any longer. In the next chapter we will present an efficient technique for the representation and manipulation of substitution sets called substitution tree indexing.

4

Indexing

Indexing supports the maintenance of large databases by providing fast access to stored data. In automated reasoning we employ databases that contain first-order terms. Typical queries to such term indexes are: Given a database D containing terms (literals) and a query term t, find all terms in D that are unifiable with, instances of, variants of, or more general than t.

So far, many successful theorem provers use term indexing to support the reasoning process. We use term indexing not only as a tool but as the fundamental data structure during the proof search. Term indexing replaces the standard implementation of literals and clauses. All operations on resolvents, like resolution and subsumption, are indexing operations. The system even communicates by sending sets of resolvents stored in indexes.

In the first section we present a classification scheme for indexing techniques. An indexing method called *substitution tree indexing* [Gra94b] is presented in detail in Section 2. We use substitution trees for the representation of sets of substitutions. In the last section we investigate the operations on substitution trees providing an efficient implementation of the former presented unit resulting resolution on sets of substitutions.

This chapter is mainly based on the PhD thesis of Peter Graf [Gra96]. His thesis also provides an exhaustive discussion of other term indexing methods including the following classification scheme.

4.1 Classification of Indexing Techniques

The main purpose of indexing techniques in theorem provers is to achieve efficient access to first-order terms with specific properties. To this end a set of terms $\mathcal{I}$ is inserted into an indexing data structure. A retrieval in $\mathcal{I}$ is started for a set $\mathcal{Q}$ of *query terms*. The aim of the retrieval is to find tuples (s, t) with $s \in \mathcal{I}$ and $t \in \mathcal{Q}$ in such a way that a special relation R holds for s and t. Most automated reasoning systems can profit from a retrieval based on the following relations: s and t are *unifiable*, t is an *instance* of s, and s is a *generalization* of t. The relation for unifiability can be used for the retrieval of complementary unifiable literals in a resolution based system, for example. Moreover, possibly forward or backward subsumed clauses are found by accessing more general or instance literals.

If we are interested in retrieving indexed *substitutions* instead of indexed terms, a relation $R(\sigma, \tau)$ is needed. We consider the relations of type $R(\sigma, \tau)$ as generalizations of the relations of type $R(s, t)$ since indexing substitutions using the relation $R(\{x \mapsto s\}, \{x \mapsto t\})$ is equivalent to using $R(s, t)$. An application of indexed substitutions is unit resulting resolution, where simultaneous unification of substitutions has to be performed.

Retrieval of Type 1:1, n:1, and n:m. A retrieval is of type 1:1 if both sets $\mathcal{I}$ and $\mathcal{Q}$ have cardinality 1. Since both sets $\mathcal{Q}$ and $\mathcal{I}$ solely consist of one single term or substitution, the retrieval corresponds to simply testing if $R(s, t)$ holds.

Retrieval of type n:1 is determined by a single query term t, which is used to find entries $s \in \mathcal{I}$. The set $\mathcal{I}$ of n indexed terms is represented by an indexing data structure. The result of a retrieval is a subset of $\mathcal{I}$. Note that a very inefficient retrieval of type n:1 could be performed by testing each entry of the index in a 1:1 type retrieval because such an approach would have to consider all indexed terms explicitly.

Retrieval of type n:m includes all cases in which more than a single query term is involved. Exploiting n:m indexing, the query set typically is also represented by an index. Hence, we have to deal with two indexes; one of them represents the indexed and the other one represents the query set. The result of such a retrieval is a subset of the direct product of the term sets involved.

As an example, we consider a n:m retrieval called *merge*: Suppose we are looking for simultaneous unifiers for ur-resolution. We create an index for each literal of the nucleus. Each index contains the unifiers of the literal with electrons. In this example it is of advantage if the indexing technique employed is able to index substitutions in a convenient manner. In a merge operation we look for simultaneous unifiers of the electrons and the literals of the nucleus. In case the nucleus contains more than two literals that have to be merged, we can extend the merge operation to an arbitrary number n of indexes. In this case we find n-tuples instead of pairs and call the according retrieval operation the *multi-merge*.

Maintenance of Type n:1 and n:m. In addition to the retrieval operations we also have to provide functions that *insert* entries into and *delete* entries from the indexing structure. Insertion and deletion can also be classified according to the cardinalities of the involved sets.

Maintenance of type n:1 includes all operations that modify an index by a single term. Beside the classical insertion and deletion operations of a single term, the deletion of all instances of a term, for example, also corresponds to an n:1 maintenance operation.

Maintenance of type n:m corresponds to index manipulation operations that fit into the concept of n:m indexing. For example, the union of two indexes results in a new index that contains all terms of the two sets involved. An additional n:m maintenance task is to delete all instances of $\mathcal{Q}$ that occur in $\mathcal{I}$ from $\mathcal{I}$. Such an operation is used for subsumption in the case of unit clauses, for example.

4.2 Substitution Tree Indexing

Memory requirement and retrieval times being the main criteria for judging an indexing technique, substitution tree indexing is superior to the known tree-based strategies in these points. Substitution trees can represent any set of idempotent substitutions. In the simplest case all these substitutions have identical domains and consist of a single assignment, which implies that the substitution tree can be used as a term index as well. Figure 4.1 shows an index for the three substitutions $\{u \mapsto f(a,b)\}$, $\{u \mapsto f(y,b)\}$, and $\{u \mapsto f(b,z)\}$ which obviously represents a term index for the terms $f(a,b)$, $f(y,b)$, and $f(b,z)$. As the name indicates, the labels of substitution tree nodes are substitutions. Each branch in the tree therefore represents a binding chain for variables. Consequently, the substitutions of a branch from the root node down to a particular node can be composed and yield an instance of the root node's substitution.

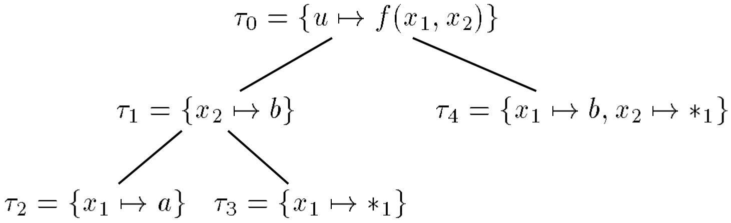

$\tau_0 = \{u \mapsto f(x_1, x_2)\}$
$\tau_1 = \{x_2 \mapsto b\}$
$\tau_4 = \{x_1 \mapsto b, x_2 \mapsto *_1\}$
$\tau_2 = \{x_1 \mapsto a\}$
$\tau_3 = \{x_1 \mapsto *_1\}$

Figure 4.1: Substitution tree

Before substitutions are inserted into the index, their codomain is renamed. This *normalization* changes all variables in the codomain of a substitution. Renamed variables are called *indicator variables* and are denoted by $*_i$. The substitutions inserted to the index in Figure 4.1 therefore were $\{u \mapsto f(a,b)\}$, $\{u \mapsto f(*_1,b)\}$, and $\{u \mapsto f(b,*_1)\}$. This renaming has two main reasons: There is more sharing in the index if the substitutions are *normalized* and, for some retrieval tasks, it is necessary to distinguish between variables occurring in the query and in the indexed terms. The latter may not be instantiated when looking for instances of query terms, for example.

Consider the substitution $\tau = \{u \mapsto f(a,b)\}$, which is represented by the chain of substitutions $\tau_0 = \{u \mapsto f(x_1,x_2)\}$, $\tau_1 = \{x_2 \mapsto b\}$, and $\tau_2 = \{x_1 \mapsto a\}$. The original substitution τ can be reconstructed by simply applying the substitution $\tau_0\tau_1\tau_2$ to u. The result of this application is

$$\begin{aligned}\tau &= \{u \mapsto u\tau_0\tau_1\tau_2\}\\ &= \{u \mapsto f(x_1,x_2)\tau_1\tau_2\}\\ &= \{u \mapsto f(x_1,b)\tau_2\}\\ &= \{u \mapsto f(a,b)\}\end{aligned}$$

The retrieval in a substitution tree is based on a backtracking algorithm. This algorithm exploits a backtrackable variable binding mechanism, similar to the one used in Prolog.

To illustrate a retrieval operation, the search for substitutions compatible with $\{u \mapsto f(a,x)\}$ in our example index is presented: We search for substitutions τ such that $u\tau$ is unifiable with $f(a,x)$. We begin by binding the variable u to the term $f(a,x)$ and start the retrieval: The substitution tree is traversed by testing at each node marked with the substitution $\tau = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ whether under the current bindings all x_i are unifiable with their appropriate

t_i. At the root node we unify the terms $f(a, x)$ and $f(x_1, x_2)$, which yields the two bindings $x_1 \mapsto a$ and $x \mapsto x_2$. Then we consider the first son of the root node marked with τ_1 and unify x_2 with b, because x_2 has not been bound yet. The resulting binding is $x_2 \mapsto b$ and the leaf node τ_2 is the next node to be investigated. As x_1 is bound to a, the unification problem is trivial and therefore the substitution represented by this leaf node is compatible with $\{u \mapsto f(a, x)\}$. After backtracking node τ_3 is found to represent another solution, because the variable $*_1$ is unifiable with a. Backtracking deletes the bindings of $*_1$ and x_2 and then proceeds with node τ_4. Obviously, retrieval can be stopped at this point, because a, which is the binding of x_1, is not unifiable with b.

4.2.1 Standard Substitution Trees

4.2.1.1 Definitions

We use a backtracking algorithm to find substitutions in the tree with specific properties. All retrieval algorithms are based on backtrackable variable bindings and algorithms for unification and matching, which take variable bindings into account. Insertion of a substitution into the index is a complex operation. Compared to insertion, the deletion of entries is much more straightforward and even complex deletion operations, like the deletion of all compatible substitutions in a substitution tree, can easily be accomplished.

Definition 4.2.1 (Substitution Tree)
A *substitution tree* is an ordered tree. We describe a substitution tree by a tuple (τ, Σ) where τ is a substitution and Σ is an ordered set of subtrees. The following conditions hold:

1. A node in the tree is a *leaf* node $(\tau, \emptyset)$ or an *inner node* (τ, Σ) with $|\Sigma| \geq 2$.
2. For every path $(\tau_1, \Sigma_1), \ldots, (\tau_n, \Sigma_n)$ from the root to a leaf of a tree we have
$$\mathsf{IM}(\tau_1 \bullet \ldots \bullet \tau_n) \subset \mathbf{V}^*$$
3. For every path $(\tau_1, \Sigma_1), \ldots, (\tau_i, \Sigma_i)$ from the root to any node of a tree we have
$$\mathsf{DOM}(\tau_i) \cap \bigcup_{1 \leq j < i} \mathsf{DOM}(\tau_j) = \emptyset$$

The first condition in the definition assures that each inner node of a substitution tree has at least two subtrees. The third condition assures that in each path of the tree a variable occurs no more than once in the domain of a substitution.

If $(\tau_1, \Sigma_1), \ldots, (\tau_n, \Sigma_n)$ is a path from the root of the tree to node (τ_n, Σ_n) and x occurs in the codomains of the τ_i but not in the domains of the τ_i, then the variable x is called *open* at node (τ_n, Σ_n). Variables that are not open at a node N are called *closed* at N. The second condition in Def. 4.2.1 implies that all non-indicator variables are closed at leaf nodes of substitution trees. The *empty* tree is denoted by ε.

4.2.1.2 Foundations of the Retrieval in Substitution Trees

Retrieval. The retrieval algorithm checks each node of the tree for certain conditions. If the conditions are fulfilled, the algorithm proceeds with the subnodes of the node. If the conditions are fulfilled at a leaf node, the entry of the index represented by this leaf is retrieved. Four different retrieval tasks are supported: Find more general substitutions, compatible substitutions, instances, and variant substitutions. The functions G, U, I, and V support the tests at the nodes of the tree.

The test functions G, U, I, and V take two substitutions τ and ρ. The substitution τ is the one that is stored at a substitution tree's node. The substitution ρ describes all variable bindings that have been established while descending from the root to the current node of the tree. When we test the root of the tree, the substitution ρ only contains the query substitution.

For each assignment $x_i \mapsto t_i$ of the current node's substitution τ the functions test whether the term $x_i\rho$ is more general, unifiable with, an instance of, or a variant of $t_i\rho$. Each of the test functions can be used as a parameter for the retrieval function `search`.

Definition 4.2.2 (Test Functions for Retrieval)
Let τ and ρ be two substitutions. Then

$$\begin{aligned}
\mathsf{G}(\tau,\rho) &:= \{\sigma \mid \forall x_i \in \mathsf{DOM}(\tau).\ x_i\tau\rho\sigma = x_i\rho\} \\
\mathsf{U}(\tau,\rho) &:= \{\sigma \mid \forall x_i \in \mathsf{DOM}(\tau).\ x_i\tau\rho\sigma = x_i\rho\sigma \ \wedge\ \sigma \text{ is most general}\} \\
\mathsf{I}(\tau,\rho) &:= \{\sigma \mid \sigma \in \mathsf{U}(\tau,\rho) \ \wedge\ \mathsf{DOM}(\sigma) \cap \mathbf{V}^* = \emptyset\} \\
\mathsf{V}(\tau,\rho) &:= \{\sigma \mid \sigma \in \mathsf{G}(\tau,\rho) \ \wedge\ \mathsf{DOM}(\sigma) \cap \mathbf{V}^* = \emptyset\}
\end{aligned}$$

The test function G checks for every assignment of the substitution τ stored in the tree if under the current variable bindings denoted by ρ a simultaneous matcher σ from all terms of the codomain to the corresponding bindings of the domain variables exists. The test function U works similar, but tests if a simultaneous unifier exists. The remaining test functions I and V are defined using U and G, respectively. The function I only allows bindings of non-indicator variables, thus avoiding variables of indexed terms to be bound. Since variant indexed terms have to be identical to the normalized versions of query terms, the function V allows bindings of the tree's auxiliary variables only.

The retrieval function `search`$(N, \rho, \mathcal{X})$ takes the substitution τ, which is stored at node $N = (\tau, \Sigma)$ in the tree and tests τ against the current variable bindings ρ using one of the test functions G, U, I, or V. Although one might only be interested in leaf nodes found in the substitution tree, the function `search` produces a set of nodes, which have successfully passed the test $\mathcal{X}$, no matter if they are leaf nodes or not. Note that before searching for variants the query substitution has to be normalized.

Insertion. The function `insert`(N, ρ) inserts a substitution ρ into a substitution tree N resulting in a modified substitution tree. As the index is used as a means for accessing data, it is possible in practice to store additional information at the leaf nodes of the tree. The insertion of entries into substitution trees is more difficult than retrieval or deletion. The exact definitions have been presented by Peter Graf [Gra94b].

Deletion. During the deletion process subtrees can be removed. Recall that in a substitution tree each inner node at least has to have two subtrees. Since the resulting tree does not automatically have this property, we have to change it in an appropriate way: The tree is "repaired". We assume that $N = (\tau, \Sigma)$ is a tree according to the definition of a substitution tree except that the set of subtrees Σ may contain less than two subnodes. The function `repair`(τ, Σ) takes the node's substitution τ and the set of subtrees Σ. The tree resulting from the application of `repair` is a substitution tree according to the original definition.

Using the deletion function `delete`(N, ρ) all variants of the substitution ρ in the substitution tree N are deleted resulting in a modified substitution tree. Note that ρ does not need to be normalized.

Combining Retrieval and Deletion: A great advantage of substitution trees is that the deletion function can easily be modified so it will remove instances, generalizations, or unifiable entries from the index. We simply have to use the test functions $\mathsf{I}(\tau, \rho)$, $\mathsf{G}(\tau, \rho)$, or $\mathsf{U}(\tau, \rho)$ instead of the test for variants $\mathsf{V}(\tau, \rho)$ used in the function `delete`.

The function `delete'`$(N, \rho, \mathcal{X})$ computes a modified substitution tree by removing substitutions ρ from tree N according to the test function $\mathcal{X}$. Again, ρ does not need to be normalized.

4.2.2 Substitution Trees in PURR

In PURR we use a combination of two standard substitution tree variants. Additionally, we added some extra information to support a fast selection of so-called lightest substitutions. The next paragraph introduces the first variant, the so-called linear substitution tree. Furthermore, we use a variant called weighted substitution tree. These two variants can easily be combined and yield significant improvements in the insertion and retrieval performance of standard substitution trees. This linear weighted substitution tree is extended again which we discuss in the third paragraph.

Linear Substitution Trees. The only difference between standard and *linear substitution trees* (LST) lies in the maximal number of occurrences of a single auxiliary variable x_i on a path from the root to a leaf of a tree. Variables that occur in substitution trees but not in indexed substitutions are auxiliary variables. In standard substitution trees the number of occurrences is not restricted. In linear trees we have the simple restriction that each auxiliary variable occurs at most once in a codomain and at most once in a domain of another substitution along a path from the root to a leaf node of the tree. As in all substitution trees, the occurrence of the auxiliary variable in the domain of a substitution must be located deeper in the tree than the occurrence in the codomain. Using linear substitution trees we try to simplify insertion and to accelerate retrieval.

Weighted Substitution Trees. The number of symbols contained in the codomain of a substitution is called the substitution's *size*. In *weighted substitution trees* (WST) additional information about the size of the indexed substitutions is added to every node of the tree. We mainly hope to increase the speed of subsumption tests by using weighted substitution trees. During subsumption we have to decide whether substitutions stored in a tree can be instances of substitutions stored in another tree. We can take advantage of the following observation: A substitution σ can only be an instance of another substitution τ if the size of σ is at least as large as the size of τ.

The substitution tree is modified by storing an interval $[\mathsf{min}, \mathsf{max}]$ at each node where min contains the minimal and max contains the maximal size of the substitutions stored in the subtrees. At leaf nodes the values of min and max are identical to the size of the represented

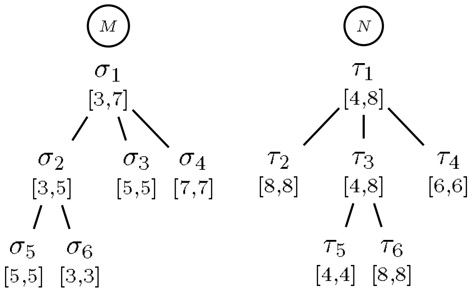


Figure 4.2: Subsumption in Weighted Substitution Trees

substitution. Since the size of a substitution is computed before insertion, the intervals stored at the nodes of the tree can easily be updated. In Figure 4.2 we see two weighted substitution trees M and N.

4.2.2.1 Definitions

Selective Substitution Trees. A heuristic that is used in many resolution-based theorem provers selects the smallest clauses in the set of kept clauses for the application of inference rules. In Purr clauses are represented by substitutions and therefore we have to select the substitutions with the minimum weight for the application of inference rules. Moreover, substitutions which have been selected must be excluded from subsequent retrieval.

We introduce a variant of substitution trees called *selective substitution trees*. This variant is similar to a weighted substitution tree and can easily be combined to a selective weighted substitution tree. We add information to every node of the tree about the size and about whether substitutions have been selected. During selection the size information adapts to the minimal size of unselected entries. The state information provides an efficient exclusion of formerly selected entries.

Definition 4.2.3 (Selective Substitution Tree (SEST))
A *selective substitution tree* is a four-tupel (τ, Σ, w, s) where the substitution τ and the ordered set of subtrees Σ correspond to Definition 4.2.1 of a substitution tree. The weight w is the smallest weight of the unselected entries in the tree. The *state* flag s contains the information whether there are unselected entries in the tree. In addition to Definition 4.2.1, the following conditions hold:

1. A leaf node $(\tau, \emptyset, w, s)$ refers to entries with the corresponding weight w. We have $s = \textsf{true}$ if all entries have been selected.

2. The weight w of an inner node (τ, Σ, w, s) is the minimum of the weights of the subtrees Σ with unselected substitutions. We have $s = \textsf{true}$ if all entries of the subtrees Σ have been selected. In this case the weight w is arbitrary.

We will use the definition of selective substitution trees later in a selection scheme for the first n lightest entries.

Note that the weight of a substitution does not necessarily have to be the number of symbols in the substitution. Thus many heuristics based on weighting functions for indexed entries are supported by selective substitution trees. For instance, we could define a weighting function which assesses constant symbols with 2 and other symbols with 1. Obviously, ground substitutions will be preferred instead of equal-sized substitutions with variables.

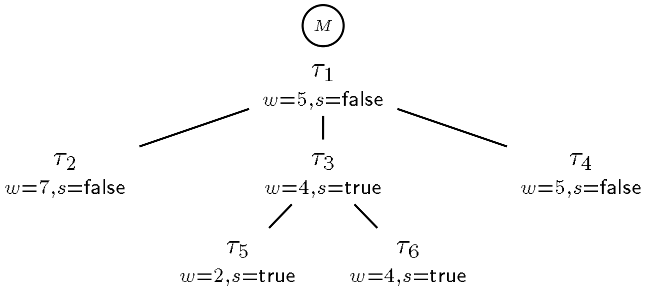


Figure 4.3: A Selective Substitution Tree

Figure 4.3 shows a selective substitution tree. The two lightest entries τ_5 and τ_6 of M have been selected and marked with true. The mark true has also been propagated to the common ancestor τ_3. The other nodes of M contain unselected entries indicated by false. The weight w and the state s of the root indicate that the lightest unselected entry in M has weight 5.

4.3 Indexing Operations

```
algorithm RESOLVE(clause graph $CG_S$)
 begin
    while proof not found do
       ⟨ Let $e = (v, (\tau, \sigma), w) \in E$ be a link in $CG_S = (V, E)$ ⟩
       ⟨ Let $u \in V$ : $\mathsf{clause}(v) = \{u, v_1, \ldots, v_m, v\}$ be the current receiver literal ⟩
       ⟨ Let $\Theta_u, \Theta_{v_1}, \ldots, \Theta_{v_m}$ be already merged indexes for $u, v_1, \ldots, v_m, v$ ⟩
       receive $\Sigma_u$ for $u$
       $(\Sigma'_u, \Theta'_u) = \mathtt{subsume}(\Sigma_u, \Theta_u)$
       $\Theta_u = \mathtt{union}(\Sigma'_u, \Theta'_u)$
       $\Gamma = \mathtt{multi\_merge}(\Sigma'_u, \Theta_{v_1}, \ldots, \Theta_{v_m})$
       ⟨ Let $\Gamma_{v,w}$ be an index of all resolved substitutions for link $e$ ⟩
       $(\Gamma', \Gamma'_{v,w}) = \mathtt{subsume}(\Gamma, \Gamma_{v,w})$
       $\Gamma_{v,w} = \mathtt{union}(\Gamma', \Gamma'_{v,w})$
       ⟨ Let $\Delta_{v,w}$ be an index of already sent substitutions for link $e$ ⟩
       $\Delta = \mathsf{strategy}(\Gamma_{v,w} \setminus \Delta_{v,w})$; $\Delta_{v,w} = \Delta_{v,w} \cup \Delta$
       $\Sigma = \mathtt{apply}(\Delta, \tau, \sigma)$
       send $\Sigma$ to $w$
 end
```

Figure 4.4: An Algorithm for the Transition Rule RESOLVE

In this section we present several indexing operations for substitution trees. These indexing operations efficiently implement the set operations of the ur-resolution rule on substitution sets introduced in Chapter 3.

Recall the transition system PURR with subsumption in Definition 3.3.6. The according informal algorithm for the transition rule RESOLVE is depicted in Figure 4.4. We omitted the sets *REC*, *MRG*, *RES*, and *SNT* in the algorithm for reasons of simplicity. Since substitution sets are implemented as indexes we also refer to the substitution sets as indexes. We briefly discuss the main steps of the algorithm.

As the transition rule RESOLVE can be applied to all outgoing links of non-unit clauses in a clause graph, the algorithm arbitarily chooses an appropriate link e in the graph. Literal u of the nucleus is considered as the receiver literal. The index Σ_u contains received substitutions which have not yet been considered for ur-resolution.

In line 8 input subsumption is performed with Σ_u and the index Θ_u of already received and merged substitutions for u. In the following line the results are united to a modified Θ_u. In line 10 the simultaneous unifiers of the remaining received substitutions Σ'_u and the already merged substitutions of the other literals are computed and stored as common instances in Γ. In the next line output subsumption is performed with Γ and the index $\Gamma_{v,w}$ of formerly resolved substitutions for link e. The remaining substitutions in Γ' represent the new ur-resolvents. In line 13 new and old ur-resolvents are united to a modified $\Gamma_{v,w}$. At this point the ur-resolution step is finished.

Finally, a selection of ur-resolvents in $\Gamma_{v,w}$ is considered for subsequent ur-resolution steps. In line 15 the 'best' unselected ur-resolvents are selected according to a certain strategy. The substitution set $\Delta_{v,w}$ contains the formerly selected resolvents. Note that the set $\Delta_{v,w}$ actually is not required. In practice, this set is implemented by a marking scheme in the index $\Gamma_{v,w}$. In line 16 the substitutions in Σ which belong to the receiver literal w are computed with the selected resolvents Δ and the link substitutions τ and σ. In other words, we anticipate the application of selected resolvents as electrons in subsequent ur-resolution steps. The substitutions in Σ represent the result of combining ur-resolvents with the receiver literal w.

The algorithm RESOLVE reveals the required set operations which have been implemented as indexing methods:

1. **Subsumption.** In line 8 and 12 subsumption is performed between two indexes. Considering subsumption as an $n{:}m$ indexing task corresponds to maintaining two indexes M and N and to delete in M all instances of substitutions stored in N, and vice versa.

2. **Union.** In line 9 and 13 substitutions of one index are inserted into another index. The union of indexes is considered as an $n{:}m$ indexing task in order to maintain two indexes M and N and to add the substitutions stored in N to M.

3. **Multi-Merge.** In line 10 the computation of simultaneous unifiers corresponds to an indexing operation on n indexes. We refer to this indexing operation as the multi-merge on n indexes M_i. The multi-merge computes all unifiable combinations of substitutions in the M_i and returns the according common instances in a new index.

4. **Selection.** The selection of substitutions according to a certain strategy in line 15 is represented as an indexing operation on two indexes M and N. The selected substitutions

in N are added to M. In order to avoid multiple selections of the same substitutions, the already selected substitutions are marked in N as selected.

Note that most of the indexing operations are defined on standard substitution trees. The additional effort to maintain linear weighted substitution trees has been omitted, as these extentions only improve the efficiency of certain operations. Only the selection operation explicitly requires the features of selective substitution trees.

4.3.1 Subsumption

Considering subsumption as an $n{:}m$ indexing task [Gra96] corresponds to maintaining two indexes M and N and to delete in M all instances of substitutions stored in N. We can accomplish this task in a very elegant and efficient way by using substitution trees.

Definition 4.3.1 (Subsumption)
Let M and N be two substitution trees. The sets of variables occurring in the trees must be disjoint. The function $\texttt{subsume}(M, N)$ returns a modified version of M in which all instances of substitutions stored in N are deleted.

$$\begin{array}{rcl}
\texttt{subsume}(\varepsilon, N) & := & \varepsilon \\
\texttt{subsume}(M, \varepsilon) & := & M \\
\texttt{subsume}(M, N) & := & \texttt{subs}(M, N, \emptyset)
\end{array}$$

The subsumption is defined using an auxiliary function $\texttt{subs}(M, N, \rho)$ that actually traverses the trees and that has a substitution as an additional parameter. The function $\#_1$ returns the first element of a tuple.

$$\begin{array}{rcl}
\texttt{subs}((\tau^M, \emptyset), N, \rho) & := & \varepsilon \\
 & & \text{if } \exists \sigma \in \mathsf{I}^*(\tau^M, \rho) \\
 & & \text{and } \exists\ (\mu, \emptyset) \in \texttt{search}(N, \rho\sigma, \mathsf{U}^*) \\
\texttt{subs}((\tau^M, \emptyset), N, \rho) & := & (\tau^M, \emptyset) \\
\texttt{subs}(M, (\tau^N, \emptyset), \rho) & := & \texttt{delete}'(M, \rho\sigma, \mathsf{I}^*) \\
 & & \text{if } \exists \sigma \in \mathsf{U}^*(\tau^N, \rho) \\
\texttt{subs}((\tau^M, \Sigma^M), (\tau^N, \Sigma^N), \rho) & := & \texttt{repair}(\tau^M, \{M_1', \ldots, M_m'\}) \\
 & & \text{if } \exists \sigma' \in \mathsf{I}^*(\tau^M, \rho) \\
 & & \text{and } \exists \sigma'' \in \mathsf{U}^*(\tau^N, \rho\sigma') \\
 & & \text{and for all } M_i \in \Sigma^M\text{:} \\
 & & M_i' = \#_1(\langle M_i, \Sigma^N, \rho\sigma'\sigma''\rangle \downarrow_{\texttt{SUBSUME}})
\end{array}$$

For the traversal of the tree we use the two different test functions I^* and U^*. They are modified versions of the original functions I and U in the sense that both functions do not need to perform an occur-check and that I^* may bind indicator variables occuring in N but not those occuring in M. In order to backtrack if a subtree has already been deleted, we use the transition rule SUBSUME.

$$\texttt{SUBSUME:}\qquad \frac{\langle M, \Sigma \uplus \{N\}, \rho\rangle}{\langle \texttt{subs}(M, N, \rho), \Sigma, \rho\rangle} \text{ if } \; M \neq \varepsilon$$

Subsumption has to consider three major situations occurring during the traversal of the trees. First, in tree M we may find a leaf node. In this situation we have to check if tree N contains a generalization in the corresponding subtree. If this is the case, the leaf node in M is deleted. Second, tree M is not a leaf node, but the corresponding node in N is. Here we simply call a deletion routine that deletes all instances of the current bindings in M. Third, if two inner nodes are considered, we proceed by considering all possible combinations of subnodes until tree M has been completely deleted or no more combinations are available.

The definition of subsumption is not easy to understand. We give an example: Suppose we have to deal with the two trees M and N depicted in Figure 4.5 and we would like to compute the tree resulting from $\texttt{subsume}(M, N)$. In tree M we maintain the substitutions $\{u \mapsto f(a, b)\}$, $\{u \mapsto f(x, c)\}$, and $\{u \mapsto f(d, c)\}$. Tree N contains the substitutions $\{u \mapsto f(a, c)\}$, $\{u \mapsto f(a, y)\}$, and $\{u \mapsto f(z, c)\}$. Obviously, the substitution $\{u \mapsto f(a, y)\}$ stored in N subsumes the substitution $\{u \mapsto f(a, b)\}$ stored in M. Moreover, $\{u \mapsto f(z, c)\}$ subsumes the two substitutions $\{u \mapsto f(x, c)\}$ and $\{u \mapsto f(d, c)\}$. Hence, the tree resulting from subsumption should be empty.

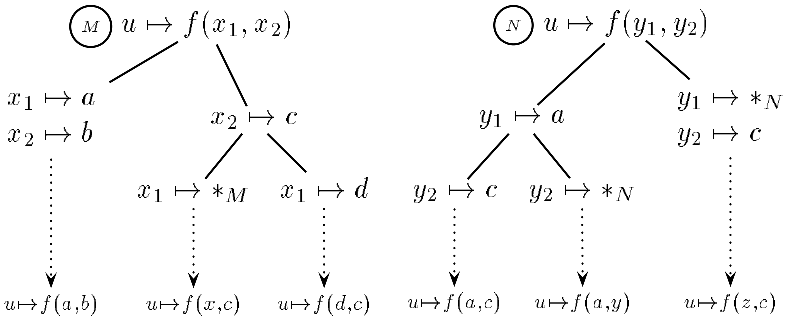


Figure 4.5: Subsumption as an $n{:}m$ indexing task

We start at the root node of tree M where the function I^* establishes the binding $\{u \mapsto f(x_1, x_2)\}$. Considering the root of N the test function U^* creates the bindings $\{y_1 \mapsto x_1, y_2 \mapsto x_2\}$. Using the transition SUBSUME we recursively traverse the subtrees. First, we consider the left subtree in M where the test function I^* yields the bindings $\{x_1 \mapsto a, x_2 \mapsto b\}$. Keeping the current bindings, the left subtree of N is traversed searching for leaf nodes that correspond to substitutions more general than $\{u \mapsto f(a, b)\}$. Such a substitution is found in $\{u \mapsto f(a, y)\}$ and the leaf node representing $\{u \mapsto f(a, b)\}$ is deleted. The transition SUBSUME is not applicable any more and from now on the right subtree of M is considered. Since the left subtree of N does not yield any deletions (because the test function I^* must not bind the variable $*_M$), we immediately consider the right subtree of N, which is a leaf marked with $\{y_1 \mapsto *_N, y_2 \mapsto c\}$. Applying function U^* on this node yields the bindings $\{x_2 \mapsto c, x_1 \mapsto *_N\}$. According to the definition of subsume we delete all instances in the right subtree of M. Finally, both subtrees of M have been deleted and repairing the resulting tree leads to ε, i.e. tree M has been completely deleted.

4.3.2 Union

The union of indexes is considered as an $n{:}m$ indexing task [Gra96] to maintain two indexes M and N and to add the substitutions stored in N to M.

Definition 4.3.2 (Union of Substitution Trees)
Let M and N be two substitution trees. The sets of variables occurring in the trees must be disjoint. The function $\texttt{union}(M, N)$ adds the substitutions stored in tree N to M.

$$\begin{aligned} \texttt{union}(M, \varepsilon) &:= M \\ \texttt{union}(M, N) &:= \texttt{add}(M, N, \emptyset) \end{aligned}$$

The insertion is defined using an auxiliary function $\texttt{add}(M, N, \rho)$ that actually traverses the trees and that has an additional parameter containing a substitution. The function $\#_1$ again returns the first element of a tuple.

$$\begin{aligned} \texttt{add}(M, (\tau^N, \emptyset), \rho) &:= \texttt{insert}(M, \rho \bullet \tau^N) \\ \texttt{add}(M, (\tau^N, \Sigma^N), \rho) &:= \#_1(\langle M, \Sigma^N, \rho \bullet \tau^N \rangle{\downarrow_{\texttt{UNION}}}) \end{aligned}$$

The following transition rule repeatedly modifies tree M.

$$\texttt{UNION:} \qquad \frac{\langle M, \Sigma \uplus \{N\}, \rho \rangle}{\langle \texttt{add}(M, N, \rho), \Sigma, \rho \rangle}$$

The idea of the definition is to traverse tree N and to recompute the stored substitutions. Whenever a leaf node is reached, we insert the corresponding substitution into tree M using the regular insertion procedure.

When we first thought about inserting a tree into another, we wanted to perform the insertion in a merge-like manner: We tried to create an algorithm that traverses the two trees in parallel hoping to be able to do lots of insertions at a time. However, the technique presented above, which only traverses one of the trees, has to be preferred because the order in which auxiliary variables are mapped to terms may differ in two different trees. Suppose, for instance, there is a path $\{u \mapsto f(x_1, x_2)\} - \{x_1 \mapsto a\} - \{x_2 \mapsto b\}$ in M and the substitution represented by the path $\{u \mapsto f(y_1, y_2)\} - \{y_2 \mapsto c\} - \{y_1 \mapsto a\}$ in N has to be inserted. Obviously, the information that the first argument of the codomain is the constant a can be shared. However, a merge-like algorithm could not detect this: First, we match from $f(x_1, x_2)$ to $f(y_1, y_2)$ and create bindings for x_1 and x_2. Second, we establish the bindings $y_1 \mapsto a$ and $y_2 \mapsto c$. Finally, the test for the substitution $\{x_2 \mapsto b\}$ fails. As a consequence, the resulting tree contains a lot of redundant information. Moreover, the failure could be detected much earlier if the information on the substitution to be inserted was complete. Even worse, a merge-like algorithm cannot employ an insertion heuristic, not even the simple first-fit technique.

4.3.3 Multi-Merge

The multi-merge [Gra96] is a generalization of the merge operation [Ohl90] on two substitution trees. The merge operation computes the set of compatible substitutions stored in two different trees. Substitutions are compatible if the codomains of identical variables in the two substitutions are simultaneously unifiable. For example, the substitutions $\{x \mapsto f(u,b), z \mapsto h(w)\}$ and $\{x \mapsto f(a,v), y \mapsto g(v)\}$ are compatible and the result of the merge of the two substitutions is the common instance[1] of the two original substitutions $\{x \mapsto f(a,b), y \mapsto g(b), z \mapsto h(w)\}$.

Suppose we want to merge three substitution trees M, N, and O. Using the ordinary merge operation for two trees, we first merge M and N. The resulting tree finally is merged with O. However, a great advantage of substitution trees is that the merge does not necessarily have to be performed on just two trees in a single merge operation. Instead of performing two merges and creating an intermediate result, we use a backtracking algorithm that traverses the three trees in parallel. In this way, we avoid the creation of intermediate results.

The multi-merge operation takes an arbitrary number of substitution trees and traverses the trees in parallel. To this end we test nodes on the same level in all of the trees. If a combination of leaf nodes is reached, the resulting common instance can be stored in a new substitution tree.

Definition 4.3.3 (Multi-Merge)
Let $TS = \{(\tau_1, \Sigma_1), \ldots, (\tau_m, \Sigma_m)\}$ be a set of substitution trees with $m > 1$. Let SN be a tuple with arbitrary arity containing leaf nodes only. The concatenation $\circ$ of an m-tuple TM and an n-tuple TN is an $m+n$-tuple $TT = TM \circ TN$. Moreover, let $M = (\tau, \Sigma)$ be a single substitution tree and ρ a substitution. The retrieval function `multi-merge` is recursively defined as follows:

$$\texttt{multi-merge}(TS) \;:=\; \texttt{multi}(TS, (), \emptyset) \tag{4.1}$$

$$\texttt{multi}(\{(\tau, \Sigma)\}, SN, \rho) \;:=\; \{SN \circ (N) \mid N \in \texttt{search}((\tau, \Sigma), \rho\sigma, \mathsf{U})\} \quad \text{if } \exists \sigma \in \mathsf{U}(\tau, \rho) \tag{4.2}$$

$$\texttt{multi}(\{(\tau_1, \emptyset), \ldots, (\tau_n, \emptyset), (\tau_{n+1}, \Sigma_{n+1}), \ldots, (\tau_{m-j}, \Sigma_{m-j})\}, SN, \rho) := \tag{4.3}$$

$$\{((\tau_1, \emptyset), \ldots, (\tau_n, \emptyset), (\tau_{n+1}, \Sigma_{n+1}), \ldots, (\tau_{m-j}, \Sigma_{m-j})) \circ SN\} \cup$$
$$\bigcup_{N_{n+1} \in \Sigma_{n+1}, \ldots, N_{m-j} \in \Sigma_{m-j}} \texttt{multi}(\{N_{n+1}, \ldots, N_{m-j}\}, ((\tau_1, \emptyset), \ldots, (\tau_n, \emptyset)) \circ SN, \rho\sigma_1 \ldots \sigma_{m-j})$$
$$\text{if } 0 \le j < m-1 \text{ and } \exists \sigma_1, \ldots, \sigma_{m-j}.\ \sigma_1 \in \mathsf{U}(\tau_1, \rho) \text{ and } \forall i.\ 1 < i \le m-j.\ \sigma_i \in \mathsf{U}(\tau_i, \rho\sigma_1 \ldots \sigma_{i-1})$$

The result of the function `multi-merge` is a set of m-tuples where m is the number of merged substitution trees. Every m-tuple contains tree nodes which have successfully been passed while traversing the trees. An m-tuple of leaf nodes represents the successful simultaneous unification of m substitutions in the trees.

Suppose we perform a multi-merge operation on m substitution trees. The first rule 4.1 initializes the auxiliary function `multi` with an empty tuple SN of leaf nodes and an empty substitution ρ. The substitution trees which have finally been traversed to leaf nodes are considered in the tuple SN such that the tuple contains these leaf nodes. During the traversal bindings are established and stored in the substitution ρ.

The next rule 4.2 considers the case that $m-1$ substitution trees have successfully been traversed down to leaf nodes while one substitution tree M still has subtrees. In this case the

[1] In some applications, e.g. hyperresolution, the unifying substitution $\{u \mapsto a\}$ itself is also needed.

function `multi` performs an n:1 retrieval on the tree M. Note that the tuple SN contains the $m-1$ leaf nodes.

The last rule 4.3 is performed if there are more than one substitution tree left to be merged and, of course, if the current node's substitutions τ_i are simultaneously unifiable. In general, there are n trees which have been traversed to leaf nodes and $m-j-n$ trees which still have subtrees. Note that we only consider $n-j$ trees where j is the number of trees which have been traversed to leaf nodes before. Obviously, SN is a j-tuple of the according leaf nodes. The result of this rule is an m-tuple of the current tree nodes together with SN, and all tuples of subsequent steps. The function `multi` is called on all permutations of subtrees of the $m-j-n$ trees. Note that the n leaf nodes are added to the j-tuple SN.

In our application, however, we are interested in the common instances of the unifiable substitutions. These instances can easily be stored in another substitution tree. Furthermore, subsumption might be performed on these instances and on formerly produced results. In our implementation, we integrated insertion and subsumption operations in the multi-merge. These operations, together with the multi merge itself, have strong impact on PURR's time and space performance. We present an algorithm with these features and an example in Chapter 6.

4.3.4 Selection

The selection is another n:m maintenance operation on two indexes M and N where N must be a selective substitution tree. The operation corresponds to a set difference on indexes in a way that a set of lightest substitutions in N are added to M. The selected substitutions are not removed from N, but marked as selected preventing multiple selection. The marked substitutions in N are still considered for conventional retrieval. We define several auxiliary functions which maintain the consistency of the selective substitution tree N.

Definition 4.3.4 (State of Selection)
Let Σ be a set of selective substitution trees. The auxiliary predicate `allselected`(Σ) holds if all substitutions in the selective substitution trees in Σ have been selected.

$$\texttt{allselected}(\Sigma) \quad :\Longleftrightarrow \quad \forall N \in \Sigma : N = (\tau, \Sigma, w, \mathsf{true})$$

During selection the states s of inner nodes are required to be updated according to the states of selection of the subtrees. The predicate `allselected` considers the state of subtrees and provides the appropriate state for the common ancestor.

Definition 4.3.5 (Lightest Subtree)
Let Σ be a set of selective substitution trees with at least one tree containing unselected entries. The auxiliary function `lightesttree`(Σ) returns the selective substitution tree occurring in Σ which has the lightest weight w and which contains unselected entries.

$$\begin{array}{lll} N = \texttt{lightesttree}(\Sigma) & \text{if} & \exists N \in \Sigma : N = (\tau, \Sigma, w, \mathsf{false}) \text{ and} \\ & & \forall M \in \Sigma \setminus N : M = (\tau^M, \Sigma^M, w^M, \mathsf{false}) \Rightarrow w^M \geq w \end{array}$$

Definition 4.3.6 (Weight of Lightest Subtree)
Let Σ be a set of selective substitution trees. The auxiliary function `lightestweight`(Σ, w_{def}) returns the weight of the selective substitution tree occurring in Σ with the lightest weight of

the trees having unselected entries. If there are no trees in Σ with unselected entries, it returns the default weight w_{def}.

$$w = \texttt{lightestweight}(\Sigma, w_{def}) \text{ if } \begin{array}{l} \neg\texttt{allselected}(\Sigma) \text{ and } (\tau, \Sigma, w, \textsf{false}) = \texttt{lightesttree}(\Sigma) \\ \text{or } \texttt{allselected}(\Sigma) \text{ and } w = w_{def} \end{array}$$

The selection operation considers a selective substitution tree N as being separated into partitions of different weights. Each partition contains substitutions with identical weight. The lightest partition can be selected within a single retrieval operation. Note that this operation also modifies tree N to provide consistency of N due to Definition 4.2.3 of selective substitution trees. In particular, a new lightest weight w and the state s are propagated to the root. If the lightest partition is completely retrieved the weight w in the root of N corresponds to the weight of the new lightest partition. Thus a single selection operation retrieves at most the substitutions of the lightest partition.

In the following definition we introduce a function `selection` which provides a transparent selection operation of an arbitrary number of substitutions. We use an auxiliary function `partition` which repeatedly retrieves the lightest partition.

Definition 4.3.7 (Selection of Substitutions)
Let M be a substitution tree and let N be a selective substitution tree. The sets of variables occurring in the trees must be disjoint. The function $\texttt{selection}(M, N, n)$ adds the n lightest substitutions stored in tree N to M and marks these substitutions in N as selected.

$$\begin{array}{rcl} \texttt{selection}(M, \varepsilon, n) & := & (M, \varepsilon, n) \\ \texttt{selection}(M, N, n) & := & \#_1(\langle (M, N, n)\rangle\downarrow_{\texttt{PARTITION}}) \end{array}$$

$$\texttt{PARTITION:} \quad \frac{\langle (M, N, n)\rangle}{\langle \texttt{partition}(M, N, n, \emptyset)\rangle} \text{ if } n > 0$$

The selection is defined using a transition rule `PARTITION` which repeatedly calls the auxiliary function $\texttt{partition}(M, N, n, \rho)$. The function retrieves up to n substitutions of the lightest partition in N and copies the substitutions to M. The bindings established while traversing tree N are stored in the substitution ρ. If the partition contains i substitutions with $i < n$, then the transition rule `PARTITION` again calls `partition` to retrieve the remaining $n - i$ substitutions in the next lightest partition.

$$\texttt{partition}(M, (\tau, \emptyset, w, \textsf{false}), n, \rho) := (\texttt{insert}(M, \rho \bullet \tau), (\tau, \emptyset, w, \textsf{true}), n{-}1) \tag{4.4}$$

$$\begin{array}{l} \texttt{partition}(M, (\tau, \Sigma, w, \textsf{false}), n, \rho) := \\ \quad (M', (\tau, \Sigma', \texttt{lightestweight}(\Sigma', w), \texttt{allselected}(\Sigma')), n') \\ \quad \text{if } \langle M, \emptyset, \Sigma, n, \rho \bullet \tau, w\rangle\downarrow_{\texttt{SCAN}} = \langle M', \Sigma', \emptyset, n', \rho \bullet \tau, w\rangle \end{array} \tag{4.5}$$

$$\texttt{partition}(M, N, n, \rho) := (M, N, n) \tag{4.6}$$

The first rule 4.4 considers tree N as a leaf node with the minimal weight w. The according substitution is inserted in M and marked as selected in N. Inner nodes of N with unselected

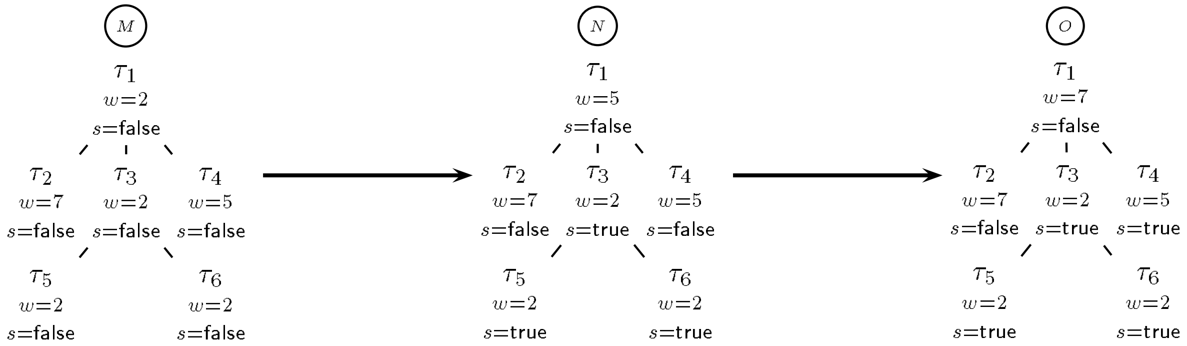


Figure 4.6: A Sequence of Selective Substitution Trees

substitutions of weight w are considered in rule 4.5. The current inner node is updated according to the lightest weight and to the state of selection of the modified subtrees Σ'. The transition system `SCAN` searches the subtrees in Σ for unselected substitutions with weight w. The last rule 4.6 considers tree N if it contains selected substitutions or too heavy substitutions only.

Definition 4.3.8 (Transition System `SCAN`)
Let M be a substitution tree and let Σ and Δ be two sets of selective substitution trees. Moreover, let ρ be a substitution containing the established bindings while traversing the trees in Δ. The transition system `SCAN` searches the set Δ for trees containing unselected substitutions of weight w. These trees match the transition rule `SELECT` which calls the function `partition` accordingly. The modified trees N' are moved to the set Σ. Other trees in Δ are moved unchanged to Σ by the transition rule `SKIP`.

`SELECT:`
$$\frac{\langle M\,,\ \Sigma\qquad\qquad,\ \Delta \uplus \{(\tau, \Sigma, w, \mathsf{false})\},\ n\ ,\ \rho,\ w\rangle}{\langle M',\ \Sigma \cup \{N'\},\ \Delta\qquad\qquad\qquad\qquad,\ n',\ \rho,\ w\rangle}$$

if $(M', N', n') = \texttt{partition}(M, (\tau, \Sigma, w, \mathsf{false}), n, \rho)$

`SKIP:`
$$\frac{\langle M,\ \Sigma\qquad\qquad\qquad\qquad,\ \Delta \uplus \{(\tau, \Sigma_d, w_d, s)\},\ n,\ \rho,\ w\rangle}{\langle M,\ \Sigma \cup \{(\tau, \Sigma_d, w_d, s)\},\ \Delta\qquad\qquad\qquad\qquad,\ n,\ \rho,\ w\rangle}$$

A sequence of selective substitution trees in different states of selection is depicted in Figure 4.6. The entries in tree M are unselected and marked with false. The tree contains three partitions of substitutions. The entries τ_5 and τ_6 belong to the lightest partition with weight 2. There is one entry τ_4 with weight 5 and one entry τ_2 with weight 7.

Suppose we select two substitutions in M yielding tree N. Obviously, the two lightest entries τ_5 and τ_6 have been selected and marked with true. The new state has also been propagated to the common ancestor τ_3. Note that the root of N contains the new minimal weight 5.

The tree O depicts the state of selection if we select another entry in N. The next lightest entry τ_4 has been selected. The new minimal weight of the tree is 7.

5 Parallelism

In general, a program working on a problem which is composed of several 'independent' parts can be divided into concurrent processes with each process working on one part of the problem. There are cases in which the performance of such a program can be improved with increasing concurrency. The possible *speedup* is limited by the degree of *dependence* inherently to the problem and by the parallel machine employed. In automated reasoning we can discover a large variety of such dependencies. They really complicate the investigation of possible improvements.

Concurrent processes working on dependent parts of a problem have to solve these dependencies with *communication*. The more processes work on a problem, the more communication usually is required. In sum, the main task during the design of a distributed parallel system is to find a reasonable balance between the degree of parallelism and the amount of communication overhead. In other words, we try to find an optimal partition of the problem set in order to obtain a truly efficient system.

In this chapter we present aspects of parallelism in the context of logic and on the basis of practical issues. Furthermore, we provide brief presentations of programming environments that support the development of parallel programs. Finally, we describe the programming library PVM that was used to implement Purr.

5.1 Notions of Parallelism

5.1.1 Parallelism in Logic

The exploitation of parallelism in the context of automated reasoning requires the investigation of the relation of logic and parallelism. A detailed discussion of parallelism in logic programming is provided by Franz Kurfeß [Kur91]. He introduces a variety of categories of parallelism, extending the traditional AND/OR-parallelism usually found in the literature. His categories might help in the discussion of further improvements of Purr. However, to understand the implementation of Purr only the knowledge of *OR-Parallelism* and *AND-Parallelism* is necessary.

OR-Parallelism. In general, OR-parallelism is achieved if possible solutions of a problem are investigated simultaneously. It sufficies to find one solution to solve the whole problem. This corresponds to the semantics of a logical or-connective. For example, a problem given as a formula $A \vee B \vee C$ is solved if one of the A, B, or C is solved. The first found solution represents a solution for the whole problem. In the context of deduction systems, OR-parallelism refers to the fact that different paths in the search space are investigated simultaneously. For example, on the level of clauses OR-parallelism means that different clauses are inferred in parallel.

The potential for parallelism is determined by the number of possible inferences and, of course, by the number of available processors. Maximal OR-parallelism is achieved, if for each possible inference a new process is initiated, documenting that the amount of potential parallelism grows during the search process. Since the number of possible inferences grows exponentially with each step in the search space, the speedup is at least limited by the number of available processors. In practice, however, the communication overhead also has an increasing influence on the overall performance.

AND-Parallelism. If a problem composed of several subproblems is solved if all subproblems are solved, then AND-parallelism corresponds to the concurrent computation of these subproblems. This definition corresponds to the semantics of the logical and-connective. A problem given as a formula $A \wedge B \wedge C$ is solved if each A, B, and C is solved. In the field of deduction systems, AND-parallelism refers to the concurrent computation of a single inference. An example is the concurrent computation of a simultaneous unifier within an inference rule. The simultaneous unifier of substitutions σ_1, σ_2, σ_3, and σ_4 can be computed by unifying two pairs σ_1, σ_2 and σ_3, σ_4 concurrently. Finally, the resulting unifiers $\sigma_{1,2}$ and $\sigma_{3,4}$ also have to be unified.

Problems arise when the substitutions share variables with each other. For example, incompatible unifiers $\sigma_{1,2}$ and $\sigma_{3,4}$ can be detected by exchanging additional information among the unification processes. Thus the concurrent processes generally are not independent which limits the degree of parallelism. If inferences are represented by clauses, AND-parallelism usually provides a reasonable speedup only on large clauses.

5.1.2 Parallelism in Practice

We now give an overview of notions and basic definitions that occur in context with parallel programming. We discuss the two most important models of parallelism and go on with the general properties of parallel programs. The definitions are mainly based on the work of Hwang [Hwa93] and Carriero and Gelernter [CG90].

5.1.2.1 Parallel Programming Models

The parallel programming model in Figure 5.1 provides a simplified and transparent view of the computer hardware/software system. A *program* is a collection of *processes* forming the basic computational units of the program. Parallelism is exploited depending on how *interprocess communication* (IPC) is implemented.

Shared-Variable Communication. The *shared-variable model* is based on the use of shared variables in a common memory for IPC. The shared variable model is also referred to as the

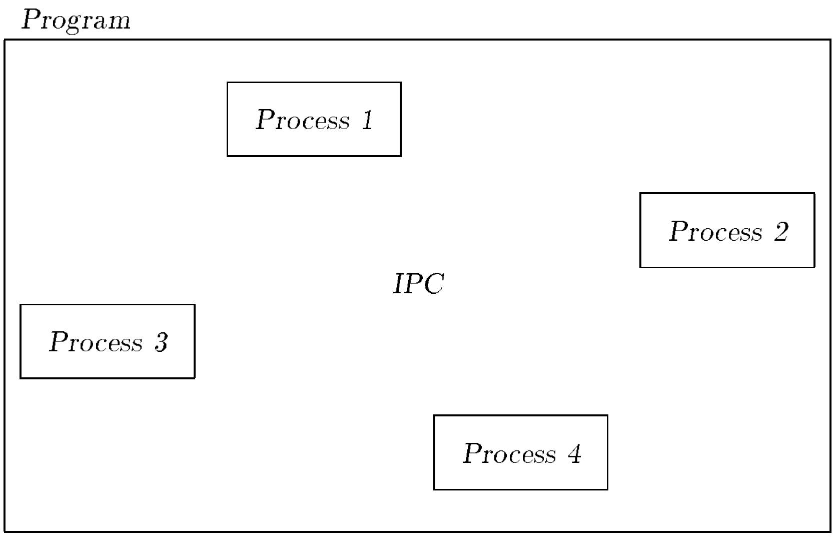


Figure 5.1: A Parallel Programming Model

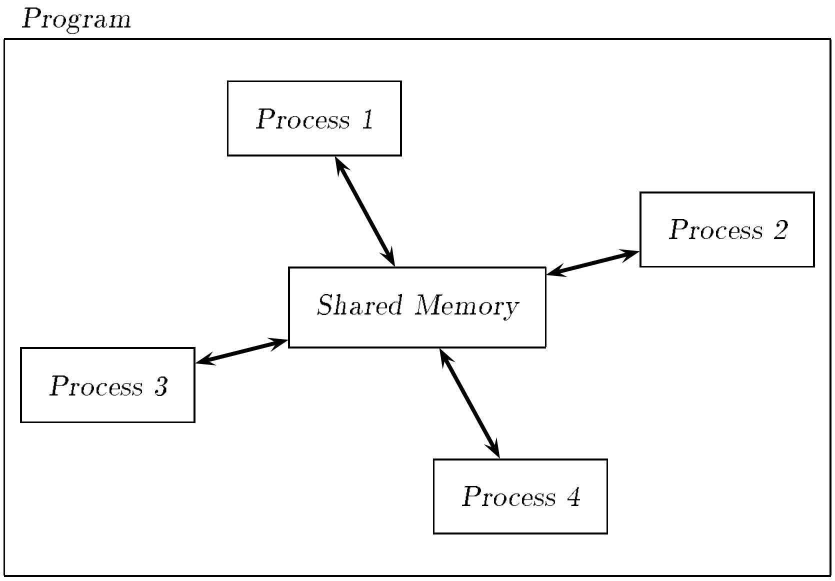


Figure 5.2: Shared-Memory Model

shared memory model which actually is a more technical term. Nonetheless, it is commonly accepted. Shared-variable IPC demands the use of shared memory and mutual exclusion among multiple processes accessing the same set of variables at a time. Figure 5.2 depicts the shared-memory model.

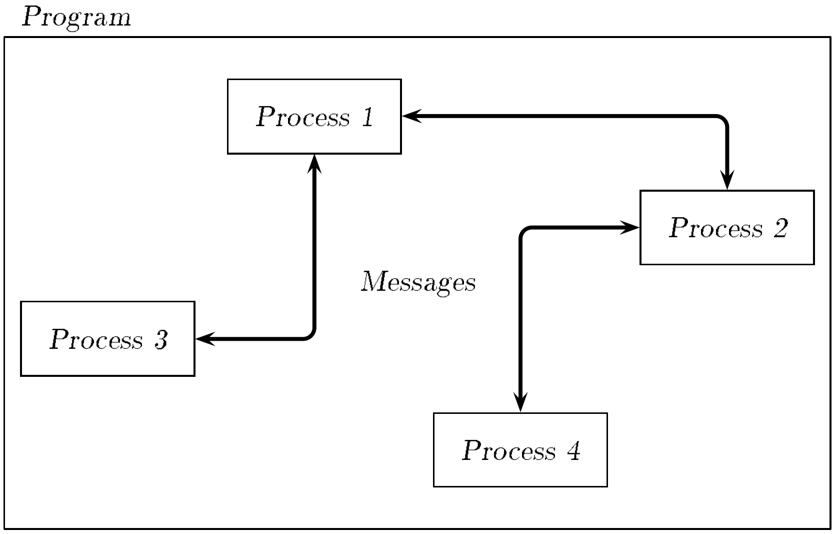


Figure 5.3: Message Passing Model

Message-Passing Model. In the *message-passing model* processes communicate with each other by passing messages through a network. Since there is no shared memory, mutual exclusion is not needed. We distinguish *synchronous* and *asynchronous message passing*. Synchronous message passing synchronizes the sender process and the receiver process in time and space. Then the data is transfered with both processes communicating at the same moment. Asynchronous message passing does not require that message sending and receiving are synchronized. Instead, asynchronous communication requires the use of buffers to hold the messages along the path of the connecting channels. Since these *message queues* are finite, the sender will eventually be blocked. On the other side, the receiver process may either be blocked in order to wait for a certain message or proceed no matter if a message arrived or not. The former is called *nonblocking receive*, the latter *blocking receive*. Figure 5.3 illustrates the message-passing model.

5.1.2.2 Properties of Parallel Programs

Parallel programs have properties that are described in terms of Dependence, Granularity, Latency, Communication Patterns, Deadlock, and Load Balancing.

Dependencies. The parallel execution of several program segments requires each segment to be independent from other segments. The so-called *data dependence* describes how different program segments depend on each other concerning the input and output data of these segments. Data dependence reduces the possible *speedup* of a parallel program. Concurrent processes working on dependent segments have to solve the dependence with *communication*.

Granularity. The *grain size* or the *granularity* of a parallel program corresponds to the amount of computation involved in one process contributing to the program. The grain size determines the smallest program segment chosen for parallel processing. Grain sizes are classified as *fine*, *medium*, or *coarse*. Usually, the finer the granularity of a program, the higher is the degree of parallelism. However, a higher degree of parallelism results in high communication demands and scheduling overhead. Ideally, the granularity of a program is not fixed but can arbitrarily be selected.

Latency. Latency is a time measure for communication overhead. For example, the time required for two processes to synchronize with each other is called the *synchronization latency*. Computational granularity and communication latency are closely related. By balancing granularity and latency, one can influence the performance of a parallel program.

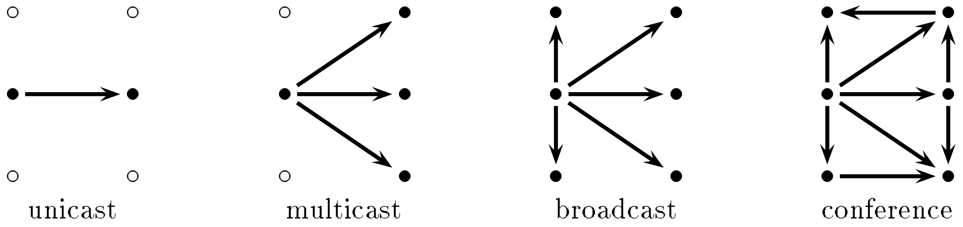


Figure 5.4: Communication Patterns

Communication Patterns. We specify four types of *communication patterns*. The one-to-one *unicast* pattern has one source and one destination. A *multicast* pattern corresponds to one-to-many communication in which one source sends the same message to multiple destinations. A *broadcast* pattern corresponds to the case of one-to-all communication. Finally, the most generalized pattern is the many-to-many *conference* communication.

Deadlock. A *deadlock* situation occurs if a program enters a state in which a process A is waiting for data to be produced by another process B while B is waiting for data produced by A. In general, we speak of a deadlock if an arbitrary-long cycle of processes exists such that each process is waiting for data provided by the previous process in the cycle. *Deadlock detection* tries to distinguish deadlock situations from others. For example, some processes might work slowly or might have been terminated for some reason.

Load Balancing. The problem of *load balancing* occurs if the amount of parallelism inherent in a program does not match the parallelism provided by the computer system. For example, a program that decomposes into four processes will not achieve a higher speedup if more than four processors are employed. *Static scheduling* techniques try to find a good static distribution of the processes to processors of a parallel machine.

In sum, the design of efficient parallel systems involves the reduction of latency, the prevention of deadlock, the minimization of blocking in communication patterns, and a reasonable tradeoff between parallelism and communication overhead.

5.2 Parallel Programming Systems

A few years ago, the development of parallel programs required extremely high effort: One had to invest in expensive parallel hardware and highly specialized programming environments. Usually, programs developed for a specific machine could not be used on other systems.

Parallel programming systems try to abstract from underlying hardware. Sometimes they even allow heterogeneous clusters of workstations. The portability of programs on top of an abstract parallel system has significantly been improved. Nevertheless, abstraction usually implies a loss of performance which we will discuss later.

Since our parallel prover PURR should run on many different hardware platforms (including workstation clusters), we decided to develop the system on top of a parallel programming system. The following overview sketches the systems we took into consideration.

5.2.1 Overview

In this thesis we can only present a small survey on the wide range of parallel programming systems. An exhaustive collection has been presented by Turcotte [Tur93]. A comprehensive overview of parallel computing issues can also be found on the world wide web [IPC].

Standards. Just before we started to implement PURR, the *Message Passing Interface* (MPI) forum developed an interface standard for message passing systems. There is an official world wide web home page reporting that standard [MPI]. The standard became necessary to improve the portability of application software from one message passing system to another and, moreover, to improve the commercial development of tools and libraries. A first draft of the MPI standard was published by the MPI Forum [For93]. A user-oriented presentation is provided by Gropp, Lusk, and Skjellum [GLS94]. The current MPI standard definition is also available in the official home page [MPI].

However, we did not use the MPI standard in PURR because available implementations have not been tested as intensive as other parallel programming systems. At this time, MPI supported less platforms than other systems. Experiments describing the loss of performance or stability when using an implementation of the MPI standard were not available. Therefore, we considered other parallel programming systems with the following areas being of major concern:

- Paradigm, i.e. message passing or shared memory.
- Efficiency.
- Support.
- Ease of use.
- Portability, i.e. support of different hardware.

TCGMSG. The *Theoretical Chemistry Group Message Passing System* [Har91] is a very efficient message passing system with communication over direct, point-to-point TCP/IP sockets. TCGMSG is a simple message passing system providing the user with an easy to use environment. Built in global operations, e.g. global summation, simplify the implementation of data parallelism. However, global operations are not required in Purr. A significant drawback of the package is that it seems to be poorly supported.

p4. The package *p4* [BE93] is a distributed computing environment providing both the programming of a variety of MIMD machines and the computation in workstation clusters. This package includes shared memory and message passing primitives. The communication protocoll is based on dynamic TCP sockets. Just like the TCGMSG package, p4 also provides global operations. The differences in the ease of use compared to TCGMSG are relatively small. The support of p4 is better than the support of TCGMSG. p4 runs on a wide range of platforms.

PVM. The *Parallel Virtual Machine* [GBD$^+$94] is a pure message passing system. The PVM package is the de-facto standard for message passing systems; it has by far the largest number of users. As a main difference to the other packages, PVM is especially designed for computing on heterogeneous networks. In addition to dynamic TCP sockets for the communication protocol, PVM also provides UDP communication. Compared to the other two packages coding in PVM is more complicated: explicit buffer management has to be done by the user and task identifiers have to be maintained. The support for PVM is outstanding since it is based on the experience of a large number of users. PVM supports many different platforms.

Linda. There are many packages also supporting the shared memory paradigm. An important approach is the associative, virtual shared memory system called *Linda* [Gel85]. There are many implementations for Linda, for example, the commercial product *C-Linda* or *p4-Linda* [BLL93] which implements Linda on top of p4. There is another public domain implementation of Linda called *POSYBL* [Sch92]. All these packages (except for p4-Linda) have been compared in detail [Mat94].

5.2.1.1 Assessment

The main reason to prefer the message passing paradigm is that our approach to an inference system does not require any common data. If we identify nodes in a clause graph with concurrent processes, communication is represented by exchanging messages via links in the graph.

The TCGMSG package seems to be the most efficient system of this survey followed by PVM and p4. The best support and portability is provided by the PVM package. Since the shared-memory based Linda is the most generalized approach, it is the package which is easiest to use. However, concerning the ease of use the differences of the remaining packages are relatively small.

Finally, the PVM package appeared as a good compromise between efficiency, support, ease of use, and portability. PVM is an efficient message passing system with good support. Unfortunately, PVM is not as easy to use as a Linda-based system. However, we only need a fraction of the capabilities of PVM and thus the effort to become familiar with PVM was acceptable. PVM also supports a large variety of multiprocessor platforms and parallel computing on workstation clusters. The next section contains some more details about PVM.

5.2.2 PVM

The PVM (Parallel Virtual Machine) package [GBD+94] is a software system providing distributed processing capabilities with message passing primitives. The system profits from a large user base and intensive discussion [Mat94] [DGMJ93] [MP95].

PVM supports a wide range of different platforms and also a variety of different networks. The system abstracts from a heterogeneous network of computers to a collection of hosts. Each processor in the network forms one host independently whether the processor is a single workstation or a processor in a multiprocessor machine. Message routing and data conversion for incompatible architectures are handled transparently. Thus PVM theoretically permits a network of heterogeneous computers to be used as a single parallel computer, a parallel virtual machine.

The PVM system is composed of two parts, a library of PVM interface functions and a daemon program. The library contains user-callable routines and must be linked with the application program. The daemon resides on all machines in the heterogeneous network and makes up the virtual machine. A PVM console task enables the user to control the virtual machine, i.e. to add and remove machines, to monitor status information, etc. After making up a virtual machine, PVM applications may be started. PVM can handle multiple users and overlapping virtual machine configurations.

Next, we focus on a selection of library methods of PVM; for further details of PVM see [GBD+94]. PURR only requires a small subset of the PVM routines:

Process initiation and

Point-to-point communication.

Process Initiation. In PURR process initiation refers to a master process starting several slave processes at the beginning of the proof procedure. PVM provides a library call which spawns a certain number of slave processes. More sophisticated features like the mapping of specific processes to specific processors currently are not exploited by PURR. Thus PVM decides where processes are executed. During the proof the number of slave processes remains fixed until all processes are terminated by the master process. Thus processes are started at the beginning of the proof search only.

Communication Requirements. The slave processes in PURR use unicast and multicast communication patterns. The master process also uses broadcasting to control the slave processes. Process groups are not needed. A send operation in PVM is composed of two phases. First, the message has to be packed into a buffer. The packing routine may perform a data conversion called XDR encoding in order to avoid incompatibilities in the representation of data on different architectures. If the user knows that all machines in the network understand the same format, the data conversion might be disabled. Second, the buffer contents is sent to its receiver process. Additionally, for messages consisting of a vector of elements of equal type PVM provides a more efficient send operation which includes packing. Since a message in PURR is a homogeneous vector of integers, this method is used to send messages.

Receiving a message works in the reversed direction. A message is received either with a blocking or non-blocking routine and has to be unpacked. Unfortunately, PVM only provides a blocking routine for receiving and unpacking a message simultaneously. In order to enable a non-blocking receive mechanism with unpacking we use a PVM function which only checks the message queue for a certain type of message and calls the blocking routine when needed.

6

The Prover

In the last three chapters we have presented the theoretical background of Purr. We introduced a modified unit resulting resolution scheme working on substitution sets. The modified inference rule addresses the main issues of this work: The investigation of term indexing as a fundamental base of reasoning operations and the exploitation of parallelism in automated theorem proving.

In the first part of this chapter detailed algorithms of the indexing operations in Purr are presented. The algorithms have been introduced as *advanced indexing operations* by Graf and Meyer [GM96]. In the second part we investigate the potential for parallelism and combine the presented ideas such that they can be used in the implementation of Purr.

In the last section we present a data structure called *context* which can be used to maintain variable bindings efficiently. Finally, we discuss how substitution trees can be used as a compact protocol for the exchange of sets of substitutions among concurrent processes.

6.1 Indexing Algorithms

We present algorithms for the four indexing operations `subsume`, `union`, `multi-merge`, and `selection`. The subsumption operation deletes in one substitution tree all instances of substitutions that occur in another substitution tree. The union of two substitution trees integrates all entries of one tree into the other tree. The multi-merge operation computes simultaneous unifiers of substitutions which are stored in several substitution trees. The result of such a multi-merge is a substitution tree containing the common instances of the unified substitutions. The selection operation searches a substitution tree for entries with lowest "weight" and adds these entries to another substitution tree.

6.1.1 Subsumption

The subsumption operation is an n:m maintenance task on two indexes M and N. Suppose index N contains substitutions which have been tested for subsumption and index M contains new substitutions. Forward subsumption corresponds to the deletion of all instances of N in M. To this end we traverse M and N in parallel. During the traversal we map variables occurring in N to subterms stored in M. This mapping is exactly the same as just looking for generalizations in a n:1 retrieval. Whenever we reach a leaf node in the index M we may delete it. Note that the deletion in the index M can cause the whole tree to be removed. We obtain backward subsumption by simply changing the roles of M and N.

Consider the algorithm for subsume depicted in Fig. 6.1. In our implementations a stack *STK* of variable bindings is maintained. The function match(N, *STK*, *BINDINGS*) checks if N's substitution is a generalization of the current bindings. New bindings are established on the stack *STK*. The function match_reverse(M, *STK*, *BINDINGS*) is accordingly defined to test whether M's substitution is an instance of the bindings. The function backtrack(*STK*, *BINDINGS*) resets the stack *STK* by popping *BINDINGS* bindings from it. Note that before the subsumption operation is called the function match_reverse(M, *STK*, *BINDINGS*) and match(N, *STK*, *BINDINGS*) have been successfully called. In this way we avoid unnecessary recursive calls in the algorithm.

```
 1  algorithm subsume(tree $M$,tree $N$,stack $STK$)
 2   begin
 3      ⟨Assume match_reverse($M$) and match($N$) hold⟩
 4      if is_leaf($M$) then
 5          if genexist($N, STK$) then
 6              $M = \varepsilon$
 7      elsif is_leaf($N$) then
 8          $M$ = delete_instances($M, STK$)
 9      else
10          forall subtrees $M'$ of $M$ do
11              if match_reverse($M', STK, BINDINGS^M$) then
12                  forall subtrees $N'$ of $N$ do
13                      if match($N', STK, BINDINGS^N$) then
14                          $\Sigma = \Sigma \cup$ subsume($M', N', STK$)
15                      backtrack($STK, BINDINGS^N$)
16              backtrack($STK, BINDINGS^M$)
17          $M$ = repair($M, \Sigma$)
18      return $M$
19   end
```

Figure 6.1: Algorithm for subsume

Subsumption has to consider three major situations occurring during the traversal of the trees. First, in tree M we may find a leaf node. In this situation the n:1 retrieval operation genexist checks if tree N contains a generalization of the current bindings in the corresponding subtree. If this is the case, the leaf node in M is deleted (s. line 6). Second, tree M is not a leaf node, but the corresponding node in N is. Here we simply call a n:1 deletion routine delete_instances that removes all instances of the current bindings in M (s. line 8). Third, if two inner nodes are considered, we proceed by considering all possible combinations of subnodes until tree M has been completely deleted or no more combinations are available. Note that in line 17 node M has to be "repaired" if all subtrees of M but one have been deleted[1].

[1] According to definition 4.2.1 of substitution trees an inner node has at least two sons.

6.1.2 Union

The union is an n:m maintenance operation of two indexes M and N. The entries stored in tree N are added to M.

```
1   algorithm union(tree $M$,tree $N$,stack $STK$)
2   begin
3       bind($N, STK$, BINDINGS)
4       if is_leaf($N$) then
5           $M$ = insert($M, STK$)
6       else
7           forall subtrees $N'$ of $N$ do
8               $M$ = union($M, N', STK$)
9       backtrack($STK, BINDINGS$)
10      return $M$
11  end
```

Figure 6.2: Algorithm for union

Consider the algorithm for union in Fig. 6.2. In order to reconstruct the substitutions in tree N, the function bind establishes N's substitution on the stack (s. line 3). If node N is a leaf, then insert adds the corresponding substitution, which is represented by the established bindings, to tree M (s. line 5). If node N is an inner node, then the subtrees of N are recursively traversed in order to find all entries of N (s. line 8).

6.1.3 Multi-Merge

The multi-merge operation computes the compatible substitutions stored in an arbitrary number of substitution trees. Substitutions are compatible if the codomains of identical variables in the substitutions are simultaneously unifiable. To this end the algorithm traverses the trees in parallel. If a combination of leaf nodes is reached, the resulting common instance of the substitutions represented by these leaves can be stored in a new substitution tree. Furthermore, subsumption might be performed thus reducing the amount of substitutions to be maintained.

Consider the algorithm multi-merge in Fig. 6.3 which employs n:1 insertion and subsumption operations. The algorithm has four parameters: A substitution tree RES, two ordered sets $CURRENT$ and $NEXT$ of substitution trees, and a stack STK of bindings. The common instances resulting from the simultaneous unification are inserted into the substitution tree RES which does not have to be empty at the beginning. The tree may contain previously obtained results which are then considered in the subsumption phase of the multi-merge operation. Initially, the ordered set $CURRENT$ contains the substitution trees to be merged whereas the ordered set $NEXT$ is empty. We assume that the substitutions of the root nodes have been successfully unified before multi-merge is called. In this way we avoid unnecessary recursive calls of the algorithm. The variable bindings of the unification are pushed on the stack STK.

The function unify(N, STK, $BINDINGS$) implements the test for unifiability by checking for each assignment $x_i \mapsto t_i$ of N's substitution $\tau = \{\ldots, x_i \mapsto t_i, \ldots\}$ whether x_i is unifiable with t_i. The bindings of variables in the unifier are pushed on the stack STK and are counted in $BINDINGS$. This unification considers variable bindings in the terms to be unified. Additionally, the function backtrack(STK, $BINDINGS$) resets the stack STK by popping $BINDINGS$ bindings

```
1  algorithm multi-merge(tree $RES$,set $CURRENT$, set $NEXT$,stack $STK$)
2  begin
3      ⟨ Let $CURRENT = \{N_i, \dots, N_m\}$ be an ordered set of trees ⟩
4      ⟨ Let $NEXT = \{N_1, \dots, N_{i-1}\}$ be an ordered set of trees ⟩
5      if $\forall N \in CURRENT \cup NEXT$ : is_leaf($N$) then
6          ⟨ Simultaneous Unifier ⟩
7          if $\neg$genexist($RES, STK$) do
8              delete_instances($RES, STK$)
9              $RES$ = insert($RES, STK$)
10     else
11         if $CURRENT = \emptyset$ then
12             $RES$ = multi-merge($RES, NEXT, CURRENT, STK$)
13         elsif is_leaf($N_i$) then
14             $RES$ = multi-merge($RES, CURRENT \setminus N_i, NEXT \cup \{N_i\}, STK$)
15         else
16             ⟨ Let $(\tau, \Sigma) = N_i$ be the root of $N_i$ ⟩
17             forall $N' \in \Sigma$ do
18                 if unify($N', STK, BINDINGS$) then
19                     $RES$ = multi-merge($RES, CURRENT \setminus N_i, NEXT \cup \{N'\}, STK$)
20                 backtrack($STK, BINDINGS$)
21     return $RES$
22 end
```

Figure 6.3: Algorithm for `multi-merge`

from it. After a successful unification at leaf nodes the function **genexist** performs n:1 forward subsumption using the established bindings. If no generalization of the found common instance σ exists in RES, the function **delete_instances** removes all instances of σ from RES by a n:1 backward subsumption. Finally, the function **insert** normalizes and inserts σ into the substitution tree RES. Note that all functions work with bindings instead of really instantiated substitutions. In this way we delay (and often avoid) the allocation of memory as long as possible.

The main idea of the algorithm `multi-merge` is to traverse the trees in parallel. All combinations of subnodes of the *CURRENT* set of inner nodes have to be considered. The subnodes which pass the test for unifiability are moved to the *NEXT* set of nodes (s. line 19). If *CURRENT* is empty we simply exchange *CURRENT* with the *NEXT* level (s. line 12). *CURRENT* leaf nodes are also moved to the *NEXT* level in order to uphold the original order of trees (s. line 14). Each combination of leaf nodes represents a simultaneous unifier which corresponds to the established bindings on the stack *STK* (s. line 5).

A sequence of stacks resulting from the simultaneous unification of substitutions stored in three substitution trees is depicted in Fig. 6.4. Originally, the stack is empty. Before we start the multi-merge algorithm, the substitutions of the root nodes have to be unified, resulting in the bindings pushed on the stack (compare stack "Init"). The sequence $\textcircled{\scriptsize A}\textcircled{\scriptsize U}\textcircled{\scriptsize X}$ denotes the tree nodes which have been considered in this step. The recursive algorithm is started on the subnodes of the root nodes. In case it succeeds in testing the current substitution for unifiability, the modified stack is marked with "Success". If a combination of leaf nodes has been found, "**Success**" is written boldface. The first common instance is $\{u \mapsto f(d, g(d))\}$ which is backward subsumed by the second common instance $\{u \mapsto f(v, g(v))\}$. The last found

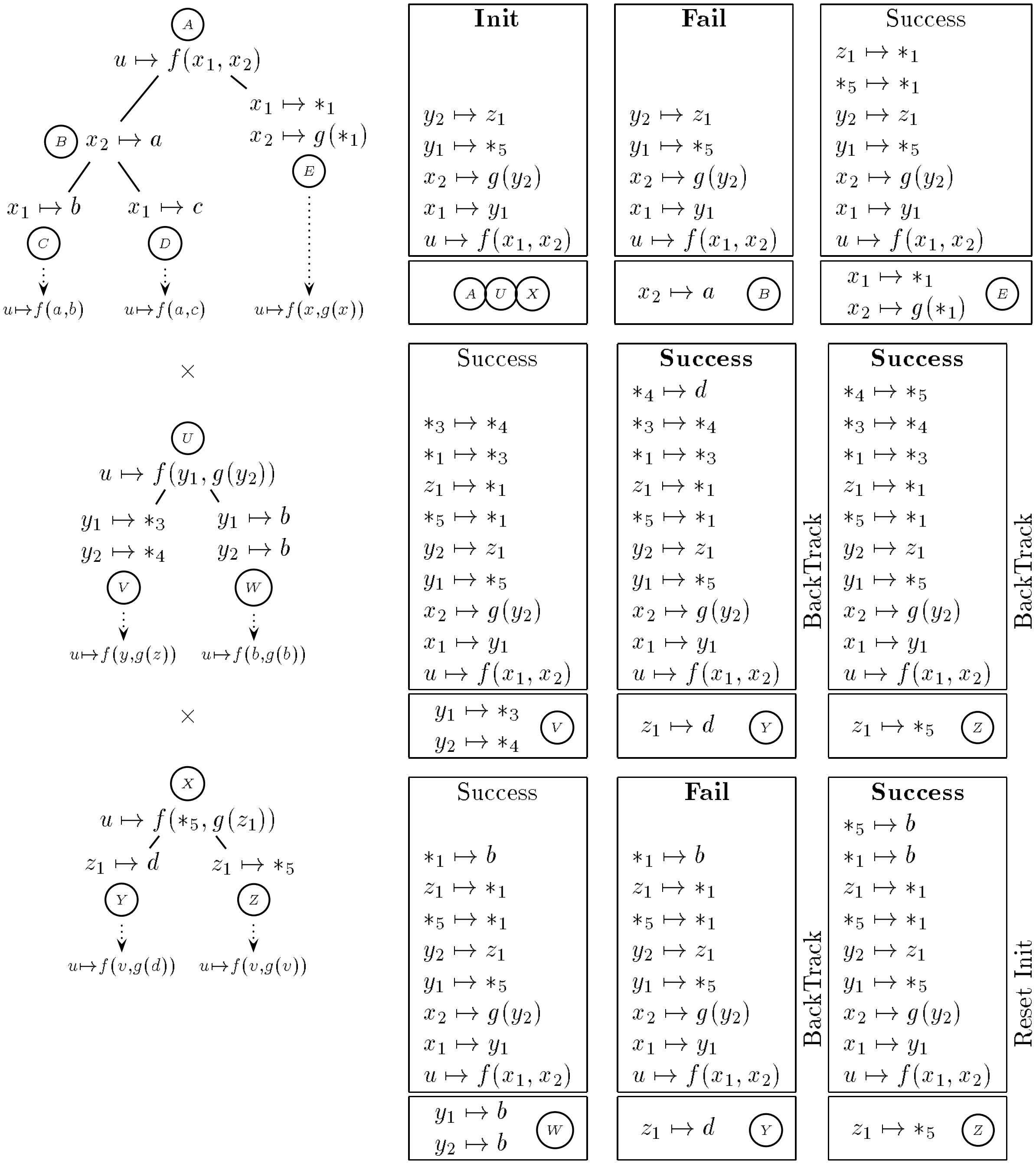


Figure 6.4: Multi-Merge with Three Substitution Trees

substitution $\{u \mapsto f(b, g(b))\}$ is forward subsumed by the second. Therefore, the result of the multi-merge is a substitution tree only containing $\{u \mapsto f(v, g(v))\}$.

6.1.4 Selection

The selection is an $n{:}m$ maintenance operation on two indexes M and N. The set of lightest substitutions occurring in N is added to M. The selected substitutions are not removed from N, but marked as selected preventing multiple selection. The marked substitutions in N are still considered for conventional retrieval.

The selection considers a selective substitution tree N as being separated into partitions of different weights. Each partition contains substitutions with identical weight. The lightest partition can be selected within a single retrieval operation. Note that this operation also modifies tree N to provide consistency of N (In particular, a new lightest weight w and the state s are propagated to the root). If the lightest partition is completely retrieved, the weight w in the root of N corresponds to the weight of the new lightest partition. Thus a single selection operation retrieves at most the substitutions of the lightest partition.

```
1  algorithm selection(tree $M$,tree $N$,nat $n$,stack $STK$)
2   begin
3      while $n > 0 \wedge N$ is unselected do
4          $(M, N, n)$ = partition($M, N, n, STK$)
5      return $(M, N, n)$
6   end
```

Figure 6.5: Algorithm for `selection`

The function `selection` depicted in Fig. 6.5 implements a selection of n substitutions in a selective substitution tree N and stores the retrieved substitutions in tree M. The auxiliary function `partition` repeatedly retrieves the lightest partition in N (s. line 4). Thus, the number n of demanded substitutions does not depend on the size of the lightest partition in N.

Consider the algorithm for `partition` in Fig. 6.6. We assume that `partition` is called only with substitution trees containing unselected substitutions. Furthermore, the current tree node N in `partition` always contains substitutions with the lightest weight w, i.e. node N is marked with weight w. Therefore, a parameter for the lightest weight in tree N is not needed in the algorithm. In order to reconstruct the selected substitutions in tree N, the function bind establishes N's substitution on the stack (s. line 4). If node N is a leaf, then insert adds the corresponding substitution represented by the established bindings to tree M (s. line 7). Note that leaf N will be marked as selected (s. line 9). If node N is an inner node, then the subtrees of N containing unselected entries with the lightest weight w are recursively searched (s. line 14). As the algorithm propagates a new lightest weight and the state of selection from the leaves to the root, the set Σ' contains the updated subtrees of N (s. line 15). The new weight of node N is the lightest weight of the subtrees in Σ' with unselected entries computed by the function `lightestweight`. The function `allselected` is true if all entries in all subtrees in Σ' have been selected (s. line 17).

```
algorithm partition(tree $M$,tree $N$,nat $n$,stack $STK$)
 begin
    ⟨ Assume that $N$ has unselected entries ⟩
    bind($N, STK$, BINDINGS)
    if is_leaf($N$) then
        $\langle N = (\tau, \emptyset, w, \mathsf{false})\rangle$
        $M = \mathsf{insert}(M, STK)$
        backtrack($STK, BINDINGS$)
        return $(M, (\tau, \emptyset, w, \mathsf{true}), n - 1)$
    else
        $\langle N = (\tau, \Sigma, w, \mathsf{false})\rangle$
        forall subtrees $N'$ of $N$ do
            if $n > 0 \wedge N'$ is unselected with weight $w$ then
                $(M, N', n) = \mathtt{partition}(M, N', n, STK)$
            $\Sigma' = \Sigma' \cup N'$
        backtrack($STK, BINDINGS$)
        return $(M, (\tau, \Sigma', \mathtt{lightestweight}(\Sigma'), \mathtt{allselected}(\Sigma')), n)$
 end
```

Figure 6.6: Algorithm for `partition`

6.2 The Implementation

The first section gives a brief overview of Purr. As the system operates in three different stages, the following sections discuss each stage seperately. The first stage is the preprocessing in which the problem set is processed and the clause graph is computed. The following reasoning phase makes up the central part of the system. The final postprocessing includes the generation of a proof scheme.

6.2.1 Overview

The implementation of Purr is based on the *master/slave* paradigm with message passing. When Purr is started, a single process called *master* process is initiated. The master process executes, controls, and terminates *slave* processes. In turn, a slave process will not execute another process. Note that in our approach the master process only initiates a certain number of slave processes just once. During the run of the system the number of slave processes remains fixed. Finally, the master process terminates all slave processes.

```
algorithm master_process(file $F$)
 begin
    ⟨ Let $CG_S = (V, E)$ be a clause graph ⟩
    $CG_S = \mathtt{preprocessing}(S)$
    ⟨ Let $\Sigma$ be a set of answer substitutions ⟩
    $\Sigma = \mathtt{reasoning\_phase}(CG_S)$
    ⟨ Let $P$ be a set of substitutions representing a proof ⟩
    $P = \mathtt{postprocessing}(CG_S, \Sigma)$
 end
```

Figure 6.7: The Master Process

Figure 6.7 shows three main phases of the master process. The first step is the preprocessing which is solely performed by the master process. The result of the preprocessing is a clause graph CG_S derived from the input set S of formulas in the file F. In the following reasoning phase, the master process uses the clause graph to determine the number of slave processes and their interconnections. Then, the master process initiates the slave processes and remains idle until a proof situation could be detected. Before the master process enters the postprocessing, all slave processes are suspended, but not terminated. The result of a successful reasoning phase is a set of answer substitutions Σ. An answer substitution instantiates one of the query clauses and represents the last substitution of a successful proof. The following postprocessing substantially includes the generation of a proof scheme P. The postprocessing solely is a sequential process directed by the master process. The slave processes merely serve as databases of inferences. In the following sections we discuss the three phases of the system in detail.

6.2.2 Preprocessing

The preprocessing in PURR creates a clause graph. The graph corresponds to the initial set of clauses. In a first step, the master process performs the preprocessing in a solely sequential computation. Concurrent processing starts after the preprocessing has been finished.

```
algorithm preprocessing(file $F$)
 begin
    Read in file $F$ and create set of clauses $S$
    $\langle$ Create clause graph $CG_S$ according to $S$ $\rangle$
    $CG_S$ = create_graph($S$)
    $\langle$ Optimize clause graph $CG_S$ $\rangle$
    delete_tautologies($CG_S$)
    delete_pure($CG_S$)
 end
```

Figure 6.8: The Preprocessing

Consider the preprocessing depicted in Figure 6.8. First, the master process reads in a file F which contains the problem set. Currently, PURR merely supports clause normal form. A set of clauses S according to the formulas in F is established. The next routine creates the corresponding clause graph CG_S. Finally, the clause graph CG_S is further optimized. In the following, we describe the individual steps in detail.

The Input Process. Since PURR is an experimental system, it requires the input to be a set of clauses in disjunctive normal form. Additionally, we added a keyword to mark some clauses as query clauses. Currently, the options of PURR are passed as parameters of the master process. Experiments with the system showed that the input language should support the user to individually affect the creation of the clause graph. Note that the number of processes and their interconnections directly depend on the clause graph. It is important to increase or reduce the 'potential' of a clause, i.e. to influence the number of processes designated to one clause. Thus the input language ideally supports an individual marking mechanism of clauses and literals. However, we discuss this issue later on the basis of experiments.

Clause Graph Creation. In Purr a clause graph provides information about how clauses might be used as inference partners in the search process. Each literal in the set of clauses is associated with a vertex in the graph. The links in the graph explicitly show potential applications of ur-resolution. Consequently, the creation of a clause graph provides the computation of these links and their corresponding test and send substitutions.

```
algorithm create_graph(clauses $S$)
 begin
     $\langle$ Let $CG_S = (V, \emptyset)$ be a clause graph corresponding to $S$ $\rangle$
     $VISITED := \emptyset$
     create_links($CG_S, \{v | v \in V \wedge \mathsf{query}_{(}v)\}$)
     return $CG_S$
 end
```

Figure 6.9: Create Clause Graph

The algorithm `create_graph` depicted in Figure 6.9 computes the labelled links on a clause graph CG_S which is initialized with the vertices associated to the literals $L_{i,j}$ in the set of clauses S. The algorithm `create_graph` introduces a global set of vertices *VISITED* which contains the worked-off vertices.

```
1   algorithm create_links(clause graph $CG_S$, vertices $V_{in}$)
2    begin
3        $\langle$ Let $CG_S = (V, E)$ be a clause graph $\rangle$
4        forall $v_{in} \in V_{in} \setminus VISITED$ do
5            $VISITED = VISITED \cup \{v_{in}\}$
6            forall $v_{out} \in V \setminus \{v_{in}\}$ do
7                if $\exists \sigma : \mathsf{literal}(v_{out})\sigma = \neg\mathsf{literal}(v_{in})\sigma$ then
8                    $E = E \cup (v_{out}, \mathtt{split\_mgu}(\mathsf{literal}(v_{out}), \mathsf{literal}(v_{in})), v_{in})$
9                    create_links($CG_S, \mathsf{clause}(v_{out}) \setminus \{v_{out}\}$)
10   end
```

Figure 6.10: Create all links leading to vertices in V_{in}

The global set *VISITED* is used by the algorithm `create_links` depicted in Figure 6.10. The algorithm is called on the clause graph CG_S and the literals of the query clauses. Recall that links in CG_S are solely established in the direction to query clauses. The recursion in `create_links` is based on the fact that a linked clause can be seen as an intermediate query clause. The recursive call in line 9 considers the remaining literals of the currently processed clause as new goals. The recursion is repeated until a unit clause is reached. The number of possible links might be reduced by restricting the direction of links from positive to negative literals, or vice versa. In line 4 we only have to consider vertices v_{in} associated with positive literals, or negative literals.

Aßmann estimates the complexity of this algorithm to $O(n^2)$ [Aßm92]. Since Purr was not developed to handle large initial sets of clauses, the square complexity should not become a problem.

Clause Graph Optimization. Obviously, a clause graph might further be optimized in many ways. We merely present two optimization routines which have been discussed in Chapter 3.

The first routine removes tautologies from the clause graph. A clause is a tautology if the clause contains two complementary literals linked by an empty unifier.

```
algorithm delete_tautologies(clause graph $CG_S$)
 begin
    ⟨ Let $CG_S = (V, E)$ be a clause graph ⟩
    forall $v \in V$ do
       if $\exists(v, (\emptyset, \emptyset), w) \in E$ with $w \in \mathsf{clause}(v)$ then
          $E = E \setminus \{e | e = (v_1, (\tau, \sigma), v_2) \in E \wedge (v_1 \in \mathsf{clause}(v) \vee v_2 \in \mathsf{clause}(v))\}$
          $V = V \setminus \mathsf{clause}(v)$
 end
```

Figure 6.11: Delete tautologies

The tautology algorithm depicted in Figure 6.11 removes the vertices corresponding to clauses, which have internal links with empty test and send substitutions. All links connected to the removed vertices are also deleted. Obviously, the deletion of tautology clauses can be performed in a single pass process, since further tautologies cannot arise.

The second optimization considers so-called pure literals. Pure literals have neither incoming nor outgoing links, i.e. the corresponding clauses will never participate the reasoning phase. Consequently, the vertices and links of these clauses can also be removed from the clause graph.

```
algorithm delete_pure(clause graph $CG_S$)
 begin
    ⟨ Let $CG_S = (V, E)$ be a clause graph ⟩
    while $v \in V : \neg\exists(v, (\tau, \sigma), w) \in E \wedge \neg\exists(w, (\tau, \sigma), v) \in E$ do
          $E = E \setminus \{e | e = (v_1, (\tau, \sigma), v_2) \in E \wedge (v_1 \in \mathsf{clause}(v) \vee v_2 \in \mathsf{clause}(v))\}$
          $V = V \setminus \mathsf{clause}(v)$
 end
```

Figure 6.12: Delete clauses with pure literals

The purity algorithm depicted in Figure 6.12 implements the deletion of vertices and links according to the previous definition. In contrast to the deletion of tautologies, the purity algorithm has to be performed repeatedly after removing links from the clause graph, since the deletion of links possibly yields new pure literals. Therefore, removing clauses with pure literals is a recursive process. Each time a clause is deleted the whole clause graph has to be scanned again for pure literals.

In sum, the preprocessing computes a transformation of the initial set of clauses into an optimized clause graph. The final state of the clause graph contains all information which is required to begin the refutation process. Note that the preprocessing may also yield an empty clause graph. Usually, the set of query clauses has not been properly chosen by the user in this case. In the next section we discuss the reasoning phase of Purr.

6.2.3 Reasoning Phase

The reasoning phase in Purr is based on the distribution of newly inferred unit clauses among concurrent processes. Each process works on a certain part of the clause graph depending on the degree of granularity. In Purr fine granularity with OR-parallelism is obtained if each link in the clause graph is processed in parallel. Coarser granularity means that each process works on all outgoing links of one literal, for example. In this section we focus on these issues in detail.

6.2.4 A Sequential Approach

Before we discuss more technical aspects of the reasoning phase, we present a solely sequential version of Purr's reasoning phase and point out where parallelism might be obtained.

```
 1  algorithm sequential_reasoning_phase(clause graph $CG_S$)
 2   begin
 3      ⟨ Let $CG_S = (V, E)$ be a clause graph ⟩
 4      ⟨ Initialize the sets REC, MRG, RES, and SNT of $CG_S$ ⟩
 5      loop
 6          forall $C \in \{\mathsf{clause}(v) | v \in V$ and all literals of $\mathsf{clause}(v)$ have incoming links$\}$ do
 7              if TERMINATE applied on $C$ yields non-empty set $\Sigma$ then
 8                  return $\Sigma$
 9          forall $v \in V : \mathsf{clause}(v) = \{u, v_1, \ldots, v_m, v\}$ with $m \geq 0$ do
10              forall $e = (v, (\tau, \sigma), w) \in E$ do
11                  Apply RESOLVE to $e$
12   end
```

Figure 6.13: Sequential Reasoning in Purr

Consider the algorithm depicted in Figure 6.13. The initialization procedure in line 4 corresponds to the creation of the initial state of the sets *REC*, *MRG*, *RES*, and *SNT* introduced in Chapter 3. The following loops provide several ways of parallelization. The first loop in line 6 calls the rule TERMINATE on all clauses which have incoming links on all literals. Obviously, the rule TERMINATE can be processed in parallel, since every process implementing the rule TERMINATE works on its own clause. One clause will never be processed by several different processes for termination. Thus processes performing the rule TERMINATE are per definition completely data-independent among themselves.

The parallelization of the loops in the lines 9 and 10 introduces data dependence in the following way: Consider a clause $C = \{L_1, L_2, L_3\}$ with three literals. Moreover, let literal L_1 and literal L_2 have several incoming links and let literal L_3 have two outgoing links. Therefore, we could employ two concurrent processes performing the rule RESOLVE on the two outgoing links. Obviously, the two processes depend on the same substitution sets in *REC* and in *MRG* of the literals L_1 and L_2. In order to obtain data independence, each process has to maintain its own substitution sets. Data dependence among processes for RESOLVE exists as long as there are several concurrent processes working on the same clause.

Another interdependence occurs if literal L_3 also has incoming links. In this case a process performing the rule TERMINATE is applied to clause C. Thus this process has to maintain its own substitution sets, accordingly. In general, data dependence among processes for RESOLVE and TERMINATE exists if clauses have incoming links on all literals and outgoing links on some literals.

Note that there remains another type of data dependence among the processes. This dependence corresponds to the links of the clause graph and reflects the fact that substitutions have to be exchanged between the substitution sets attached to the clause graph. The degree of parallelism is reduced if there are directly or indirectly connected processes with varying load such that one process has to wait for the output of another process. The waiting process remains idle and thus does not support the reasoning phase. In fact, this data dependence is the only remaining dependence in our approach and thus it is responsible for non-linear speedup. For example, the behavior of processes performing the rule TERMINATE on unit clauses corresponds to this phenomenon. Such a process is going to receive at most one substitution which obviously corresponds to a proof situation. Most of the time before the proof is found, this process remains idle and thus does not support parallelism. We provide a solution for this example later.

Again, consider the loops in line 9 and 10. The parallelization of both loops yield the finest possible granularity with OR-parallelism in this approach. Each link in the clause graph is processed in parallel with the rule RESOLVE. Further parallelization of the rule RESOLVE would enter the field of AND-parallelism. On the other hand, if we solely parallelize the loop in line 9, we obtain coarser granularity. Then, one process performs the rule RESOLVE on all outgoing links of one literal. In PURR, we implemented these two levels of parallelism.

Of course, there are more levels of parallelism. For example, one process could work on several more or all outgoing links of one clause or even more clauses, yielding coarser granularity. In practice, coarser granularity has to be considered when working with large sets of clauses in order to reduce the number of concurrent processes.

In sum, the transition rules RESOLVE and TERMINATE can be implemented on different levels of parallelism. Therefore, our approach provides different granularity levels. The finer the granularity, the more copies of substitution sets in the clause graph have to be maintained in different processes. Consequently, since the substitution sets are generally data-dependent, more communication is required to maintain data integrity. In practice, if more than one process works on the same clause, the send operation becomes a multicast operation.

6.2.5 The Parallel Implementation

This section describes the implementation of the transition rules of Chapter 3. The application of transition rules on the substitution sets attached to a clause graph is implemented by concurrent slave processes. One slave process implements the repeated application of a transition rule on certain substitution sets in the sets *REC*, *MRG*, *RES*, and *SNT*.

A slave process which implements the transition rule RESOLVE is called a *resolution process*. A resolution process might work on a single link or a collection of links. There is also a second type of slave process called the *terminator process* which implements the transition rules TERMINATE and TRYTERMINATE. A terminator process is required for each clause with incoming links on all literals in order to provide complete detection of proof situations. If a terminator process finds a simultaneous unifier, a proof message is immediately sent to the master process which in turn stops all slave processes.

Resolution Process on a Single Link. Consider the algorithm for a resolution process on a single link depicted in Figure 6.14. The algorithm mainly reflects the transition rule RESOLVE which we introduced in Chapter 3. We assume that the source vertex v of the processed link e belongs to a non-unit clause C. In line 5, any vertex u of $C \setminus \{v\}$ is selected to receive new substitutions Σ_u. Note that the set Σ_u of newly received substitutions corresponds to the substitution sets in *REC*. In practice, Σ_u can always be deleted after the loop has been finished. The sets Θ correspond to the substitution sets in *MRG* and have to be maintained seperately

```
1  algorithm resolution_process_link(link $e$)
2   begin
3      ⟨ Let $e = (v, (\tau, \sigma), w)$ be a link ⟩
4      while proof not found do
5          ⟨ Let $u \in V$ : $\mathsf{clause}(v) = \{u, v_1, \ldots, v_m, v\}$ be the current receiver literal ⟩
6          ⟨ Let $\Theta_u, \Theta_{v_1}, \ldots, \Theta_{v_m}$ be already merged indexes for $u, v_1, \ldots, v_m, v$ ⟩
7          receive $\Sigma_u$ for $u$
8          $(\Sigma'_u, \Theta'_u) = \mathtt{subsume}(\Sigma_u, \Theta_u)$
9          $\Theta_u = \mathtt{union}(\Sigma'_u, \Theta'_u)$
10         $\Gamma = \mathtt{multi\text{-}merge}(\Sigma'_u, \Theta_{v_1}, \ldots, \Theta_{v_m})$
11         ⟨ Let $\Gamma_{v,w}$ be an index of all resolved substitutions for link $e$ ⟩
12         $\Gamma_{v,w} = \mathtt{union}(\mathtt{subsume}(\Gamma, \Gamma_{v,w}))$
13         $\Sigma = \mathtt{link\_mgu}(\mathtt{select}(\Gamma_{v,w}), \tau, \sigma)$
14         send $\Sigma$ to $w$
15  end
```

Figure 6.14: UR-Resolution on Link $(v, (\tau, \sigma), w)$

in each resolution process. For example, the set Θ_u contains all previously received and merged substitutions of vertex u.

In line 10, the multi-merge operation computes the simultaneous unifiers Γ among the subsumed rest Σ'_u of newly received substitutions and all substitutions previously received for the remaining literals of C. The result Γ is tested for subsumption and inserted into the set $\Gamma_{v,w}$ of resolved substitutions in *RES*. Therefore, $\Gamma_{v,w}$ is also required to be maintained in each resolution process. This set contains all previously computed simultaneous unifiers.

The next step is the selection of 'best' substitutions and the following link unification. Recall that we used the substitution sets in *SNT* to identify previously selected substitutions. Like the sets in *REC*, the sets in *SNT* are not explicitly required since previously selected substitutions are identified in $\Gamma_{v,w}$ by a special marking mechanism. Finally, the newly selected substitutions are tested for unifiability with the test substitution. The successful unifiers are applied to the send substitution and the resulting instantiations are collected in Σ. The resulting set Σ is sent to all processes working on vertex w as a receiver literal. Note that the send operation actually is a multicast operation, because in general several processes may consider vertex w as a receiver literal.

Figure 6.15 shows an optimized version of a resolution process for single links. The optimization concerns the reduction of possible simultaneous unifiers and the avoidence of intermediate data. Obviously, we obtain less simultaneous unifiers if the test substitution for the link is considered earlier as before since not all simultaneous unifiers are also unifiable with the test substitution. Actually, the link unification is an integral part of an optimized multi-merge operation. Consequently, we also integrated the instantiation of the send substitution into the multi-merge. Therefore, the result $\Gamma_{v,w}$ contains instantiated send substitutions which belong to the literal of w instead of substitutions which belong to v. Line 12 shows that we merely have to select substitutions Σ. The test unification and the send instantiation have been performed in the multi-merge.

In order to avoid large intermediate data, we also added a simultaneous subsumption test to the multi-merge. Every new simultaneous unifier is inserted into $\Gamma_{v,w}$ after performing $n:1$ forward and backward subsumption with $\Gamma_{v,w}$. Note that the simultaneous subsumption test

```
algorithm resolution_process_link_opt(link $e$)
 begin
    ⟨ Let $e = (v, (\tau, \sigma), w)$ be a link ⟩
    while proof not found do
       ⟨ Let $u \in V$ : clause$(v) = \{u, v_1, \ldots, v_m, v\}$ be the current receiver literal ⟩
       ⟨ Let $\Theta_u, \Theta_{v_1}, \ldots, \Theta_{v_m}$ be already merged indexes for $u, v_1, \ldots, v_m, v$ ⟩
       receive $\Sigma_u$ for $u$
       $(\Sigma'_u, \Theta'_u)$ = subsume$(\Sigma_u, \Theta_u)$
       $\Theta''_u$ = union$(\Sigma'_u, \Theta'_u)$
       ⟨ Let $\Gamma_{v,w}$ be an index of all resolved substitutions for link $e$ ⟩
       $\Gamma_{v,w}$ = union(subsume(link_mgu(multi-merge$(\Sigma'_u, \Theta_{v_1}, \ldots, \Theta_{v_m}), \tau, \sigma), \Gamma_{v,w}$))
       $\Sigma$ = select$(\Gamma_{v,w})$
       send $\Sigma$ to $w$
 end
```

Figure 6.15: Optimized UR-Resolution on Link $(v, (\tau, \sigma), w)$

also provides subsumption among new simultaneous unifiers. In fact, this test is one of the most important reduction methods in PURR.

Resolution Process on Some Links. In the following we discuss the algorithm for a resolution process working on all outgoing links of one literal together. Recall that this type of resolution process yields coarser granularity, i.e. the number of required resolution processes is reduced while the amount of computation per resolution process is increased.

```
algorithm resolution_process_literal_opt(vertex $v$)
 begin
    while proof not found do
       ⟨ Let $u \in V$ : clause$(v) = \{u, v_1, \ldots, v_m, v\}$ be the current receiver literal ⟩
       ⟨ Let $\Theta_u, \Theta_{v_1}, \ldots, \Theta_{v_m}$ be already merged indexes for $u, v_1, \ldots, v_m, v$ ⟩
       receive $\Sigma_u$ for $u$
       $(\Sigma'_u, \Theta'_u)$ = subsume$(\Sigma_u, \Theta_u)$
       $\Theta_u$ = union$(\Sigma'_u, \Theta'_u)$
       ⟨ Let $\Gamma_{v,\{w_1,\ldots,w_n\}}$ be an index of all resolved substitutions for all outgoing links of $v$ ⟩
       $\Gamma_{v,\{w_1,\ldots,w_n\}}$ = union(subsume(multi-merge$(\Sigma'_u, \Theta_{v_1}, \ldots, \Theta_{v_m}), \Gamma_{v,\{w_1,\ldots,w_n\}}$))
       $\Delta$ = select$(\Gamma_{v,\{w_1,\ldots,w_n\}})$
       forall $e = (v, (\tau, \sigma), w) \in E$ do
          $\Sigma$ = link_mgu$(\Delta, \tau, \sigma)$
          send $\Sigma$ to $w$
 end
```

Figure 6.16: Optimized UR-Resolution on Literal Associated with v

Figure 6.16 depicts an already optimized algorithm. Obviously, the multi-merge operation has to be performed only once for all links. In order to obtain a common representation of simultaneous unifiers for all links, we introduce a set $\Gamma_{v,\{w_1,\ldots,w_n\}}$ which contains the simultaneous unifiers before the link unification is performed. In other words, the substitutions in $\Gamma_{v,\{w_1,\ldots,w_n\}}$ belong to the literal of v. This approach provides more potential subsumption on simultaneous unifiers than the representation of unifiers seperated into sets Γ_{v,w_i} for each outgoing link. In the next step, substitutions Δ are selected out of $\Gamma_{v,\{w_1,\dots,w_n\}}$ for the following send operation. Finally, the instantiated send substitutions Σ have to be computed seperately for each link.

Note also that this approach shows a certain degree of unfairness. Consider a resolution process working on a literal with outgoing links a and b. The test substitutions of the two links are denoted by τ_a and τ_b. We assume that the test substitution τ_a provides a significantly higher probability for a successful link unification with simultaneous unifiers than τ_b. For example, τ_b only contains ground terms while τ_a only contains variables in its codomain. The select operation in line 11 does not consider this difference. For instance, ten best entries could have been selected for the send operation, but only one entry is unifiable with both test substitutions while the other 9 entries are unifiable with τ_a only. In other words, in our example link a usually allows more substitutions per send operation than link b depending on the probability of successful link unifications.

The current implementation of Purr does not consider this unfairness. If we hold on the approach of a common set $\Gamma_{v,\{w_1,\dots,w_n\}}$, one problem in a solution is to provide the information about which entries have been delivered on which links. Another approach is to provide separated sets Γ_{v,w_i} for the simultaneous unifiers. In the worst case, this approach yield sets Γ_{v,w_i} which are more or less copies of each other.

```
algorithm terminator_process_clause(vertices $C$)
 begin
    while proof not found do
       ⟨ Let $u \in C = \{u, v_1, \dots, v_m\}$ be the current receiver literal ⟩
       ⟨ Let $\Theta_u, \Theta_{v_1}, \dots, \Theta_{v_m}$ be already merged indexes for $u, v_1, \dots, v_m$ ⟩
       receive $\Sigma_u$ for $u$
       $(\Sigma'_u, \Theta'_u)$ = subsume($\Sigma_u, \Theta_u$)
       $\Theta_u$ = union($\Sigma'_u, \Theta'_u$)
       $\Gamma$ = multi-merge($\Sigma'_u, \Theta_{v_1}, \dots, \Theta_{v_m}$)
       if $\Gamma \neq \emptyset$ then
          send $\Gamma$ to Master Process
 end
```

Figure 6.17: Termination on Clause C

Terminator Process on a Clause. Figure 6.17 shows an algorithm for a terminator process working on a clause C. A terminator process also performs input subsumption on the received substitutions, see the transition rule TRYTERMINATE in Chapter 3. The following multi-merge operation searches simultaneous unifiers which provide a successful instantiation of clause C to the empty clause. A proof is obtained if at least one simultaneous unifier could be computed. The answer is immediately sent to the master process which in turn invokes the postprocessing.

Terminator Process on Unit Resolvents. The termination on clauses with many literals usually performs poor, since the multi-merge operation is difficult and many concurrent resolution processes produce a large amount of information which cannot be handled efficiently. Therefore, we introduce another termination approach referred to as *unit terminator process*. A unit terminator process implements the well-known unit conflict method. This approach also provides a solution to the problem of idle terminator processes working on unit clauses which we mentioned before.

```
1   algorithm terminator_process_unit(predicate $P$)
2    begin
3        while proof not found do
4            ⟨ Let $\Theta_{pos}$ and $\Theta_{neg}$ be already merged indexes ⟩
5            ⟨ $\Theta_{pos}$ contains substitutions belonging to positive units, $\Theta_{neg}$ respectively ⟩
6            receive $\Sigma$
7            if $\Sigma$ belongs to positive units then
8                $(\Sigma', \Theta'_{pos})$ = subsume($\Sigma, \Theta_{pos}$)
9                $\Theta_{pos}$ = union($\Sigma', \Theta'_{pos}$)
10               $\Gamma$ = binary_merge($\Sigma', \Theta_{neg}$)
11           else
12               $(\Sigma', \Theta'_{neg})$ = subsume($\Sigma, \Theta_{neg}$)
13               $\Theta_{neg}$ = union($\Sigma', \Theta'_{neg}$)
14               $\Gamma$ = binary_merge($\Sigma', \Theta_{pos}$)
15           if $\Gamma \neq \emptyset$ then
16               send $\Gamma$ to Master Process
17   end
```

Figure 6.18: Termination on Units with Predicate P

Again, the clause graph provides sufficient information in order to determine on which predicate symbols a unit conflict might be detected. Note that we cannot completely discard termination on clauses in general. Clauses which do not participate in the search, but have incoming links for all literals, are still required to be tested for termination. Nonetheless, termination processes on unit clauses and the according resolution processes for their incoming links are no more required. Instead, we use a unit terminator process for each predicate symbol which occurs in a literal with incoming and outgoing links in the clause graph. Note that for predicate symbols occurring in literals connected to unit clauses, a unit terminator process is also required.

The algorithm for a unit terminator process is depicted in Figure 6.18. The binary merge operation in line 9 and 14 computes the unit conflict between two sets containing the substitutions of complementary literals. Note that the received substitutions in a unit terminator process can be discarded, if one of the sets Θ_{pos} or Θ_{neg} does not receive any unit resolvents except for the initial instances of the unit clauses.

In sum, the unit terminator approach improves the empty clause detection by reducing redundancies in the original concept.

6.2.6 Initiation, Control, and Termination

The following section describes the technical aspects of slave process initiation, control, and termination performed by the master process. Recall that we continue to discuss the algorithm of the master process right after the preprocessing.

The individual steps of the master process are depicted in Figure 6.19. There are mainly three phases. The first stage is the startup of the required slave processes. The following stage is the control of the parallel system. Finally, the third stage performs the suspension of all slave processes.

```
algorithm reasoning_phase(clause graph $CG_S$)
 begin
     ⟨ Let $CG_S = (V, E)$ be a clause graph ⟩
     Count number of required slave processes in $CG_S$
     Spawn slave processes
     Initialize slave processes
     Start slave processes
     Collect intermediate output and wait for proof message $\Sigma$
     Suspend slave processes
     return $\Sigma$
 end
```

Figure 6.19: Start and Control of the Parallel System

The Number of Slave Processes. The number of slave processes is determined by the clause graph and, under restrictions, by the user's choice which part of the clause graph should be processed by a single slave. Since the clause graph directly depends on the problem type, the user's influence on the level of granularity is limited. The current implementation provides two levels. Either each slave process works on a single link, or each slave process works on all outgoing links of one literal yielding coarser granularity.

Slave Process Initiation. All slave processes are initiated at a time. PVM provides the required method and thus determines on which processor the slaves will run. An open problem is the specific distribution of slave processes on different processors in the parallel machine. A better matching of the 'expected' potential of a slave process and the processor's performance together with the properties of the network would certainly improve the overall performance and load balance of the system. Obviously, the estimation of both the process' load and the produced amount of communication is non-trivial.

In order to estimate the load of a process, we consider the multi-merge as the most expensive operation. The amount of computation during the multi-merge increases with the number and size of indexes to be merged and the number of shared domain variables in the indexes. The number of indexes and their common domain variables are fixed and directly problem-dependent. The number of entries in the indexes directly depends on the amount of received information, i.e. on the behavior of other slave processes and the network's capabilities. And thus the number of entries in the indexes is directly associated with the amount of communication and is also indirectly problem-dependent. Therefore, we are faced with a variety of interdependencies between cpu load, communication, and the properties of the problem and the hardware. We further discuss these issues in Chapter 7 with the help of some experiments.

In the following we work with the trivial approach, i.e. slave processes are randomly spawned over the parallel machine.

Slave Process Initialization. The master process has to initialize the slave processes after they have successfully been started on the parallel machine. First, optional settings are transmitted. Thereafter, the required clause graph information is sent. For a single slave process we have:

1. Routing information, i.e. process identifications of the master process and all processes which are directly connected to this process.

2. The variable domain of each literal of the processed clause.

3. The test and send substitutions of the processed links.

Note that exact information about the literal terms of the processed clauses is not required. This fact stresses the concept of PURR which models the reasoning phase solely based on indexes. The concept of literals and clauses is not needed anymore.

Starting Slave Processes. Recall that the initial state of the sets *REC*, *MRG*, *RES*, and *SNT* introduced in Chapter 3 represents the beginning of the reasoning phase. The initial state of the sets corresponds to the distribution of all send substitutions that come from links originating in unit clauses. The master process searches the clause graph for unit clauses with outgoing links and sends the appropriate send substitutions to the processes which work on the connected literals. The slave processes with direct connections to unit clauses will immediately start reasoning.

Waiting for Proof Messages. The remaining work of the master process merely consists of general control of the slave processes and the collection and printing of their output. The collection of slave process output in a distributed environment is a non-trivial problem. Especially, when output of asynchronous slave processes has to be listed in a specific order.

During the reasoning phase the user may ask the master process to dump detailed statistics of the slave processes. Therefore, the master process broadcasts a request for statistics to all slave processes and, thereafter, collects the output generated by the slave processes. A slave process suspends its work when such a request has been received. The desired output is generated and the process resumes computation. Sometimes the output of one slave process arrives in several packages at the master process. These output packages might be interrupted by output packages originating from other slave processes yielding an unreadable output. Therefore, the master process has to focus on the output of one slave process and may proceed only if the output is completed.

In this context, another problem arises when the master process has to wait for a slave process. For example, some slave processes usually spend a lot of time in the multi-merge operation. The result is an unacceptably long delay until the generation of slave output starts. In order to avoid repeatedly probing for request messages during intensive operations in slave processes, we employ the concept of *signals*. A signal causes a process to suspend the current work and to execute a so-called *signal handler*. When the signal handler is finished the process resumes the current work. Note that a signal handler might perform strictly restricted operations only, since the current state of the interrupted process is uncertain.

Consequently, the master process actually broadcasts signals instead of request messages to the slave processes. In turn, when a slave process receives a specific signal, its signal handler immediately sets a certain global variable which is also accessable in the normal process context. Therefore, we exchanged a time-consuming probe operation against a variable access. For example, the global variable may also be accessed during a multi-merge operation. When the variable is set, the multi-merge is suspended and the request can be served. This concept yields a better answer performance, even when some slave processes are under heavy load. On the other hand, signal handling might reduce the portability of the system.

In addition to the output collection, the master process also provides a shut down routine. Like the output collection, a controlled shut down of all slave processes is also implemented with signals.

Another task of the master process is to wait for proof messages sent by terminator processes. When the master process receives a proof message the reasoning phase is finished and the postprocessing begins.

Suspension. The next operation of the master process after receiving a proof message is to suspend all slave processes. The suspension is achieved by broadcasting a wait message or, as an optimized approach, by broadcasting signals to all slave processes. The optimized suspension mechanism does not only provide faster generation of a proof, but also improves precision of timing information, since the amount of unuseful computation is reduced. Recall that we use signals to immediately interrupt large scale operations like the multi-merge. After the slave processes received the suspension command they are ready to answer the requests of the master process.

6.2.7 Postprocessing

The postprocessing mainly performs the grabbing of a proof. It is a solely sequential process since all slave processes are forced to wait for requests of the master process one after another. During the postprocessing the resolution processes merely work as databases on the produced substitutions while the terminator processes wait for termination. The whole operation is controlled by the master process. Figure 6.20 shows the algorithm of the master process in detail.

```
 1  algorithm postprocessing(clause graph $CG_S$, set $\Sigma$)
 2  begin
 3      ⟨ Let $CG_S = (V, E)$ be a clause graph ⟩
 4      ⟨ Let $\Sigma$ be a set of answer substitutions ⟩
 5      forall $\sigma \in \Sigma$ do
 6          ⟨ Let $O$ be the origin list of $\sigma$ ⟩
 7          apply $\sigma$ to appropriate query clause and print
 8          collect_proof($\sigma, O$)
 9      Collect output and terminate slave processes
10  end
```

Figure 6.20: The Postprocessing

Proof Generation. In general, the proof message sent by a terminator process may contain several different answer substitutions each representing a different proof. Actually, a terminator process uses a modified multi-merge operation which only searches for the first existance of a simultaneous unifier. Possible subsequent simultaneous unifiers of the current multi-merge operation are not considered, but the multi-merge operation is immediately stopped. Nevertheless, we present the general algorithm of the proof collection. Thus in line 5 all answer substitutions are considered for the generation of a proof.

```
1  algorithm collect_proof(substitution $\sigma$, list $O$)
2   begin
3      ⟨ Let $O$ be a list of pairs $p$ ⟩
4      ⟨ Let $p = (pid, n)$ be a pair of a process id and a number ⟩
5      forall $p = (pid, n) \in O$ do
6         if $pid$ belongs to master process then
7            print appropriate unit clause
8         else
9            query process $pid$ for substitution $n$ and its origin list
10           ⟨ Let $\sigma_p$ be the substitution $n$ in process $pid$ ⟩
11           ⟨ Let $O_p$ be the origin list of $\sigma_p$ ⟩
12           collect_proof($\sigma_p, O_p$)
13           apply $\sigma_p$ to appropriate literal and print
14  end
```

Figure 6.21: Recursive Proof Generation

The algorithm `collect_proof` depicted in Figure 6.21 implements the collection of the involved substitutions of a proof in a distributed environment. The reconstruction of a proof requires a data structure which provides information about the origin of a substitution. Therefore, a substitution σ is associated with a pair (pid, n) of a process identification number pid and a unique number n within this process. In other words, the pair (pid, n) together with σ means that the process with id pid inferred σ as the nth produced substitution. The indexing conveniently associates an inserted substitution with any kind of information. Nearly every index in PURR contains substitutions associated with these pairs.

The index containing the result of a multi-merge operation is the only exception. Each substitution σ in this index is required to provide information about which substitutions contributed to the creation of σ. Therefore, in this case we extend a single pair to a list of pairs called the *origin list*. The length of the origin list corresponds to the number of literals of the processed clause. The first pair corresponds to the original pair, i.e. the first pair contains the pid and n of the process which performs the multi-merge operation. The rest of the origin list contains the pairs of the substitutions which contributed to the creation of the simultaneous unifier. Thus a request for (pid, n) is answered by the process pid by searching the index containing simultaneous unifiers for substitution n. The process answers with the found substitution and the rest of the associated origin list. Note that substitutions selected for a send operation are associated with the first pair of the origin list only.

The `collect_proof` procedure is a recursion which traverses the proof tree until unit clauses are reached. Obviously, a substitution associated with the pid of the master process corresponds to a unit clause. On the other hand, a substitution associated with the pid of a resolution process corresponds to a unit resolvent with a origin list. In line 9 the process pid is requested to answer with the substitution itself and its origin list. Then the origin list is recursively processed until unit clauses are reached.

In the implementation of PURR, the proof tree is not printed immediately, but stored in an appropriate data structure providing better output format. For example, repeated unit resolvents can easily be printed only once. We also added a verifyable output format. This format can be interpreted by a script program which in turn uses another theorem prover to verify the single steps of the proof.

Termination. The final output collection is implemented as discussed before. Each slave process produces statistical output including timing information. The timing information is seperately collected by the master process in order to compute the total cpu time consumption of all slave processes. Finally, all slave processes are terminated by a regular termination message instead of signals since the slave processes are currently idle.

6.2.8 Options in Purr

Finally, we discuss the most important options in Purr. We distingiush traditional parameters like the subsumption switch and options controlling the special indexing operations and the communication of Purr. Most of the following options are used in many resolution based theorem provers:

- The **forward/backward** subsumption can be seperately turned off. The default is on for both subsumption operations.

- A limit on the **weight** of the produced substitutions can be imposed. During the multi-merge generated substitutions exceeding the limit are discarded. The default is no limit. This limit can also be introduced automatically by an assessment mechanism presented in the theorem prover Otter [McC94]. Initially, a fixed amount of memory, say 12 MBytes of RAM, is available. When one third of this memory has been filled, a limit is imposed on the number of symbols in deduced clauses/substitutions. The limit – referred to as *max* – is selected by computing the maximum size of the formulas/substitutions contained in the smallest 5% of all *set of support* formulas or of all substitutions stored in the tree containing the simultaneous unifier in a resolution process. Every tenth iteration of the main loop a prospective new limit m is calculated in the same way. If $m < max$, the limit is reset to m. McCune arrived at the values $\frac{1}{3}$ and 5% by trial and error.

- A limit on the **weight** of the substitutions to be sent can be imposed. The selection of substitutions only considers substitutions lighter than this limit. If there are no appropriate candidates, the selection chooses only one lightest substitution in order to keep the system running. This technique prevents too heavy substitutions to be considered for subsequent resolution steps if no better candidates are available. Instead, the system restricts the use of heavy substitutions and tries to find lighter substitutions. Note that all substitutions including the heavy substitutions are tested for termination. The default is no limit.

- A limit on the proof **level** restricts the depth of the search space. Substitutions with a creation path deeper than the limit are discarded. The default is no limit.

- The **output** of given, kept, and received substitutions can be turned on. Note that the output of slave processes has to be redirected to the master process. An efficient solution is to send the output also as compact indexes of substitutions instead of ready formatted output. The generation of a readable format can also be performed by the master process. The default is no output.

- The output of time and space **statistics** of the slave processes can be requested. The default is no statistics.

Since Purr works with indexes of substitutions as the fundamental data structure in the system, we also developed options on the level of indexes and indexing operations:

- The number of **multi-merges/send** operation determines how many multi-merges have to be performed before a selection and send operation is initiated. The default is one multi-merge per send operation. Depending on the problem, more multi-merges per send operation prevents too heavy substitutions to be considered for subsequent resolution steps. Note that the effect is similar to the weight limit on substitutions that are to be sent.
- The number of **indexes/receive** operation corresponds to the length of the receive queue of indexes. Before a slave process starts the multi-merge it receives indexes for one literal in the nucleus until the queue of the literal is full. The queue is sorted according to the minimal weight of the entries in the indexes. The effect is that indexes with lighter entries are considered first. The default is one index per receive operation.
- The number of given **substitutions/send** operation determines the size of the sent indexes. The default is ten substitutions per send operation. The usual size of a sent index ranges between five and fifty substitutions.

6.3 Techniques

In this section we discuss two different low-level aspects of our implementation. Both methods crucially improve the performance of Purr. The first technique called *contexts* addresses the efficient maintenance of variable bindings. This method also improves the *renaming* of variables. In the second part we present an efficient transformation procedure of substitution trees into process-independent form. The transformation provides the communication protocol with indexes among distributed processes.

6.3.1 Contexts

We present a technique called *contexts* which has been introduced by McCune for the theorem prover Otter [McC94]. Contexts allow the use of the same variables in actually variable disjoint terms or substitutions. Even substitution trees containing the same variables can be considered as variable disjoint with contexts.

A context is a data structure containing variable bindings. To this end we associate every new variable with a unique natural number. With this number we access a context that actually is an array containing an arbitrary maximum of binding elements. A binding element contains a pointer to a term and the name of a context. Thus contexts are similar to the notion of a substitution. Note that the number of possible variables is limited to the size of the smallest context.

Consider the example depicted in Figure 6.22. We employ two contexts C_1 and C_2 for the unification of the terms $t_1 = f(x, g(y, z), z)$ and $t_2 = f(g(x, y), y, x)$. We assume that both terms t_1 and t_2 are variable disjoint. Using the contexts C_1 and C_2 the terms do not have to be renamed. Variables occurring in t_1 are bound in C_1 whereas variables occurring in t_2 are bound in C_2. An important technical aspect is that a context is accessed by the unique number of a variable. In our example the variable x corresponds to the natural number 1, variable y to 2,

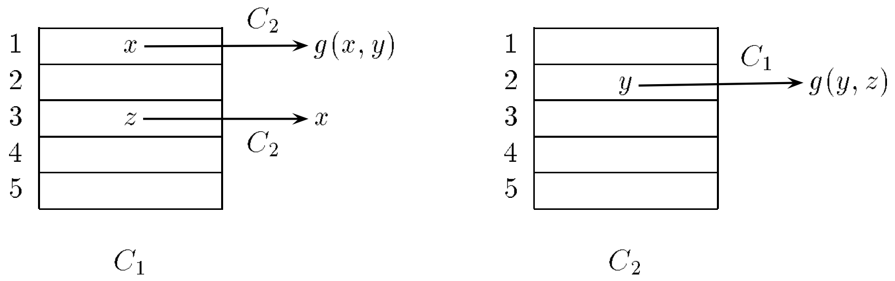


Figure 6.22: Two Contexts C_1 and C_2 with Bindings

and so on. This detail provides an efficient access of variable bindings that really improves the performance of the system.

In order to extract the common instance of the two terms, we simply apply the bindings in context C_1 to the term t_1 and rename the variables yielding the term $f(g(x, g(y, x)), g(y, x), x)$. Note that we also use contexts to rename variables efficiently. To this end we add an extra renaming slot to each binding element of the contexts. In Purr variable renaming always corresponds to normalization since substitution trees represent normalized terms or substitutions more effectively. We obtain a normalized term by renaming the variables in a specific order. Here, the order is determined by the unique number of every variable. Renaming always is started with variable x which corresponds to the natural number 1.

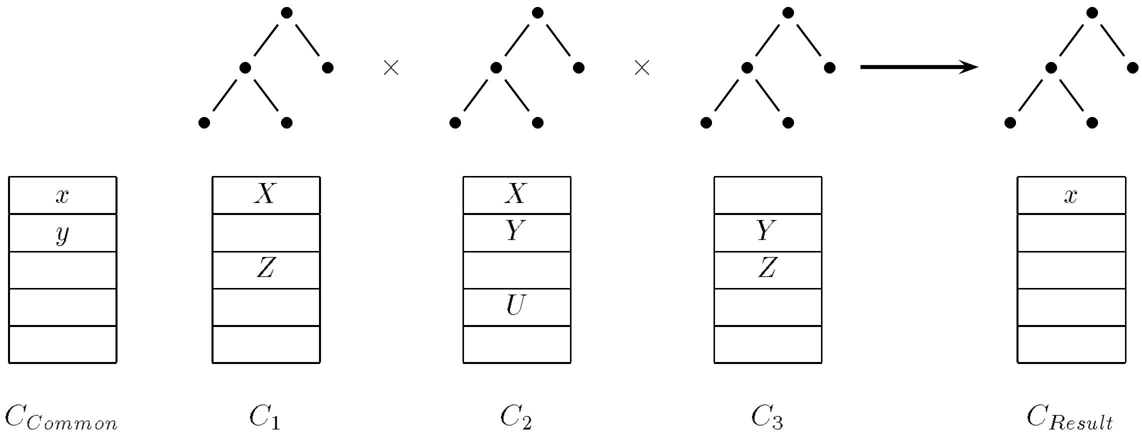


Figure 6.23: The Multi-Merge Operation with Three Substitution Trees

Multi-Merge. In Purr all operations involving the maintenance of variable bindings use contexts. In particular, the multi-merge operation can be supported by contexts very efficiently. Consider the example depicted in Figure 6.23. We employ five contexts in order to merge three substitution trees: The context C_{Common} maintains the *common* variables occurring in the nucleus. Common variables of the nucleus are denoted by lowercase characters x, y, and so on. Variables that belong to the electrons are referred to as *private* variables denoted by uppercase characters X, Y, and so on. The private variables occurring in the substitutions of the three trees as well as the index variables of the trees are maintained seperately in the contexts $C_1, \ldots, C_3$. Finally, context C_{Result} contains the variables of the resulting substitution tree. The integrated subsumption and insertion operations of the multi-merge use the context

C_{Result}. The other $n{:}m$ indexing operations use contexts accordingly.

Test Unification and Send Instantiation. The splitted unifier (τ, σ) of two connected literals containing a test substitution τ and a send substitution σ is also computed using contexts. The computation merely corresponds to the unification of literals followed by the extraction of substitutions. Here, we also distinguish variables occurring in the two different literals by using common and private variables. We will show that two variable types suffice to represent the required test and send substitutions as well as the substitutions being unifiers or ur-resolvents. Variables with indices "s" and "r" of the original approach in Chapter 3 are replaced.

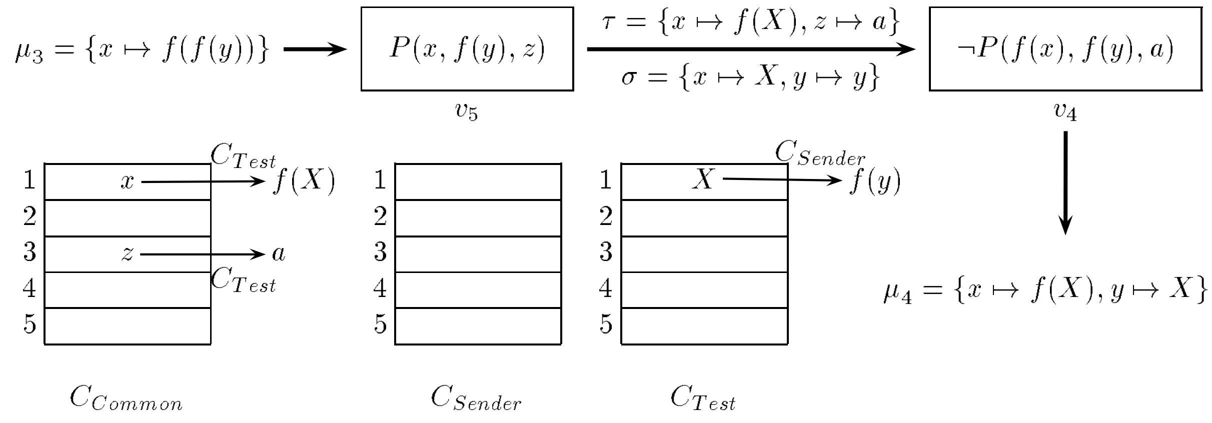

$\mu_3 = \{x \mapsto f(f(y))\}$
$P(x, f(y), z)$
$\tau = \{x \mapsto f(X), z \mapsto a\}$
$\sigma = \{x \mapsto X, y \mapsto y\}$
$\neg P(f(x), f(y), a)$
v_5
v_4
C_{Test}
1
2
3
4
5
x
$f(X)$
z
a
C_{Test}
C_{Sender}
X
$f(y)$
$\mu_4 = \{x \mapsto f(X), y \mapsto X\}$
C_{Common}
C_{Sender}
C_{Test}

Figure 6.24: Test Unification and Send Instantiation

Figure 6.24 depicts a fragment of an example which has been discussed in the last section of Chapter 3. The literals $P(x, f(y), z)$ and $\neg P(f(x), f(y), a)$ belong to the same clause and are connected by the internal link $(v_5, (\tau, \sigma), v_4)$. The common variables of the test substitution $\tau = \{x \mapsto f(X), z \mapsto a\}$ belong to the sender literal. The private variables like the variable X belong to the receiver literal.

Since only the codomain of the send substitution $\sigma = \{x \mapsto X, y \mapsto y\}$ is affected by the send instantiation, we consider the common variables of the codomain as being different from the variables in the domain. Therefore, the variable y in the codomain of the send substitution σ is different from the variable y in the domain which, however, implicitly means that the send substitution conforms to the definition of substitutions. The common variables in the codomain (like the variable y) belong to the same common variables in the test substitution τ while the private variables belong to the private variables in τ.

We consider the send operation of the ur-resolvent $\mu_3 = \{x \mapsto f(f(y))\}$ to the literal v_4 receiving the substitution $\mu_4 = \{x \mapsto f(X), y \mapsto X\}$. Actually, we unify the ur-resolvent $P(x, f(y), z)\mu_3$ with the receiver literal $\neg P(f(x), f(y), a)$ of a nucleus and extract the according unifier μ_4. This operation corresponds to the first send operation in the original example depicted in Figure 3.7.

For the test unification with the test substitution τ and the ur-resolvent μ_3 three contexts are required: The context C_{Common} maintains the common variables, i.e. the lowercase variables of τ and μ_3. The private variables of μ_3 are bound in the context C_{Sender} whereas the private variables of τ are bound in the context C_{Test}. The test unification first establishes the assignments $\{x \mapsto f(X)\}$ and $\{z \mapsto a\}$ of τ in the context C_{Common}. The context of the codomain terms is the context C_{Test}. Then μ_3 successfully is unified with respect to these bindings yielding the binding $\{X \mapsto f(y)\}$ with C_{Sender} being the context of $f(y)$.

The following send instantiation applies the bindings of the context C_{Test} to the codomain of the send substitution σ yielding the substitution $\mu = \{x \mapsto f(y), y \mapsto y\}$. Depending on context and type, the variables in this substitution have to be renamed or converted from common type to private type and vice versa. Unbound common variables like the variable y in μ are converted to private variables and then renamed yielding the substitution $\mu_4 = \{x \mapsto f(X), y \mapsto X\}$. The conversion is due to the interpretation that common variables like the variable y belong to the sender literal. Unbound private variables are converted depending on their context. Private variables of the test substitution simply are converted to common variables because these variables actually belong to variables of the receiver literal. Private variables of the ur-resolvents are not converted but renamed. Assignments of the send substitution that become assignments of the form $\{x \mapsto x\}$ are omitted.

In sum, contexts provide an efficient and flexible technique for the maintenance of variable bindings and for other operations involving variable handling like renaming and conversion.

6.3.2 Indexing and Process Communication

In the distributed theorem prover Purr indexes are used in the communication protocol. We now present a transformation of substitution trees and the underlying term structure to a process-independent representation. This representation can be transmitted between processes. After receiving the process-independent form, it is transformed back into the internal representation of substitution trees.

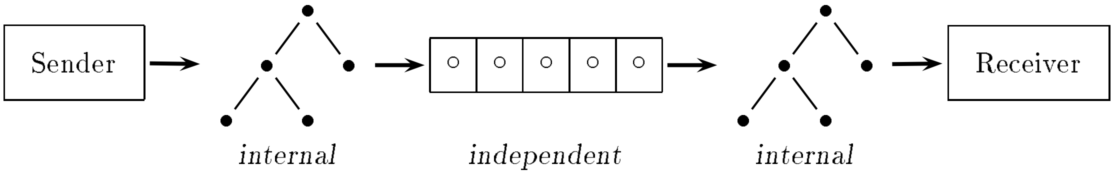


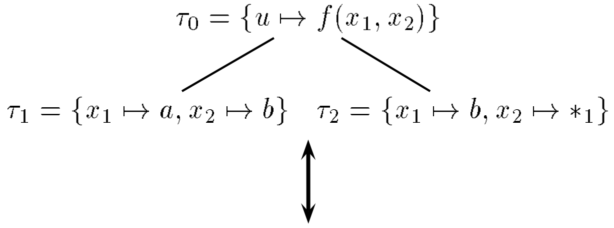
Figure 6.25: Process Communication

Early experiments with Purr showed that sending the new substitutions one by one produces too much communication overhead. In order to improve the communication performance of the system, the number of messages had to be decreased. As the prover produces substitution sets in each resolution step anyway, it is straightforward to exchange substitution sets among processes instead of single substitutions.

Process-independent Representation. A process-independent representation of substitution trees does not contain absolute addresses. Therefore, a procedure which yields an independent representation has to transform absolute into relative addresses. This transformation should enable us to reconstruct the substitution tree after transmission. The independent data structure has to be as simple as possible in order to send and receive a whole index at a time. Such a data structure is a vector of uniform elements.

6.3.3 Transformation of Substitution Trees

In PURR both substitution trees and terms are internally represented as recursive tree structures. Thus the transformation maps the tree representation of indexes and terms to a flat vector representation. The recursive structure can be represented in a vector by relative instead of absolute addresses. The vector itself is designed as a simple array of integers. We informally present the transformation procedure in a top-down manner.


$\tau_0 = \{u \mapsto f(x_1, x_2)\}$
$\tau_1 = \{x_1 \mapsto a, x_2 \mapsto b\}$
$\tau_2 = \{x_1 \mapsto b, x_2 \mapsto *_1\}$

I	$subst(\tau_0)$	2	L	$subst(\tau_1)$	1	*data*	L	$subst(\tau_2)$	1	*data*

Figure 6.26: Transformation of a Substitution Tree

Substitution Trees. We only consider standard substitution trees. The extention to other types of substitution trees is straightforward. An example of a transformation is depicted in Figure 6.26.

An inner node of a substitution tree consists of a substitution and a list of subtrees. Instead of subtrees, leaf nodes refer to a list of entries which correspond to inserted variants of the same substitution. First, a flag 'I' indicates that the following node is an inner node. A flag 'L' indicates a leaf node. Thereafter, the node's substitution τ is transformed to $subst(\tau)$. As the number of subnodes or entries is arbitrary, the number is inserted to the vector as a reconstruction reference. Finally, if the current node is an inner node the transformation recursively starts on the subnodes. Otherwise, a transformation procedure is called for all entries of the leaf node. In our example, both leaf nodes contain one entry indicated by 1 in the transformation.

This concept is extendable as long as there is a transformation for the indexed *data*. For example, if history information is added to each entry of an index the transformation of leaf nodes has to consider the transformation of history information accordingly.

$$\sigma = \{x \mapsto f(g(a), b), y \mapsto g(f(z, c))\} \longleftrightarrow \begin{array}{|c|c|c|c|c|}\hline 2 & x & term(x\sigma) & y & term(y\sigma) \\ \hline \end{array}$$

Figure 6.27: Transformation of a Substitution

Substitutions. A substitution σ consists of a finite set of variable-term pairs. See Figure 6.27 for an example. The internal representation of a substitution is a list of variable-term pairs. We transform a substitution into a vector of symbols such that the number of variable-term pairs is stored first. Then each pair is transformed with the transformed variable symbol followed by the transformed term.

$$t = f(g(a), b) \longleftrightarrow \begin{array}{|c|c|c|c|}\hline f & g & a & b \\ \hline\end{array}$$

Figure 6.28: Transformation of a Term

Terms. Consider the example in Figure 6.28. In PURR terms are represented in a conventional tree structure. A term consists of a top symbol and a pointer to an argument list containing terms. The string representation of a term corresponds to a process-independent representation. Thus the transformation maps each symbol in a term to the vector in a specific order, e.g. depth-first. A term is reconstructed according to the transformation order and the known, fixed arity of function symbols.

Symbols. A symbol is represented by an integer value. Variables are positive integers. Constant and function symbols are negative. The least significant bits of an integer are used to encode different types of variables (common or private) or the arity of the function symbols. Obviously, the integer representation of symbols is process-independent. Therefore, the transformation of symbols is trivial.

6.3.4 Assessment

In many experiments PURR showed best performance in the range of 10 to 100 entries per index which means that a vector of 10 to 50 kB is needed. The transformation to the vector representation yielded an average compression level of about one third compared to the original memory consumption of a substitution tree with conventional terms. The time needed to transform a tree in both directions is negligible compared to operations like the multi-merge or subsumption.

7

Experiments

In this chapter we show that our prototype implementation of PURR is able to achieve very high performance. To this end, we present a selection of experiments that focus on the effect of both indexing and parallelism on the systems overall performance.

The impact of indexing on PURR's performance is investigated by a first set of experiments, which use the well-known *Condensed Detachment* principle introduced by C. A. Meredith [LMM^{+}57]. Our experiments are based on problem sets that have been presented by William McCune and Larry Wos [MW92]. With respect to the experiments reported in that paper we will compare PURR and the theorem prover OTTER [McC94]

In order to investigate increasing degree of parallelism, we also performed experiments on problem sets that consist of more initial clauses than the condensed detachment examples, in order to investigate an increasing degree of parallelism. Experiments cover the well-known *Sam's Lemma* [MOW76b] and the *Steam-Roller* [Pel86] problems. The axiomatization of all problems were taken from the TPTP Problem Library [SSY94].

Experiments covering all problem sets of the TPTP library which are refutable by ur-resolution cannot be performed automatically because PURR's implementation is still unstable. The main problem is that the prover has been developed as a pure distributed system which is very difficult to debug. The current implementation cannot simulate concurrency using a single sequential process. At least errors related to the complex indexing operations could be found more easily in a sequential process. Nevertheless, our results have been verified automatically by a shell script using the theorem prover OTTER.

7.1 Indexing

Term indexing supports fast access to and maintenance of large databases in automated theorem provers. Therefore, the investigation of indexing methods within an automated theorem prover is supported by problem sets which require the prover to create and maintain a large knowledge base during the search process. The study of logic calculi with condensed detachment is recognized as a very challenging field for theorem provers. Some problems that arise in this context are extremely difficult, i.e. a huge number of inferences has to be drawn. Additionally, all problems in this area can be processed by a single process of PURR providing a fair basis for the comparison with sequential theorem provers like OTTER.

In the following we merely provide the first-order axiomatization of the condensed detachment inference rule. A detailed theoretical background of condensed detachment has been presented by J. Lukasiewicz [Luk70].

Definition 7.1.1 (Condensed Detachment)
For a binary function symbol i and a unary predicate symbol $P \in \mathbf{P}_1$ condensed detachment is defined as:

$$\forall x \forall y (P(i(x,y)) \wedge P(x) \Rightarrow P(y))$$

which corresponds to the clause normal form:

$$\{\neg P(i(x,y)), \neg P(x), P(y)\}$$

Condensed Detachment combines detachment (modus ponens) and instantiation for a binary operation i. This binary operation usually represents the implication or equivalence within a calculus. The unary predicate P applied to a subformula α is interpreted as "α is a *theorem*" or "α holds". Therefore, the condensed detachment inference rule can be used to derive new theorems or axioms within a certain calculus.

Our experiments with condensed detachment fit into a specific presentation scheme. In particular, the structure of the clause graph that corresponds to the experiment's axiomatization remains almost unchanged. Only axioms and theorems represented by unit clauses change. Therefore, the number of employed processes in these experiments is fixed. The degree of difficulty is solely determined by the axioms and the denied theorem.

Implicational Propositional Calculus. The problems are presented in the following way: Given the formulas

(IC-1)	$i(x,x)$	(IC-4)	$i(i(x,y), i(i(y,z), i(x,z)))$
(IC-2)	$i(x, i(y,x))$	(IC-5)	$i(x, i(i(x,y), y))$
(IC-3)	$i(i(i(x,y),x),x)$	(IC-JL)	$i(i(i(x,y),z), i(i(z,x), i(u,x)))$

each holding in IC. Each of the sets {IC-2,IC-3,IC-4} and {IC-JL} axiomatizes IC. The problems 63–68 depicted in Table 7.2 are to derive each system from the other. For example, problem 67, IC-JL⇒IC-4, is to find a refutation of the clauses:

$\{\neg P(i(x,y)), \neg P(x), P(y)\}$	Condensed Detachment
$\{P(i(i(i(x,y),z), i(i(z,x), i(u,x))))\}$	IC-JL
$\{\neg P(i(i(a,b), i(i(b,c), i(a,c))))\}$	Denial of IC-4 (skolemized)

The corresponding clause graph for the proof of IC-JL⇒IC-4 is depicted in Figure 7.1. The denied theorem IC-4 is marked with "?" and thus represents the only query clause. Axiom IC-JL is connected to the condensed detachment clause representing a possible inference. Obviously, the internal links $(v_4, (\tau_3, \sigma_3), v_3)$ and $(v_4, (\tau_4, \sigma_4), v_2)$ represent by far the highest potential concerning the computation of inferences. Link $(v_4, (\tau_5, \sigma_5), v_5)$ merely expects one substitution which corresponds to a successful refutation.

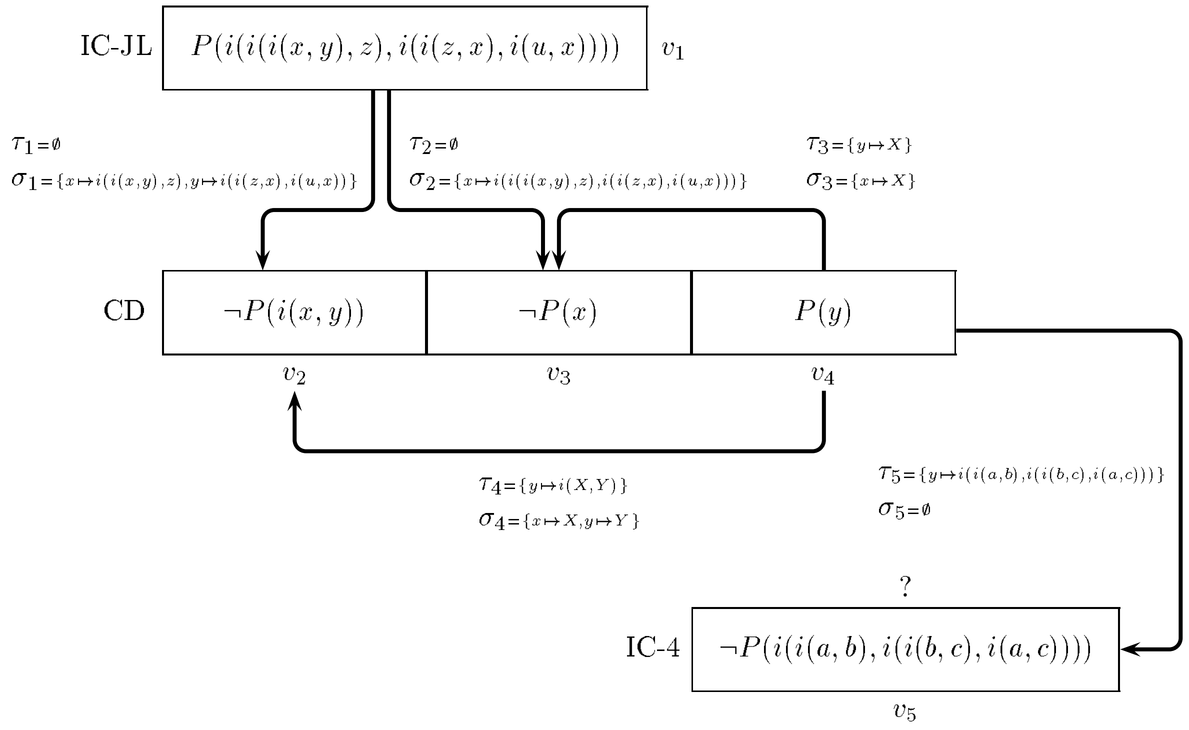

IC-JL
$P(i(i(i(x,y),z),i(i(z,x),i(u,x))))$
v_1
$\tau_1=\emptyset$
$\sigma_1=\{x\mapsto i(i(x,y),z),y\mapsto i(i(z,x),i(u,x))\}$
$\tau_2=\emptyset$
$\sigma_2=\{x\mapsto i(i(i(x,y),z),i(i(z,x),i(u,x)))\}$
$\tau_3=\{y\mapsto X\}$
$\sigma_3=\{x\mapsto X\}$
CD
$\neg P(i(x,y))$
$\neg P(x)$
$P(y)$
v_2
v_3
v_4
$\tau_4=\{y\mapsto i(X,Y)\}$
$\sigma_4=\{x\mapsto X,y\mapsto Y\}$
$\tau_5=\{y\mapsto i(i(a,b),i(i(b,c),i(a,c)))\}$
$\sigma_5=\emptyset$
?
IC-4
$\neg P(i(i(a,b),i(i(b,c),i(a,c))))$
v_5

Figure 7.1: Clause Graph of the Problem IC-JL⇒IC-4

Recall that our approach provides various degrees of granularity. In fact, PURR can work on the clause graph either with a single resolution process for all outgoing links of a literal or with resolution processes for each particular link. Experiments with resolution processes for all outgoing links of a literal are denoted by PURR$_{Literal}$-n. The number n denotes the number of available processors. Experiments providing resolution processes for each link are denoted by PURR$_{Link}$-n.

In practice, two slave processes are employed using PURR$_{Literal}$-n on the condensed detachment examples. One slave process is the resolution process, the other process is a unit terminator process for the predicate P. The current implementation of PURR$_{Link}$-n does not support the unit termination concept: PURR$_{Link}$-n yields four slave processes for the condensed detachment experiments. Three resolution processes work on the three outgoing links of v_4 and one clause terminator observes v_5. Note that in the current example the unit termination concept could reduce the number of slave processes to three since the resolution process for link 5 is actually not required. In other words, the resolution process on link 5 and the clause terminator process on v_5 could be combined to one unit terminator process on predicate P.

Nonetheless, the four slave processes in the PURR$_{Link}$-n approach only provide a degree of parallelism of two since the two resolution processes working on internal links consume by far the most cpu time. We observe a similar behavior of PURR$_{Literal}$-n which provides almost no parallelism since here the unique resolution process is dominant.

Test Conditions. In the following tables “fail” indicates that no proof has been found within four hours with a maximum of 12 Mb for all processes. The reported times are the seconds needed to find a refutation on a SuperSparc 10 computer with two processors. In order to obtain experiments with $n = 1$ on the two processor machine we estimate the elapsed real time by summing up the cpu ticks needed in all processes involved. The times for experiments with $n = 2$ are real times. Note that this timing method does not provide exact results. But it suffices to compare the different configurations of Purr. Otter uses hyperresolution without backward subsumption. Purr employs the restriction of unit resulting resolution to positive unit clauses, which in this case is equivalent to hyperresolution, as can be seen in Figure 7.1. Purr uses a limit of 10 substitutions per message. Backward subsumption is also switched off.

#	Theorem	Otter	Purr$_{Literal}$-1	Purr$_{Link}$-1	Purr$_{Link}$-2
63	IC-2,IC-3,IC-4 $\Rightarrow$ IC-JL	19	$\boxed{\mathbf{32}}$	74	52
64	IC-JL $\Rightarrow$ IC-1	3	$\boxed{\mathbf{1}}$	$\boxed{\mathbf{1}}$	$\boxed{\mathbf{1}}$
65	IC-JL $\Rightarrow$ IC-2	3	$\boxed{\mathbf{1}}$	$\boxed{\mathbf{1}}$	$\boxed{\mathbf{1}}$
66	IC-JL $\Rightarrow$ IC-3	12	$\boxed{\mathbf{2}}$	7	4
67	IC-JL $\Rightarrow$ IC-4	3135	$\boxed{\mathbf{1324}}$	2039	1373
68	IC-JL $\Rightarrow$ IC-5	857	$\boxed{\mathbf{213}}$	320	239

Figure 7.2: Experiments with the Implicational Propositional Calculus

Consider the column for Purr$_{Literal}$-1 in Figure 7.2. Purr employs one resolution process and one unit terminator process in this configuration. In these experiments the resolution process consumes almost all cpu-seconds during the run. The most time-consuming operation within the resolution process is the multi-merge. The unit terminator merely has to perform a n:1 indexing operation for unifiable partners of the denied theorem in the set of produced inferences. The costs of this test are small compared to the multi-merge operation. Thus Purr runs sequentially in this configuration providing a basis for the comparison to the sequential Otter system. Although Purr produces communication overhead, it shows better performance.

The two columns for Purr$_{Link}$-1 and Purr$_{Link}$-2 contain the proof times for Purr working with two resolution processes on condensed detachment. Obviously, a maximum speedup of two might be obtained in this configuration. The times of Purr$_{Link}$-1 compared to Purr$_{Literal}$-1 are nearly doubled since the multi-merge operation on condensed detachment is employed twice in the Purr$_{Link}$-1 configuration.

The Purr$_{Link}$-2 configuration again achieves similar results as the sequential variant of Purr$_{Literal}$-1. The proof times of Purr$_{Link}$-2 corresponds to a speedup of about 1.5 compared to the proof time of Purr$_{Link}$-1.

Detailed Statistics. For a more detailed analysis consider the time and space statistics of a single resolution process depicted in Figure 7.3. These statistical data have been obtained by trying to prove problem 67, i.e. IC-JL$\Rightarrow$IC-4, with Purr$_{Literal}$-1. Recall that in the Purr$_{Literal}$-1 setting a single resolution process works on both internal links $(v_4, (\tau_4, \sigma_4), v_2)$ and $(v_4, (\tau_3, \sigma_3), v_3)$. Statistical data about the according unit terminator process is omitted due to its minor contribution to time and space consumption. Actually, the unit terminator process

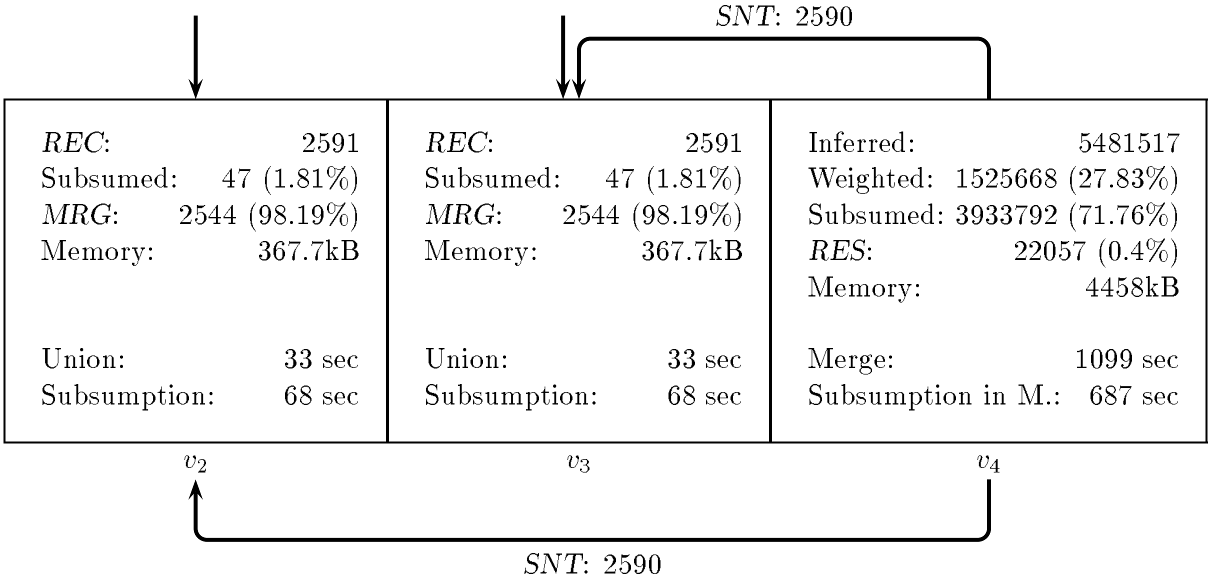


Figure 7.3: Statistics of the Resolution Process in Purr$_{Literal}$-1 on Problem IC-JL⇒IC-4

consumes only a few seconds of cpu-time. Note that this example is a good representative for the behavior of Purr on condensed detachment.

The presentation of statistical data resembles the original clause graph in Figure 7.1. The nodes v_2 and v_3 contain information on the input behavior of the resolution process. Both nodes produce identical data due to the properties of condensed detachment. The process received 2591 substitutions on each node and performed forward subsumption in 68 seconds with almost no effect. The remaining 2544 substitutions were inserted into a 367.7 kilobytes substitution tree in 33 seconds.

Node v_4 provides information about the generation of new inferences performed on the received substitutions. About 5.5 million new clauses were inferred. The automatic weight control rejected 27% of these clauses. Forward subsumption avoided the maintenance of 71% of the new inferences. Note that forward subsumption represents the most expendable part of the multi-merge (687 seconds for forward subsumption of 1099 seconds for the whole computation of new inferences). Only 0.4% of all new inferences were kept and inserted into a 4.4 megabytes substitution tree. The strategy selected 2590 inferences for both outgoing links for subsequent resolution steps. Note that all 22057 saved inferences were sent to the unit terminator process for testing unit conflicts.

We observe that the multi-merge operation consumes by far most of the cpu-time and its result also consumes most of the memory. The forward subsumption within the multi-merge takes about 62.5% of the overall merge time. The impact of subsumption on the amount of kept inferences is very strong. Due to this observation the subsumption test has been integrated into the multi-merge operation. New inferences are kept in memory only if they passed the weighing and subsumption tests.

With an increasing size of the kept inferences the subsumption test becomes more time-consuming. Nevertheless, experiments with very large substitution trees (several hundreds of megabytes) still show reasonable inference rates.

Equivalential Calculus. The formulas in the left column hold in the equivalential calculus (EC). Each of the formulas in the right column is a single axiom for the equivalential calculus.

(EC-1)	$e(e(e(x,y),e(z,x)),e(y,z))$
(EC-2)	$e(e(x,e(y,z)),e(e(x,y),z))$
(EC-4)	$e(e(x,y),e(y,x))$
(EC-5)	$e(e(e(x,y),z),e(x,e(y,z)))$

(YQL)	$e(e(x,y),e(e(z,y),e(x,z)))$
(YQF)	$e(e(x,y),e(e(x,z),e(z,y)))$
(YQJ)	$e(e(x,y),e(e(z,x),e(y,z)))$
(UM)	$e(e(e(x,y),z),e(y,e(z,x)))$
(XGF)	$e(x,e(e(y,e(x,z)),e(z,y)))$
(WN)	$e(e(x,e(y,z)),e(z,e(x,y)))$
(YRM)	$e(e(x,y),e(z,e(e(y,z),x)))$
(YRO)	$e(e(x,y),e(z,e(e(z,y),x)))$
(PYO)	$e(e(e(x,e(y,z)),z),e(y,x))$
(PYM)	$e(e(e(x,e(y,z)),y),e(z,x))$
(XGK)	$e(x,e(e(y,e(z,x)),e(z,y)))$
(XHK)	$e(x,e(e(y,z),e(e(x,z),y)))$
(XHN)	$e(x,e(e(y,z),e(e(z,x),y)))$

Problems 71-84 depicted in Figure 7.4 are to start with each single axiom and to derive the system that precedes it in our listing. The results are similar to the experiments obtained with the implicational calculus. Additionally, PURR finds a refutation for problem 82 where OTTER fails. Since PURR selects the first ten substitutions as given whereas OTTER always chooses only a single clause as given at a time, PURR shows a different search behavior. It considers more clauses than only a single lightest clause for subsequent resolution steps.

#	Theorem	OTTER	PURR$_{Literal}$-1	PURR$_{Link}$-1	PURR$_{Link}$-2
69	EC-4,EC-5 ⇒ EC-1	93.3	$\boxed{\mathbf{18.4}}$	31.0	18.6
70	EC-4,EC-5 ⇒ EC-2	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$
71	YQL ⇒ EC-4	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$
72	YQL ⇒ EC-5	7.6	$\boxed{\mathbf{2.5}}$	4.1	3.7
73	YQF ⇒ YQL	$\boxed{<\mathbf{1}}$	2.5	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$
74	YQJ ⇒ YQF	12.0	6.7	8.0	$\boxed{\mathbf{5.3}}$
75	UM ⇒ YQJ	190.0	$\boxed{\mathbf{67.7}}$	119.3	72.0
76	XGF ⇒ UM	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$
77	WN ⇒ XGF	38.6	28.2	49.2	$\boxed{\mathbf{27.2}}$
78	YRM ⇒ WN	120.4	41.0	58.0	$\boxed{\mathbf{30.2}}$
79	YRO ⇒ YRM	69.4	36.1	50.9	$\boxed{\mathbf{27.9}}$
80	PYO ⇒ YRO	108.9	56.0	93.7	$\boxed{\mathbf{54.5}}$
81	PYM ⇒ PYO	88.0	18.0	29.1	$\boxed{\mathbf{16.3}}$
82	XGK ⇒ PYM	fail	$\boxed{\mathbf{63.6}}$	111.8	69.3
83	XHK ⇒ XGK	fail	fail	fail	fail
84	XHN ⇒ XHK	242.5	115.2	78.0	$\boxed{\mathbf{38.4}}$

Figure 7.4: Experiments with the Equivalential Calculus

R Calculus. Each of the following formulas is a single axiom for the R calculus:

(QYF) $e(e(e(x,y),e(x,z)),e(z,y))$
(YQM) $e(e(x,y),e(e(z,y),e(z,x)))$
(WO) $e(e(x,e(y,z)),e(z,e(y,x)))$
(XGJ) $e(x,e(e(y,e(z,x)),e(y,z)))$

Problems 85-88 depicted in Figure 7.5 are to show the four formulas equivalent in a circular manner. The results also correspond to the other experiments presented before.

#	Theorem	OTTER	PURR$_{Literal}$-1	PURR$_{Link}$-1	PURR$_{Link}$-2
85	YQM ⇒ QYF	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$	$\boxed{<\mathbf{1}}$
86	WO ⇒ YQM	6.9	2.7	1.4	$\boxed{<\mathbf{1}}$
87	XGJ ⇒ WO	fail	fail	fail	fail
88	QYF ⇒ XGJ	13.9	5.3	1.7	$\boxed{<\mathbf{1}}$

Figure 7.5: Experiments with the R Calculus

Two-Valued Sentential Calculi. The CN calculus is a version of the two-valued sentential calculus. We present three challenging examples. The operators i and n are intended to mean implication and negation. Each of the following formulas holds in CN:

(CN-1) $i(i(x,y),i(i(y,z),i(x,z)))$
(CN-2) $i(i(n(x),x),x)$
(CN-3) $i(x,i(n(x),y))$
(CN-59) $i(i(n(x),z),i(i(y,z),i(i(x,y),z)))$
(CN-60) $i(i(x,i(n(y),z)),i(x,i(i(u,z),i(i(y,u),z))))$
(CN-CAM) $i(i(i(i(i(x,y),i(n(z),n(u))),z),v),i(i(v,x),i(u,x)))$

Lukasiewicz axiomatized CN with {CN-1,CN-2,CN-3}. The three problems 22–24 in Table 7.6 are to derive CN-59, CN-60, and CN-CAM from {CN-1,CN-2,CN-3}.

#	Theorem	Proof Time	Inferred	Stored	Memory	Rate
22	CN-1,CN-2,CN-3 ⇒ CN-59	7.2h	127mio	62000	11Mb	4900
23	CN-1,CN-2,CN-3 ⇒ CN-60	10.9h	160mio	1.8mio	296Mb	4020
24	CN-1,CN-2,CN-3 ⇒ CN-CAM	fail after 3.5h	?	4.8mio	512Mb	?

Figure 7.6: Experiments with PURR$_{Literal}$-1 on the Two-Valued Sentential Calculus

For the experiments in Table 7.6 we employed a SuperSparc 10 computer with a maximum memory of 512 megabytes RAM. The results show that PURR is able to operate at the limits of currently available workstation hardware. The system even has a reasonable fast inference rate when working on problem 24 after filling 500Mb of memory with one substitution tree containing about 4.8 million clauses. The average inference rate of about 5000 inferences per second in problem 22 and still 4000 inferences per second in problem 23 also reveal the advantages of the employed indexing techniques.

The difficulties with the problems 22–24 arise due to relatively long denied theorems. Most of the proof time the system works with short clauses, since Purr prefers to infer with the shortest clauses. Thus the difficulties increase with the length of the theorem. However, some of the short clauses are involved in the proof and therefore have to be taken into account. Purr finds a refutation for the problems 22 and 23 if we limit the number of symbols per unit clause to the number of symbols of the denied theorem.

7.2 Parallelism

The exploitation of parallelism in automated theorem proving is truely a challenging task. Many experimental runs of Purr were needed to create a prover which performs well even on large clause sets. Note that our condensed detachment examples were relatively easy to master, since their parallelization potential is small. The condensed detachment examples with Purr$_{Link}$-2 represent only a first steps towards massive parallelism.

In this section we present the results of three problems with larger clause sets. The computation were executed on a multiprocessor machine containing four SuperSparc processors.

Sam's Lemma. We used an axiomatization of the well-known Sam's Lemma [MOW76b] contained in the TPTP library. It consists of 29 clauses with 17 clauses being unit clauses. All clauses contain only constants and variables. Function symbols do not occur. Purr creates 13 slave processes in the Purr$_{Literal}$-4 configuration: Each non-unit clause is represented by a resolution process. Additionally, a single unit terminator process is required to detect termination.

The 13 slave processes consumed 29 cpu-seconds to find a proof with length 25. The elapsed real time was 9 seconds with four processors yielding a speedup of about three. This result converges the theoretical speedup of four on the employed machine.

Schubert's Steam-Roller. The axiomatization of Schubert's Steam-Roller [Pel86] contained in the TPTP library consists of 26 clauses. Six clauses are unit clauses. Purr creates 24 slave processes in the Purr$_{Literal}$-4 configuration: The non-unit clauses are represented by a single resolution process. Additionally, two non-unit clauses are processed by two and three resolution processes since the clauses have outgoing links at more than one literal. A single unit terminator process is required to detect termination.

The 24 slave processes consumed about 1.5 cpu-seconds to find a proof with 19 steps. The elapsed real time also was 1.5 seconds with four processors yielding no speedup. This result is due to the overhead for the creation of the slave processes.

Group Theory. The clause set depicted in Figure 7.7 belongs to the domain of group theory. We proof that if the square of every element is the identity, the system is commutative. The exact axiomatization of this problem also has been taken from the TPTP library. The equality relation is defined by the reflexivity, symmetry, and transitivity axioms.

Purr demands six processes: Five resolution processes work on the symmetry, transitivity, and the three substitution axioms. One unit terminator process is used for the equality symbol.

Axioms (Units)

Reflexivity	$\{equal(x, x)\}$
Left Identity	$\{equal(multiply(identity, x), x)\}$
Right Identity	$\{equal(multiply(x, identity), x)\}$
Left Inverse	$\{equal(multiply(inverse(x), x), identity)\}$
Right Inverse	$\{equal(multiply(x, inverse(x)), identity)\}$
Associativity	$\{equal(multiply(multiply(x, y), z), multiply(x, multiply(y, z)))\}$

Axioms (Non-Units)

Symmetry	$\{\neg equal(x, y), equal(y, x)\}$
Transitivity	$\{\neg equal(x, y), \neg equal(y, z), equal(x, z)\}$
Inverse Substitution	$\{\neg equal(x, y), equal(inverse(x), inverse(y))\}$
Multiply Substitution	$\{\neg equal(x, y), equal(multiply(x, z), multiply(y, z))\}$
Multiply Substitution	$\{\neg equal(x, y), equal(multiply(z, x), multiply(z, y))\}$

Hypotheses

Squareness	$\{equal(multiply(x, x), identity)\}$
a times b is c	$\{equal(multiply(a, b), c)\}$

Theorem

Prove: b times a is c	$\{\neg equal(multiply(b, a), c)\}$

Figure 7.7: Clause Set in Group Theory

Altogether, the six processes consumed 45 cpu-seconds. The elapsed real time was 31 seconds with Purr$_{Literal}$-4 yielding a speedup of about one and a half.

The somewhat disappointing speedup is due to the different shape of the involved clauses. Obviously, the resolution process for the transitivity axiom has to perform the most complex merge operation since this axiom is the only one with three literals. The remaining resolution processes actually do not perform any merge. Consequently, the transitivity process consumes most of the cpu-time while other processes remain more or less idle waiting for new information. This effect directly corresponds to the data dependence inherently to the problem. Therefore, in our approach improved speedup for this problem is difficult to achieve.

Since Purr is an asynchronous parallel system, the maintenance of unprocessed messages may become a problem. Consider the transitivity axiom: The negative literals of this axiom are connected to all positive literals in the clause set. Thus the transitivity process receives new inferences from all resolution processes including the messages created by itself. However, this process cannot receive and process all created information. Most of the data is collected in growing message queues. This effect becomes a serious problem when running Purr on more difficult problems with large clause sets.

8

Conclusion

We have presented the theoretical background and implementational details of the distributed theorem prover PURR. The prover has been developed in order to investigate *indexing techniques* and to exploit *parallelism* in automated reasoning. Both indexing and distributed processing have been proved to be powerful means for accelerating theorem proving systems.

We have extended the unit resulting resolution rule to work on sets of substitutions. To this end, we introduced a *clause graph*. The nodes of the clause graph correspond to literals in the clause set that represent the problem to be solved. Two literals are connected by a link if the literals are complementary and unifiable. In other words, connected literals are possible ur-resolution partners. The clause graph does not change during the reasoning phase.

New ur-resolvents are represented by substitutions which are collected in sets of substitutions. These substitution sets are attached to nodes and links in the clause graph. We have shown that substitution sets can be represented by indexes in a natural and elegant manner. In PURR indexes become the fundamental data structure instead of the usual clauses and literals. New ur-resolvents are exchanged among substitution sets in the form of indexes. Reasoning-based operations like subsumption and the computation of simultaneous unifiers are extended to set operations based on indexing techniques. Moreover, the unit resulting resolution rule is concurrently applied to different nuclei.

High Inference Rate. In all experiments in Chapter 7 PURR showed very competitive behavior compared to OTTER even on single processor machines. First, the n:m indexing techniques in PURR improve the performance compared to the standard n:1 indexing techniques. The n:m indexing techniques also have been tested in isolation on large sets of substitutions by Graf and Meyer [GM96]. Second, PURR has a slightly different search behavior due to the selection of whole sets of given clauses. Accordingly, PURR performs a limited breadth-first search which sometimes results in extremely short proof times.

The impact of parallelism on the inference rate has not yet been examined as much as the experiments on single processor systems. Our tests merely suggest a reasonable behavior of

PURR on two and four processor machines. Experiments on massively parallel hardware are future work.

Large Sets of Inferences. In Chapter 7 we also presented experiments in which PURR had to maintain extremely large inference sets containing millions of substitutions. However, the system showed a reasonable fast inference rate. Typically, the substitution trees containing the ur-resolvents of a resolution process become very large. Thus the n:1 subsumption and insertion operations integrated into the multi-merge operation are affected as well as the selection of given substitutions. Therefore, we conclude that these operations also perform well on huge substitution trees.

Indexing Supports Distributed Automated Reasoning. In Chapter 6 we presented a transformation of substitution trees into process-independent form. The process-independent representation of a substitution tree corresponds to a string of integer values which can directly be transmitted between parallel processes. This representation usually saves one third of the memory allocated for the substitution tree with conventional term representation.

Decentralized Distributed Processing. The parallel concept of PURR is based on a decentralized structure of processes. There is no central process for subsumption or any other time-consuming operation. Every process performs subsumption by itself. High locality of the individual process reduces the probability of bottlenecks. However, the tradeoff between receiving and processing information is still a problem for processes with complex operations and many input channels. Nevertheless, avoiding central subsumption seems to improve the possible speedup significantly.

Balance of Parallelism and Communication is Difficult. An optimal balance of parallelism and communication will achieve the best possible speedup. The right balance depends on the problem and on the hardware employed. This work only addresses the problem-related balancing. To this end, we developed two variants of PURR. One variant employs resolution processes for each link in the clause graph whereas the other variant uses resolution processes for each literal with outgoing links. The former variant achieves finer granularity than the latter. Ideally, the granularity is not fixed but can arbitrarily be selected. In this work flexible granularity turned out to be a very important feature in order to study parallelism in automated reasoning. Another open problem is to decide in advance which granularity achieves better speedup than others. Currently, PURR cannot associate processes with different granularity to the clause graph.

Bibliography

[AO83] Grigorios Antoniou and Hans Jürgen Ohlbach. Terminator. In Alan Bundy, editor, *Proc. of 8^{th} International Joint Conference on Artificial Intelligence, IJCAI-83, Karlsruhe*, pages 916–919, 1983.

[Aßm92] U. Aßmann. *Parallele Modelle für Deduktionssysteme.* PhD thesis, Infix, Köln, 1992.

[BB92] K.H. Bläsius and H.J. Bürckert. *Deduktionssysteme.* Oldenbourg, 1992.

[BE93] Ralph M. Butler and L. Lusk Ewing. Monitors, messages, and clusters: the p4 parallel programming system. Technical report, University of North Florida, Argonne National Laboratory, http://www.mcs.anl.gov, 1993.

[BLL93] Ralph M. Butler, Alan L. Leveton, and Ewing L. Lusk. p4-linda: A portable implementation of linda. Technical Report MCS-P374-0793, University of North Florida, Argonne National Laboratory, 1993.

[CG90] Nicholas Carriero and David Gelernter. *How to write parallel programs : a first course.* MIT Press, Cambridge, MA, 1990.

[CS79] C.L. Chang and J.R. Slagle. Using rewriting rules for connection graphs to prove theorems. *Artificial Intelligence*, 12:159–180, 1979.

[DGMJ93] Jack J. Dongarra, Al Geist, Robert Manchek, and Weicheng Jiang. Using PVM 3.0 to run grand challenge applications on a heterogeneous network of parallel computers. In Richard F. Sincovec, David E. Keyes, Michael R. Leuze, Linda R. Petzold, and Daniel A. Reed, editors, *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 873–877, Norfolk, VI, March 1993. SIAM Press.

[Eis91] Norbert Eisinger. *Completeness, Confluence, and Related Properties of Clause Graph Resolution.* Research Notes in Artificial Intelligence. Pitman Ltd., London, 1991.

[For93] The MPI Forum. MPI: A message passing interface. In Bob Borchers, editor, *Proceedings of the Supercomputing '93 Conference*, pages 878–885, Portland, OR, November 1993. IEEE Computer Society Press.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : parallel virtual machine; a user's guide and tutorial for* *network parallel computing.* Scientific and engineering computation series. MIT Press, Cambridge, MA, 1994.

[Gel85] David Gelernter. Generative communication in Linda. *acm Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1985.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI : portable parallel programming with the message-passing interface.* Scientific and engineering computing series. MIT Press, Cambridge, MA, 1994.

[GM93] P. Graf and C. Meyer. Extended path-indexing. Technical Report MPI-I-93-253, Max-Planck-Institut für Informatik, Saarbrücken, Germany, December 1993. Full version of [Gra94a].

[GM96] P. Graf and C. Meyer. Advanced indexing operations on substitution trees. 1996. Submitted to CADE-13.

[Gra94a] P. Graf. Extended path-indexing. In *12th Conference on Automated Deduction*, pages 514–528. Springer LNAI 814, 1994.

[Gra94b] P. Graf. Substitution tree indexing. Technical Report MPI-I-94-251, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1994. Full version of [Gra95].

[Gra95] P. Graf. Substitution tree indexing. In *6th International Conference on Rewriting Techniques and Applications RTA-95*, pages 117–131. Springer LNCS 914, 1995.

[Gra96] P. Graf. *Term Indexing.* Springer LNAI series, 1996. To appear.

[Har91] R. J. Harrison. Portable tools and applications for parallel computers. *International Journal of Quantum Chemistry*, 40:847–863, 1991.

[Hwa93] Kai Hwang. *Advanced computer architecture; parallelism, scalability, programmability.* McGraw-Hill, New York, 1993.

[IPC] Current world wide web home page for the internet parallel computing archive: *http://www.hensa.ac.uk/parallel/.*

[Kow75] Robert Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22(4):572–595, October 1975.

[Kur91] Franz Kurfeß. *Parallelism in logic : its potential for performance and program development.* Artificial intelligence. Vieweg, Wiesbaden, 1991.

[LMM$^+$57] E.J. Lemmon, C.A. Meredith, D. Meredith, A.N. Prior, and I. Thomas. Calculi of pure strict implication. Technical report, Canterbury University College, Christchurch, 1957. Reprinted in *Philosophical Logic*, Reidel, 1970.

[Łuk70] J. Łukasiewicz. *Selected Works.* North Holland, 1970. Edited by L. Borkowski.

[Mat94] T. G. Mattson. Programming environments for parallel computing: A comparison of cps, linda, p4, pvm, posybl, and tcgmsg. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 27th Annual Hawaii International Conference* *on System Sciences. Volume 2 : Software Technology*, pages 586–594, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[McC94] W. McCune. Otter 3.0 reference manual and guide. Report ANL-94 6, Argonne National Laboratory, January 1994.

[MOW76a] J.D. McCharen, R. Overbeek, and L. Wos. Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications*, 2:1–16, 1976.

[MOW76b] J.D. McCharen, R. Overbeek, and L. Wos. Problems and experiments for and with automated theorem proving programs. *IEEE Transactions on Computers C-25(8)*, pages 773–782, 1976.

[MP95] Joseph W. Manke and James C. Patterson. Message passing performance of intel paragon, ibm sp1 and cray t3d using pvm. In David H. Bailey, Petter E. Bjørstad, John E. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon, and Layne T. Watson, editors, *Proceedings of the 27th Conference on Parallel Processing for Scientific Computing*, pages 768–769, Philadelphia, 1995. SIAM Press.

[MPI] Current world wide web home page for the message passing interface standard: *http://www.mcs.anl.gov/mpi/*.

[MW92] W. McCune and L. Wos. Experiments in automated deduction with condensed detachment. In *11th International Conference on Automated Deduction*, pages 209–223. Springer, LNAI 607, 1992.

[Ohl90] H.J. Ohlbach. Abstraction tree indexing for terms. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 479–484. Pitman Publishing, London, August 1990.

[Pel86] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986. Errata: *Journal of Automated Reasoning*, 4(2):235–236,1988.

[Rob65] J.A. Robinson. Automated deduction with hyper-resolution. *International Journal of Comp. Mathematics*, 1:227–234, 1965.

[Sch92] G. Schoinas. Issues on the implementation of programming system for distributed applications. Technical report, University of Crete, 1992.

[SSY94] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 252–266, Berlin, 1994. Springer.

[Tur93] Louis H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Research Center for Computational Field Simulation, P.O. Box 6176, Mississippi State, MS 39762, 1993.

Index

Symbols

- MRG set of merged substitution sets 23
- REC set of received substitution sets 23
- RES set of resolved substitution sets 23
- SNT set of sent substitution sets 23
- $\bullet$ join of substitutions 9
- s state flag 37
- $|$ sequence concatenation 8
- $\sigma|_U$ restricted substitution 9
- σ send substitution 20
- $\times$ merge of substitution sets 10
- $\times$ merge of substitutions 10
- τ test substitution 20
- w weight 37
- $\mathsf{arity}(t)$ arity of a term 7
- $\mathsf{COD}(\sigma)$ codomain of a substitution 9
- $\mathsf{depth}(t)$ depth of a term 7
- $\mathsf{DOM}(\sigma)$ domain of a substitution 8
- ε empty tree 12
- $\mathbf{F}$ function symbols 6
- $\mathsf{IM}(\sigma)$ image of a substitution 9
- $\mathsf{nodes}(T)$ nodes of a tree 12
- $\mathsf{O}(t)$ positions of a term 8
- $\mathbf{P}$ predicate symbols 6
- $\mathsf{sons}(T)$ sons of a tree 12
- $\mathsf{top}(t)$ top symbol of a term 7
- $\mathbf{T}$ set of terms 6
- $\mathbf{V}$ variable symbols 6
- $\mathsf{VAR}(t)$ variables in term t 7
- $\mathbf{V}^*$ indicator variable symbols 6
- G test function generalizations 35
- I test function instances 35
- U test function unifiable terms 35
- V test function variants 35

A

- alogrithms
  - notation of 12
- and-parallelism 48
- atom 7

B

- blocking receive 50
- broadcast communication pattern 51

C

- clash 10
  - direct 10
  - indirect 10
- clause 7
  - query 16
  - unit 7
- clause graph 13, 15, **20**
  - creation of 64
  - directed 15
  - initial state 24
  - labeled directed 16, **20**
  - optimization of 65
  - undirected 15
- codomain 9
- common variable 79
- communication 47, **50**
  - indexing and 81
  - interprocess 48
  - pattern 51
    - broadcast 51
    - conference 51
    - multicast 51
    - unicast 51
- complementary 7
- condensed detachment 85, **86**
- conference communication pattern 51
- constant 6
- context 78

D

- deadlock 51
  - detection 51
- dependence 47
  - data 50
- domain 8

F

- father 12
- function symbol 6

G

- generalization 10
- given substitution 24
- grain size 51
- granularity 51
  - coarse 51
  - fine 51
  - medium 51
- graph 11
  - acyclic 11
  - clause *see* clause graph
  - directed 11
  - directed acyclic *see* tree
  - labeled 11
  - labeled directed *see* clause graph
  - path in 11
  - undirected 11
- group theory 92

H

- hyperresolution 16

I

- image 9
- indexing 31
  - classification of 31
  - maintenance of type n:1 and n:m 32
  - retrieval of type 1:1, n:1, and n:m 32
  - substitution tree 33
- indicator variable 11
- input subsumption 28
- instance 10
- internal link 16
- interprocess communication 48
- IPC *see* interprocess communication

L

- labeled link 16
- latency 51
- Ldcg *see* clause graph, labeled directed
- lightest substitution 24
- lightest subtree 44
  - weight of 44
- linear term 7
- link
  - internal 16
  - labeled 16
- literal 7
  - receiver 16
  - sender 16
- load balancing 51
- Lst *see* substitution tree, linear

M

- master process 63
- master/slave paradigm 63
- matcher 10
- merged substitution 22
- message passing interface 52
- message queue 50
- message-passing model 50
  - asynchronous 50
  - synchronous 50
- mgu *see* most general unifier
- most general unifier 10
  - splitted 16, **20**
- MPI *see* message passing interface
- multi-merge 32, **42**
  - n:m algorithm 59
  - n:m indexing task 43
- multicast communication pattern 51

N

- nonblocking receive 50
- normalization 33
  - of substitutions 11
  - of terms 11

O

occur-check . . . 10
options . . . 77
or-parallelism . . . 48
origin list . . . 76
OTTER . . . 77, 85
output subsumption . . . 28

P

parallel programming model . . . 48
  message-passing . . . 50
  shared-memory . . . 50
  shared-variable . . . 48
parallel programming system . . . 52
  C-Linda . . . 53
  p4 . . . 53
  p4-Linda . . . 53
  POSYBL . . . 53
  PVM . . . 53, **54**
  TCGMSG . . . 52
parallelism
  and- . . . 48
  in logic . . . 47
  in practice . . . 48
  or- . . . 48
position . . . 8
postprocessing . . . 75
predicate symbol . . . 6
preprocessing . . . 64
private variable . . . 79
process . . . 48
  communication and indexing . . . 81
  independent representation . . . 81
  master . . . 63
  resolution . . . 68
  slave . . . 63
  terminator . . . 68
  unit terminator . . . 71
program . . . 48
PROLOG . . . 33
proof generation . . . 75
PURR . . . 1
PVM . . . 54

Q

query clause . . . 16

R

reasoning phase . . . 66
received substitution . . . 22
receiver literal . . . 16
renaming . . . 10
resolution
  unit resulting . . . 13
resolution process . . . 68
  on a single link . . . 68
  on some links . . . 70
resolved substitution . . . 22
restriction . . . 9

S

sam's lemma . . . 85, **92**
selection . . . 44
  n:m algorithm . . . 62
  n:m indexing task . . . 45
send instantiation . . . 18, **23**, 80
send substitution . . . 18, **20**
sender literal . . . 16
sent substitution . . . 23
SEST . . . *see* substitution tree, selective
set of support . . . 16
shared-memory model . . . 50
  Linda . . . 53
shared-variable model . . . 48
signal . . . 74
signal handler . . . 74
slave process . . . 63
son . . . 12
SOS . . . *see* set of support
speedup . . . 47, **50**
state flag . . . 37
state of selection . . . 44
static scheduling . . . 51
steam-roller . . . 85, **92**
strategy . . . 24
substitution . . . 8
  compatible . . . **10**, 33
  composition of . . . 9
  given . . . 24
  idempotent . . . 9
  join of . . . 9
  lightest . . . 24
  merge of . . . 10

- merged 22
- process-independent 82
- received 22
- resolved 22
- sent 23

substitution sets
- merge of 10

substitution tree 34
- in Purr 36
- linear 36
- process-independent 82
- selective 37, **37**
- weighted 36

substitution tree indexing 33

subsumption 40
- *n*:*m* algorithm 57
- *n*:*m* indexing task 40
- input 28
- output 28
- transition system 28

symbol
- process-independent 83

synchronization latency 51

T

term 6
- arity 7
- at position 8
- depth 7
- ground 7
- linear 7
- position in 8
- process-independent 82

terminator process 68
- on a clause 71
- on unit resolvents 71

test substitution 18, **20**

test unification 18, **23**, 80

top symbol 7

TPTP problem library 85

transition system
- PURR with subsumption 29
- PURR without subsumption 24
- SUBSUMPTION 28

tree 12
- ordered 12

U

unicast communication pattern 51

unifiable 10

unifier 10

union 42
- *n*:*m* algorithm 59
- *n*:*m* indexing task 42

unit 7

unit resulting resolution 13
- on substitution sets 21

unit terminator process 71

V

variable 6
- common 79
- indicator 33
- private 79

vertex 11

W

weight 37

Wst .... *see* substitution tree, weighted