RELAXATION PROCESSES: THEORY, CASE STUDIES AND APPLICATIONS

Ching-Tsun Chou

February 1986
CSD-860057

UNIVERSITY OF CALIFORNIA

Los Angeles

Relaxation Processes:
Theory, Case Studies and Applications

A thesis submitted in partial satisfaction of the

requirements for the degree of Master of Science

in Computer Science

by

Ching-Tsun Chou

1986

The thesis of Ching-Tsun Chou is approved.

Buz

Robert Uzgalis

S Greibach

Sheila Greibach

David E. Martin

David Martin

[illegible]

D. Stott Parker, Committee Chair

University of California, Los Angeles

1986

To my

Mom and Dad

TABLE OF CONTENTS

page

1. **Introduction** 1
   1.1. What Are Relaxation Processes? A Very Abstract Discussion 1
   1.2. An Outline of this Thesis 2

2. **Some Typical Examples of Relaxation Processes** 4
   2.1. (Single-Source) Shortest-Path Problem 4
   2.2. Term Inequalities 10
   2.3. Monadic Term Inequalities: Prefix Inequalities 12

3. **A Formal Theory of Relaxation Processes** 16
   3.1. Relaxation Problems 16
   3.2. Examples of Relaxation Problems 17
   3.3. General Relaxation Semi-Algorithm 21
   3.4. Applications to SPP and PI 24

4. **Two Case Studies on Relaxation Processes** 27
   4.1. Possible Non-Termination of GRSA 27
   4.2. Case Study I: LI-Formulated Shortest-Path Problem 30
   4.3. Case Study II: Prefix Inequalities 35

5. **An Application of Relaxation Techniques: Type Inference for Prolog** 40
   5.1. A Polymorphic Type System for Prolog 40
   5.2. Type Inference and Term Inequalities 45
   5.3. TYPEINF: A Polymorphic Type Inferencer for Prolog 47
   5.4. Demonstrations of the Capabilities of TYPEINF 49

6. **Directions of Future Research** 53

Bibliography 55

Appendix I. Some Mathematical Notions and Terminologies 56
   I.1. General Terminologies 56
   I.2. Graphs 56
   I.3. Partial Orders and Posets 57

Appendix II. Program Listing and Test Data for **TYPEINF** 59

LIST OF FIGURES

page

2.1.1. An Example of (Single-Source) Shortest-Path Problem 8

3.3.1. General Relaxation Semi-Algorithm (GRSA). 22

3.4.1. RSA-SPP: Relaxation Semi-Algorithm for LI-formulated SPP. 24

3.4.2. RSA-PI: Relaxation Semi-Algorithm for PI. 26

4.1.1. A Shortest-Path Problem with a Negative Cycle. 28

4.1.2. An Instance of PI with a Negative Cycle. 30

4.2.1. RA-SPP: A Relaxation Algorithm for LI-formulated SPP. 34

4.3.1. RA-PI: A Relaxation Algorithm for PI. 38

ACKNOWLEDGEMENTS

I wish to express my deep gratitude to my advisor, Professor Stott Parker, whose guidance, encouragement and patience were essential to the completion of my Thesis. I thank all of the other "Relaxers", Kam Chow, Paul Eggert, Koenraad Lecot, Reichi Lee and Joan Marshall, for many stimulating discussions in our Relaxation Meetings. I also acknowledge the UC-MICRO grant "RAPPORT" (co-sponsored by IBM) for financial assistance.

I am grateful to the members of my Thesis Committee, Professors Sheila Greibach, David Martin and Robert Uzgalis, for their helpful suggestions during the preparation of this Thesis. Professor Greibach is also thanked for many interesting discussions. Buz is also thanked for his tasteful inputs which helped in the fermentation and fruition of my concepts.

I am deeply indebted to Ping Wei for her encouragement, patience and understanding. Her company was greatly enjoyed.

Finally, I wish to thank my parents for their endless love for me. This Thesis is dedicated to them.

ABSTRACT OF THE THESIS

Relaxation Processes: Theory, Case Studies and Applications

by

Ching-Tsun Chou

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor D. Stott Parker, Chair

Relaxation is a powerful problem-solving paradigm in coping with problems specified using constraints. In this Thesis we present a study of the nature of relaxation processes. We begin with identifying certain typical problems solvable by relaxation. Motivated by these concrete examples, we develop a formal theory of relaxation processes and design the General Relaxation Semi-Algorithm for solving general Relaxation Problems. To strengthen the theory, we do case studies on two relaxation-solvable problems: the Shortest-Path Problem and Prefix Inequalities. The principal results of these studies are polynomial-time algorithms for both problems. The practical usefulness of relaxation is demonstrated by implementing a program called **TYPEINF**, which employs relaxation techniques to automatically infer types for Prolog programs. Finally we indicate some possible directions of future research.

CHAPTER 1.
Introduction

The purpose of this Thesis is to present a study of the nature of *relaxation processes*. Both theoretical and practical aspects of relaxation processes will be investigated. For the theoretical aspect, we shall not only develop a general theory of relaxation processes, but also examine two specific cases in depth. For the practical aspect, we shall apply relaxation techniques to the implementation of a type inference program for Prolog.

Now let us start with a very abstract discussion on relaxation processes.

1.1. What Are Relaxation Processes? A Very Abstract Discussion

Relaxation is a powerful problem-solving paradigm in coping with problems which are specified using *constraints*. The process through which a problem is solved using relaxation techniques is (of course) called a *relaxation process*. Generally speaking, the relaxation paradigm of problem solving exhibits the following characteristics:

(i) A problem to be solved by relaxation techniques has a *constraint satisfication problem* as a subset of its specification. The solution to be found is required to satisfy all constraints and (possibly) some additional optimization criteria.

(ii) A relaxation process begins with a very crude "guess" of the final solution to the underlying problem, which satisfies only part of the constraints. This guess is called the *initial approximation.*

(iii) The relaxation process proceeds to satisfy some of the violated constraints by *locally* adjusting the current approximation. In most cases, it takes many *iterations* through this adjustment step, which we shall call a *relaxation step,* before the current approximation can be adjusted to satisfy all constraints. This final approximation is the solution we are looking for. Notice that if there are any optimization requirements to be met, they are taken care of in the selection of the initial approximation and the relaxation steps.

(iv) Relaxation processes are *non-deterministic*, in the sense that before the final solution is reached, there are usually many different orders in which relaxation steps can be applied. The interesting fact is that the order does not affect the arrival at a correct solution but only the *rate* of convergence to this solution.

The reader should, however, be aware that these characteristics by no means constitute a precise definition of either relaxation problems or relaxation processes. They are just meant to give the reader some flavor of what relaxation is. We will discuss relaxation in a more precise way in the remaining chapters of this Thesis, which are outlined in the next section.

1.2. An Outline of this Thesis

This Thesis consists of six chapters and two appendices. The remaining chapters and appendices are briefly described in the following paragraphs.

In Chapter 2 some specific examples of problems subject to solution by relaxation techniques are introduced. The first is the classical single-source *Shortest-Path Problem*. It is shown that Shortest-Path Problem can be formulated using a set of *Linear Inequalities*, which typify constraint-satisfaction problems. Then we illustrate by means of examples how to solve the Linear-Inequality version of Shortest-Path Problem using relaxation techniques. The second example of relaxation-solvable problems is called *Term Inequalities*. This problem resulted naturally from the author's work in implementing a type inference program for Prolog, the details of which will be discussed in Chapter 5. In Chapter 2 we content ourselves with defining the problem of Term Inequalities and illustrating its solution by relaxation techniques through some examples. The third, and the last, example of relaxation-solvable problems is a restricted version of Term Inequalities, called *Prefix Inequalities*, which will be the subject of one of the case studies in Chapter 4.

Motivated by the concrete examples in Chapter 2, a formal theory of relaxation processes is developed in Chapter 3. We begin with defining what a *Relaxation Problem* is. It is shown that both the (Linear Inequality version of) Shortest-Path Problem and the problem of Prefix Inequalities are indeed special cases of Relaxation Problems. Then we introduce a semi-algorithm, called the *General Relaxation Semi-Algorithm*, for solving Relaxation Problems. It is called a *semi*-algorithm because its termination is not guaranteed. Several useful properties of this semi-algorithm are proved. They are used to obtain some results about Relaxation Problems themselves. Finally, we specialize the General Relaxation Semi-Algorithm to solve the Shortest-Path Problem and Prefix Inequalities, and investigate what consequences the new points of view have.

However, the reader should be aware of the fact that there is *not* a definition of either relaxation problems or relaxation processes upon which everyone would agree. The "definition" of Relaxation Problems in Chapter 3 is given in such a way that it encompasses the examples discussed in Chapter 2 and permits the development of an interesting theory of relaxation processes.

The theory developed in Chapter 3 is not strong enough, however, in the sense that we do not have a must-terminate *algorithm*, but just a *semi*-algorithm, for Relaxation Problems. Therefore, in Chapter 4 we will present the case studies on two specific Relaxation Problems -- the Shortest-Path Problem and Prefix Inequalities. The major results of these studies are improvements to the General Relaxation Semi-Algorithm which produce two *polynomial-time algorithms* for the problems under study. The reader will also notice the interesting similarities between the

Shortest-Path Problem and Prefix Inequalities, which become most apparent in Chapter 4.

In Chapter 5 our investigation turns to the practical aspects of relaxation techniques. Built upon the work of Mycroft and O'Keefe in [MyOK], we will describe a type inference program for Prolog, called **TYPEINF**, in this chapter. First, we introduce a type system for Prolog and explain why we need a type inference program. Then it is shown that the task of type inference can be reduced to that of solving Term Inequalities discussed in Chapter 2. This is followed by an outline of the type inference strategies adopted by **TYPEINF**. Finally, we demonstrate the capabilities of **TYPEINF** by applying it to some test data, one of which is *itself*!

In the last chapter, the author, in a tentative manner, points out some possible directions of future research in relaxation processes. This chapter is much more speculative and much less rigorous than its preceding chapters.

There are two appendices at the end of this Thesis. The first one of them is a short summary of some mathematical notions and terminologies used in this Thesis. The second is a complete program listing and some test data for **TYPEINF**.

CHAPTER 2.
Some Typical Examples of Relaxation Processes

In this Chapter we examine some typical examples of relaxation problems and their solution through relaxation processes. The purposes are: (1) to give the reader some concrete ideas as to what relaxation is and, (2) to motivate the development of a formal theory of relaxation processes in Chapter 3.

Three examples are examined. The first example is a classical problem in Operations Research --- the (single-source) *Shortest-Path Problem*. The second example is called *Term Inequalities*, which originated from the work in automatic type inference for Prolog. Please see Chapter 5 for details. Term Inequalities have, unfortunately, proven to be resistant to the development of a formal theory. So, in the third example, we consider a restricted version of Term Inequalities, called *Prefix Inequalities*, in which only monadic functors are allowed.

It is striking that two problems with completely different origins (namely, the Shortest-Path Problem and the Term Inequalities) can be solved using very similar techniques. In the next chapter we will try to formalize the common aspects of these problems.

2.1. (Single-Source) Shortest-Path Problem

In this section we first give the traditional definition of the (single-source) *Shortest-Path Problem* (SPP, for short) in terms of an integer-labelled di-graph and paths within this di-graph. Then we re-formulate it using *Linear Inequalities* (LI), for the latter formulation is more suitable for solution by relaxation processes. We also discuss the relationships between these two different formulations. Finally, we use an example to illustrate the relaxation techniques for solving LI-formulated SPP.

What Is SPP?

Traditionally, an instance of SPP is specified as a problem of finding the lengths of the shortest paths from a given source node to every other nodes in a di-graph. More formally, we have:

Definition 2.1.1: An (instance of) *Shortest-Path Problem* (SPP) consists of:

(a) A di-graph $G = (V, A)$ with node set $V = \{0, 1, 2, \ldots, n\}$ and arc set $A = \{1, 2, \ldots, m\}$, for some $n \geq 0$, $m \geq 1$. Node 0 is distinguished from other nodes and called the *source node.*

(b) A length-labelling mapping $L : \mathbf{A} \rightarrow \mathbf{Z}$ which assigns to each arc $j \in \mathbf{A}$ an integer-valued *length*, L_j. The concept of lengths can be naturally extended to *paths*: the length $L(P)$ of a path P in G is the sum of lengths of all arcs on P. In particular, the length of an empty path is 0.

The problem is to evaluate these quantities:

$$D_i = \min\{L(P) \mid P \text{ is a path from node 0 to node } i.\},\ 0 \le i \le n, \qquad (2.1.1)$$

if they are well-defined. (See Remark (ii) below.) D_i, if it is well-defined, is called the *shortest distance* from the source node to node i. A path from the source node to node i whose length is the shortest distance D_i is called a *shortest path* from the source node to node i.

Remarks:

(i) We define $\min \varnothing = \infty$. (See Appendix I.) So, if node i is not reachable from the source node, then $D_i = \infty$, but the shortest path to node i is undefined.

(ii) If the set after the min operator in Eq. (2.1.1) contains arbitrarily small integers, D_i (and hence the shortest paths to node i) are *undefined*. This happens when, for example, node i is on a cycle of *negative* length which is reachable from the source node.

(iii) A simple consequence of the above observation is that, if D_0 is well-defined, then $D_0 = 0$. For, $D_0 < 0$ would imply the existence of a negative cycle on which node 0 resides, thus contradicting to the well-definedness of D_0. On the other hand, there is always the empty path from node 0 to itself with length 0.

(iv) Any algorithm for SPP allowing negative arc lengths must have the capability of detecting the kind of "pathological" conditions discussed in (ii). It will be shown in Chapter 4 that a relaxation algorithm for SPP with this capability *can* be implemented.

The Linear-Inequality Formulation of SPP

The formulation of SPP in the last subsection is intuitively appealing and expalins nicely why people used the term "Shortest-Path". However, it is not suitable for solution by relaxation techniques. Therefore, we introduce an alternative formulation of SPP in this subsection.

Consider an arbitrary arc $j \in \mathbf{A}$. By the very definition of the min operator in Eq. (2.1.1), the shortest distance from the source node to node $to(j)$ can not be greater than the shortest distance from the source node to node $from(j)$ plus the length L_j of arc j. Hence the following inequality[1] holds:

[1]It is interesting to note that this inequality resembles the famous *Triangle Inequality* in Euclidean geometry.

$$D_{from(j)} + L_j \geq D_{to(j)}.$$

Also note that, since there is at least the empty path from the source node to itself, we must have

$$0 \geq D_0.$$

In summary, whenever D_i's ($0 \leq i \leq n$) are all well-defined, each of the following inequalities holds.

$$\left.\begin{array}{c} D_{from(j)} + L_j \geq D_{to(j)},\ \ 1 \leq j \leq m \\ 0 \geq D_0 \end{array}\right\}$$

The preceding discussion motivates the following definition.

Definition 2.1.2: Given a (traditional) SPP as the one in Definition 2.1.1, its associated *Linear Inequalities* (LI) are defined to be

$$\left.\begin{array}{c} \mathbf{I}_j : x_{from(j)} + L_j \geq x_{to(j)},\ \ 1 \leq j \leq m \\ 0 \geq x_0 \end{array}\right\} \qquad (2.1.2)$$

An *LI-formulated SPP* is a problem of finding the *greatest* $(n+1)$-tuple of extended integers, in the natural order of poset products, satisfying the LI associated with a (traditional) SPP.

Equivalence of the Two Formulations of SPP

We have formulated SPP in two different ways: one as an optimization problem on di-graphs and the other as the solution of a set of Linear Inequalities. A question naturally arises: are these two formulations *equivalent*? We claim that they *are* equivalent, in the sense of the following theorem.

Theorem 2.1.3: Given an instance of SPP and its associated LI, the shortest distance to each node in the di-graph is well-defined if, and only if, there is a greatest tuple of extended integers which satisfies the associated LI. Moreover, that tuple of extended integers is exactly the tuple of shortest distances from the source node to all nodes in the di-graph.

In order to avoid repetitions of arguments, we do not prove this theorem in its entirety at this point. Instead, we will finish the proof part by part. The *only-if* part is proved in Lemma 3.4.1. The *if* part is proved in Lemma 4.2.4. Only the *moreover* part is proved in the lemma below.

Lemma 2.1.4: (With the same notation as that in Definitions 2.1.1 and 2.1.2.) Suppose that $(D_0, D_1, ..., D_n)$ is well-defined, and that there is a greatest tuple of extended integers, $(D'_0, D'_1, ..., D'_n)$, which satisfies every inequality in (2.1.2). Then $D_i = D'_i$, for all $0 \leq i \leq n$.

Proof: Assume the contrary: $D_{i_0} \neq D'_{i_0}$ for some i_0. Note that $(D_0, D_1, ..., D_n)$, when it is well-defined, satisfies each inequality in (2.1.2). So $D_{i_0} < D'_{i_0}$, since $(D'_0, D'_1, ..., D'_n)$ is the greatest tuple which satisfies (2.1.2). Hence $D_{i_0} < \infty$, so there is a shortest path P from node 0 to node i_0 with length D_{i_0}. Since $D_0 = 0 \geq D'_0$, there must exist an arc j on path P such that $D_{i_1} \geq D'_{i_1}$ but $D_{i_2} < D'_{i_2}$, where $i_1 = \mathit{from}(j)$ and $i_2 = \mathit{to}(j)$. Since it is a shortest path, P contains a shortest path from node 0 to every node on P. Hence it follows that

$$D_{i_1} + L_j = D_{i_2}.$$

But this implies that

$$D'_{i_1} + L_j \leq D_{i_1} + L_j = D_{i_2} < D'_{i_2},$$

that is,

$$D'_{i_1} + L_j < D'_{i_2},$$

contradicting to the fact that D'_i's satisfy Ineq. (2.1.2)! ■

Solving LI-Formulated SPP through Relaxation Processes

Example 2.1.1: Consider the graph $\mathbf{G} = (\mathbf{V}, \mathbf{A})$ depicted in Figure 2.1.1, where $\mathbf{V} = \{0, 1, 2, 3, 4\}$, $\mathbf{A} = \{1, 2, 3, 4, 5, 6, 7, 8\}$. The nodes are represented by circles labeled by node numbers. The arcs are represented by arrows which are directed from their *from* nodes to their *to* nodes. The number and the length of each arc are written adjacent to the corresponding arrow. The arc numbers are circled; the arc lengths are boxed. As usual, node 0 is the source node.

The LI associated with this instance of SPP are given below.

$$0 \geq x_0$$

$$x_0 + 4 \geq x_1 \qquad \text{(arc 1)}$$

$$x_1 + (-3) \geq x_4 \qquad \text{(arc 2)}$$

$$x_2 + 5 \geq x_1 \qquad \text{(arc 3)}$$

$$x_4 + 1 \geq x_2 \qquad \text{(arc 4)}$$

$$x_3 + (-2) \geq x_2 \qquad \text{(arc 5)}$$

$$x_4 + 0 \geq x_3 \qquad \text{(arc 6)}$$

$$x_3 + 4 \geq x_4 \qquad \text{(arc 7)}$$

$$x_0 + 2 \geq x_4 \qquad \text{(arc 8)}$$

Now, let's use the relaxation technique to find the greatest solution to the above

Two sequences of relaxation steps:

$(x_0, x_1, x_2, x_3, x_4) \; =$

$$\begin{tikzcd}
 & (0,\infty,\infty,\infty,\infty) \arrow[d, "\text{(arc 8)}"] \\
(0,\infty,\infty,\infty,\infty) \arrow[d, "\text{(arc 1)}"] & (0,\infty,\infty,\infty,2) \arrow[d, "\text{(arc 1)}"] \\
(0,4,\infty,\infty,\infty) \arrow[d, "\text{(arc 2)}"] & (0,4,\infty,\infty,2) \arrow[d, "\text{(arc 4)}"] \\
(0,4,\infty,\infty,1) \arrow[d, "\text{(arc 6)}"] & (0,4,3,\infty,2) \arrow[d, "\text{(arc 2)}"] \\
(0,4,\infty,1,1) \arrow[d, "\text{(arc 5)}"] & (0,4,3,\infty,1) \arrow[d, "\text{(arc 6)}"] \\
(0,4,-1,1,1) & (0,4,3,1,1) \arrow[d, "\text{(arc 5)}"] \\
 & (0,4,-1,1,1)
\end{tikzcd}$$

Figure 2.1.1. (Continued)

inequalities. We start with an overly conservative guess of the solution: $(x_0,x_1,x_2,x_3,x_4) = (0,\infty,\infty,\infty,\infty)$. This initial approximation satisfies some of the inequalities (those corresponding to arcs 2, 3, ..., 7) but violates others (arcs 1 and 8). Choose a violated inequality, say (arc 1). We can lower the value of x_1 to $0+4=4$ to make (arc 1) satisfied. Now, $(x_0,x_1,x_2,x_3,x_4) = (0,4,\infty,\infty,\infty)$. We say that inequality (arc 1) is *relaxed* by the lowering of the value of x_1. However, some other inequalities are violated (arcs 2 and 8). We can choose one of them to relax, say (arc 2). This makes $(x_0,x_1,x_2,x_3,x_4) = (0,4,\infty,\infty,1)$. At this stage, inequalities (arc 4) and (arc 6) are violated. We relax (arc 6), resulting in $(x_0,x_1,x_2,x_3,x_4) = (0,4,\infty,1,1)$. Finally, we relax (arc 5) and get

$(x_0, x_1, x_2, x_3, x_4) = (0, 4, -1, 1, 1)$. The reader can verify that every inequality is satisfied by this tuple of integers.

The above sequence of relaxation steps is, of course, not the only possible one. But it happens to be one of the shortest possible sequences of relaxation steps leading to the desired solution. We will prove in Chapter 3 that, under the assumption that no negative cycle exists, any possible sequence of relaxations leads to the correct answer (namely, the greatest tuple). ■

2.2. Term Inequalities

In this section another problem solvable by relaxation processes is introduced. It is called *Term Inequalities*. The need of solving Term Inequalities arises from the author's work in implementing a type inference program for Prolog, which will be described in detail in Chapter 5. Here the reader just need to know that the task of type inference for Prolog can be reduced to that of finding a most general solution to a system of Term Inequalities, whose meaning is made precise in the following subsection.

What Are Term Inequalities?

First, we define the concept of *terms* [En]. Let **Var** and **Func** be two disjoint, infinite sets of symbols. **Var** is the set of *variables* (ranged over by Greek letters near the beginning of the alphabet, possibly with subscripts). **Func** is the set of *functors* (ranged over by *Italic* words). Each functor $f \in$ **Func** has an *arity*, which is a non-negative integer. Functors of arity 0 are also called *constants*. The set **Term** of *terms* (ranged over by σ and τ, possibly with subscripts) is defined recursively in terms of **Var** and **Func** by the following three rules:

(a) Every variable is a term. (That is, **Var** $\subseteq$ **Term**.)

(b) If f is a functor of arity n and $\tau_1, \tau_2, \ldots, \tau_n$ are terms, then $f(\tau_1, \tau_2, \ldots, \tau_n)$ is also a term.

(c) Every term is generated through finitely many applications of the above two rules.

A term can be viewed either as a string composed of variables, functors and parentheses, or as a tree with internal nodes labelled by functors of arities ≥ 1 and leaves labelled by constants and variables. Either view works well in subsequent discussions.

Secondly we need the concept of *substitutions* (ranged over by θ, ϕ, ψ). Abstractly speaking, a substitution θ is simply a finite set of pairs of variables and terms, $\theta = \{\alpha_1/\tau_1, \alpha_2/\tau_2, \ldots, \alpha_n/\tau_n\}$, with the additional requirement that variables $\alpha_1, \alpha_2, \ldots, \alpha_n$ are all distinct. Given a term σ, we can *apply* the substitution θ to σ to obtain another term $\theta\sigma$ by substituting τ_i $(1 \leq i \leq n)$ for every occurrence of α_i in σ *simultaneously*. For example, let $\sigma = f(\alpha, \beta, \alpha)$ and $\theta = \{\alpha/\beta, \beta/g(\gamma)\}$. Then $\theta\sigma = f(\beta, g(\gamma), \beta)$.

Now we can define a *pre-order* on **Term**. Given two terms σ and τ, we say that σ is *more general than* τ, or, equivalently, τ is *an instance of* σ, iff there exists a substitution θ such that $\tau = \theta\sigma$, denoted by $\sigma \geq \tau$ or $\tau \leq \sigma$. We write $\sigma \approx \tau$ iff both $\sigma \geq \tau$ and $\sigma \leq \tau$. This means that σ and τ differ only in the names of their variables and that the substitutions involved are only *renamings* of variables. We write $\sigma > \tau$ (or $\tau < \sigma$) for the case that $\sigma \geq \tau$ but not $\sigma \approx \tau$.

A system of *Term Inequalities* (TI) is just a finite collection of inequalities of this form:

$$\sigma \geq \tau,$$

where σ and τ are terms. The goal in solving a system of TI is to find a substitution θ such that, after θ is applied to each of σ's and τ's, all inequalities are satisfied. We also require that the substitution θ we are trying to find is a *most general* one, in the sense that if ϕ is another substitution satisfying all inequalities, then there is a substitution ψ such that $\phi = \psi\theta$, the composition of substitution θ followed by substitution ψ. It is evident that the most general solution to a system of TI is unique up to renamings of variables, if it exists at all.

Solving Term Inequalities by Relaxation Technique

At this point the reader might have observed some similarities between TI and the LI formulation of SPP: both of them are expressed in terms of a finite collection of inequalities and demand a *greatest* solution which satisfies every inequality in the collection. Indeed, Term Inequalities can also be solved using similar relaxation techniques. This is illustrated in the following example.

Example 2.2.1: Consider the following system of Term Inequalities, which results from Example 5.3.1 where the type of the Prolog predicate **append/3** is inferred.

Approximation 0:

$$(\mathit{list}(\eta_1),\delta_2,\delta_2) \geq (\gamma_1,\gamma_2,\gamma_3) \tag{1*}$$

$$(\mathit{list}(\eta_3),\delta_6,\mathit{list}(\eta_3)) \geq (\gamma_1,\gamma_2,\gamma_3) \tag{2}$$

$$(\gamma_1,\gamma_2,\gamma_3) \geq (\mathit{list}(\eta_3),\delta_6,\mathit{list}(\eta_3)) \tag{3}$$

Observe that only Ineq. (3) is satisfied now, both Ineq. (1) and Ineq. (2) are violated. So, let's choose an inequality, say Ineq. (1), to relax. By *relaxing* a term inequality $\sigma \geq \tau$ we mean finding a most general substitution θ such that $\theta\sigma \geq \theta\tau$ really holds. For Ineq (1), one obvious substitution that relaxes it is

$$\{\ \gamma_1 / \mathit{list}(\eta_4),\ \gamma_2 / \eta_5,\ \gamma_3 / \eta_5\ \}.$$

After applying this substitution to all inequalities, the system becomes:

Approximation 1:

$$(\mathit{list}(\eta_1),\delta_2,\delta_2) \geq (\mathit{list}(\eta_4),\eta_5,\eta_5) \tag{1}$$

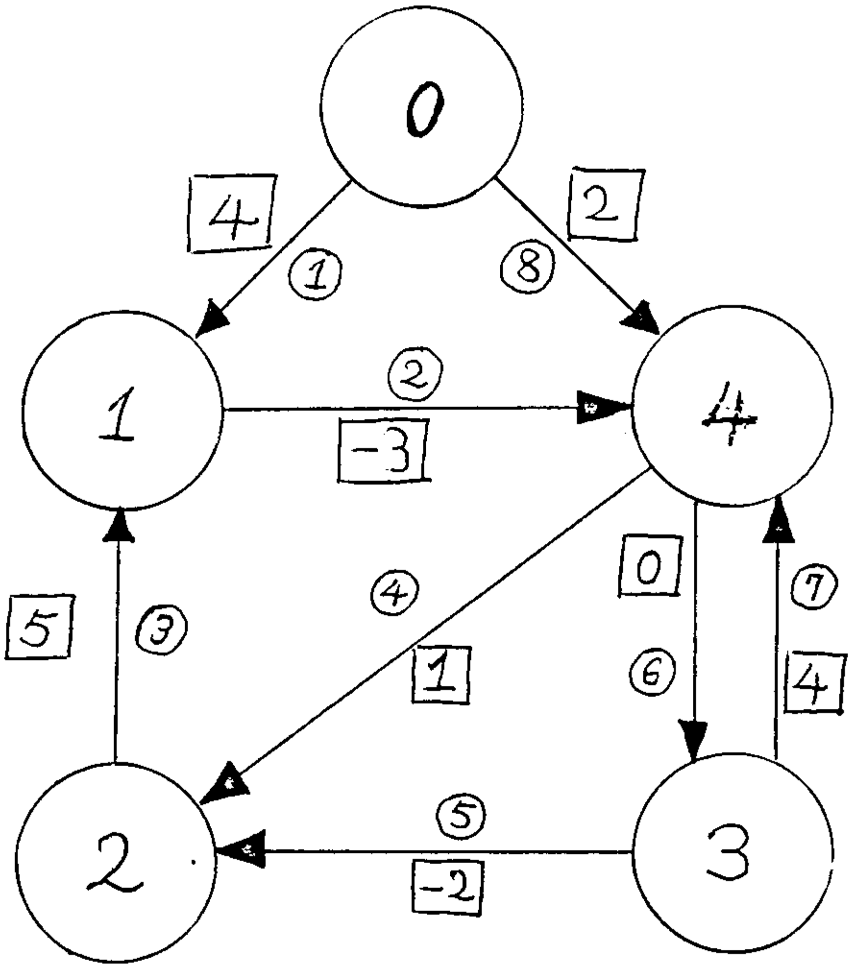


Figure 2.1.1.
An Example of (Single-Source) Shortest-Path Problem
and Its Solution by Relaxation.

$$(list(\eta_3),\delta_6,list(\eta_3)) \geq (list(\eta_4),\eta_5,\eta_5) \qquad (2^*)$$

$$(list(\eta_4),\eta_5,\eta_5) \geq (list(\eta_3),\delta_6,list(\eta_3)) \qquad (3)$$

We can repeat such a *relaxation step* again and again, until every inequality in the system is satisfied. The trace of one possible sequence of relaxation steps is listed below. The inequality chosen for relaxing at each step is marked by an asterisk (*).

Substitution:

$$\{ \eta_4/\eta_6, \eta_5/list(\eta_6) \}$$

Approximation 2:

$$(list(\eta_1),\delta_2,\delta_2) \geq (list(\eta_6),list(\eta_6),list(\eta_6)) \qquad (1)$$

$$(list(\eta_3),\delta_6,list(\eta_3)) \geq (list(\eta_6),list(\eta_6),list(\eta_6)) \qquad (2)$$

$$(list(\eta_6),list(\eta_6),list(\eta_6)) \geq (list(\eta_3),\delta_6,list(\eta_3)) \qquad (3^*)$$

Substitution:

$$\{ \eta_3/\eta_7, \delta_6/list(\eta_7) \}$$

Approximation 3:

$$(list(\eta_1),\delta_2,\delta_2) \geq (list(\eta_6),list(\eta_6),list(\eta_6)) \qquad (1)$$

$$(list(\eta_7),list(\eta_7),list(\eta_7)) \geq (list(\eta_6),list(\eta_6),list(\eta_6)) \qquad (2)$$

$$(list(\eta_6),list(\eta_6),list(\eta_6)) \geq (list(\eta_7),list(\eta_7),list(\eta_7)) \qquad (3)$$

At last, every inequality in the system is satisfied. The composition of all substitutions used in this sequence of relaxation steps is

$$\{ \gamma_1/list(\eta_6), \gamma_2/list(\eta_6), \gamma_3/list(\eta_6), \delta_6/list(\eta_7), \eta_3/\eta_7 \}.$$

■

2.3. Monadic Term Inequalities: Prefix Inequalities

Due to their complexity, it turns out to be very difficult to develop a formal theory for the relaxation processes we used to solve Term Inequalities. Yet, a formal theory is still highly desirable, for it can give us some insight into the underlying mechanisms of these relaxation processes. Therefore, we consider in this section a restricted version of Term Inequalities, which does permit a formal analysis.

The restriction which we impose on Term Inequalities is called the *Monadicity Restriction*: only monadic functors (i.e., functors of arity 1) are allowed in the terms of a system of Term Inequalities. It will be shown that the monadic version of Term Inequalities can be re-formulated as a problem concerning merely *strings*, instead of terms. This new problem is called *Prefix Inequalities*, which will be studied extensively in Chapter 3 and 4.

Effects of the Monadicity Restriction

Under the monadicity restriction every single Term Inequality is of the form: $(i, j \geq 0)$

$$f_1(f_2(...f_i(\alpha)...)) \geq g_1(g_2(...g_j(\beta)...)). \qquad (2.3.1)$$

It is easy to see that, by the definition of $\geq$, Ineq. (2.3.1) holds iff $f_1f_2...f_i$ (viewed as a string) is a *prefix* of $g_1g_2...g_j$ (also viewed as a string). Any instantiation of α will replace α by a term of this form: $(k \geq 0)$

$$h_1(h_2(...h_k(\gamma)...)),$$

which can also be viewed as a string $h_1h_2...h_k$. The substitution itself may be viewed as the *concatenation* of strings $f_1f_2...f_i$ and $h_1h_2...h_k$ (in that order). Similarly for instantiations of β. Therefore, (2.3.1) is equivalent to the following inequality on *strings*:

$$f_1f_2...f_i\,\alpha \geq g_1g_2...g_j\,\beta, \qquad (2.3.2)$$

(here $\geq$ is now read as "is a prefix of"; α and β now range over strings), in the sense that the solution of either one can be transformed into the solution of the other in an obvious way. Ineq. (2.3.2) is called a Prefix Inequality, since the relation "is a prefix of" is used to construct it.

The above argument for replacing a monadic Term Inequality by a Prefix Inequality applies equally well to the case in which there are many inequalities coupled together which need to be solved simultaneously. The reason is that, with only monadic functors allowed, we never need to identify (i.e., unify) two different terms as might be needed for the general case. For example, if we want to solve the inequality

$$f(\alpha, \alpha) \geq f(\beta, g(\gamma)),$$

it is necessary to unify terms β and $g(\gamma)$ by applying a substitution like:

$$\{\beta / g(\delta), \gamma / \delta\}.$$

Obviously, this kind of situation never happens when there are only monadic functors. Hence, under the monadic restriction, when a variable α is substituted for by a term $h_1(h_2(...h_k(\beta)...))$, the actual name of the variable β is of no importance, as long as it is different from all the other variables. The only thing that matters is the string $h_1h_2...h_k$. Therefore, we will re-formulate the problem of monadic Term Inequalities into a problem on strings -- Prefix Inequalities, because the latter is mathematically and notationally much easier to deal with than the former.

Formal Definition of Prefix Inequalities

To make the term *Prefix Inequalities* more precise, some concepts and notations need to be introduced or made more precise. This is done in the following paragraphs.

Let Σ be an *alphabet* of *symbols*, Σ^* the set of all (finite) *strings* over Σ, including the empty string ε. We will use Latin letters such as $a, b, c, \ldots$ for symbols and Greek letters such as $\lambda, \rho, \sigma, \tau, \ldots$ for strings. The *concatenation* of two strings σ and τ is defined in the usual way, denoted by $\sigma\tau$. For any string ρ, let $|\rho| \in \mathbf{N}$ denote the *length* of ρ, which is the number of symbols in ρ (including repetitions). In particular, $|\varepsilon| = 0$.

Definition 2.3.1:[1] For any two strings $\sigma, \tau \in \Sigma^*$, we say that σ is a *prefix* of τ, denoted by $\sigma \geq \tau$ or $\tau \leq \sigma$, iff there exists a $\rho \in \Sigma^*$ such that $\tau = \sigma\rho$.

Lemma 2.3.2: $\geq$ partially orders Σ^*.

Proof: It suffices to show that $\geq$ has the following three properties:

(a) *Reflexivity*. This is so since $\sigma = \sigma\varepsilon$ for any $\sigma \in \Sigma^*$.

(b) *Anti-symmetry*. Suppose $\sigma \geq \tau$ and $\tau \geq \sigma$. Then there are ρ, ρ' such that

$$\tau = \sigma\rho, \quad \sigma = \tau\rho'.$$

This implies that

$$\tau\varepsilon = \tau = \sigma\rho = (\tau\rho')\rho = \tau(\rho'\rho),$$

which in turn implies that $\rho'\rho = \varepsilon$. But this holds only if $\rho' = \rho = \varepsilon$.

(c) *Transitivity*. This can be easily verified using the associativity of the concatenation operation. ■

A property of the partial order $\geq$ which has important implications in Chapter 3 is proved below.

Lemma 2.3.3: Let $\{\sigma_k\}_{k \in \mathbf{N}}$ be a decreasing sequence of strings with a lower bound τ under ordering $\geq$, that is,

$$\sigma_0 \geq \sigma_1 \geq \sigma_2 \geq \ldots \geq \tau.$$

Then there must exist a $k_0 \in \mathbf{N}$ such that $\sigma_k = \sigma_{k_0}$ for all $k \geq k_0$.

Proof: If otherwise, then $\{\, |\sigma_k| \,\}_{k \in \mathbf{N}}$ is an unbounded sequence of natural numbers, contradicting the hypothesis that $|\tau| \in \mathbf{N}$ is an upper bound of that sequence. ■

Briefly speaking, an instance of PI is just a finite collection of inequalities of this form:

$$\lambda x \geq \rho y,$$

where λ and ρ are given strings, x and y are string variables whose values are to be

[1]The symbols $\geq$ and $\leq$ are used for many different purposes in this Thesis. The reader should, however, always be able to see the desired interpretation from the context.

found. More formally, we have:

Definition 2.3.4: An instance of *Prefix Inequalities* (PI) is a triple (Σ, $\mathbf{G}$, $\mathbf{I}$), where

(a) Σ is an alphabet;

(b) $\mathbf{G} = (\mathbf{V}, \mathbf{A})$ is a di-graph with node set $\mathbf{V} = \{ 1, 2, ..., n \}$ and arc set $\mathbf{A} = \{ 1, 2, ..., m \}$, for some $n, m \geq 1$;

(c) $\mathbf{I}$ is an assignment which assigns to each arc $j \in \mathbf{A}$ a prefix inequality

$$\mathbf{I}_j : \lambda_j x_{from(j)} \geq \rho_j x_{to(j)}, \tag{2.3.3}$$

where $\lambda_j, \rho_j \in \Sigma^*$ are given strings, $\{x_1, x_2, ..., x_n\}$ is a set of variables ranging over Σ^*.

The problem is to find the *greatest* n-tuple of strings (under the partial order $\geq$) which satisfies each $\mathbf{I}_j$, $1 \leq j \leq m$.

Observe the similarities between PI and LI-formulated SPP. Indeed, both are special cases of Relaxation Problems defined in Chapter 3. More striking similarities will be seen in Chapter 4.

CHAPTER 3.
A Formal Theory of Relaxation Processes

In this chapter a formal theory of relaxation processes is developed. We begin with the definition of *Relaxation Problems*, which is a formal characterization of problems subject to solution by relaxation techniques. The Shortest-Path Problem and Prefix Inequalities are shown to be special cases of Relaxation Problems. Then we formalize the relaxation procedures which have been exemplified in Chapter 2 into the so-called *General Relaxation Semi-Algorithm*, which can be used to solve Relaxation Problems. Some important properties of this semi-algorithm are proved, which give us some insight into, and also (partially) justify our definition of, Relaxation Problems. Finally we specialize the General Relaxation Semi-Algorithm to solve the Shortest-Path Problem and Prefix Inequalities, and study what consequences the general properties of the semi-algorithm have on these two special cases. In particular, the *only-if* part of Theorem 2.1.3 is proved.

Not all questions are settled down by the formal theory of this Chapter. The most prominent one is the possible non-termination of the General Relaxation Semi-Algorithm. We will study, and *solve*, this problem with respect to SPP and PI in the next Chapter.

3.1. Relaxation Problems

Several problems which are solvable by relaxation techniques have been presented in Chapter 2. The reader, after examining those problems and their solutions through relaxation processes, must have sensed some striking similarities among them. In this section we will attempt a formal characterization of this kind of problems, which will be termed *Relaxation Problems*. The justification of our formal definition, however, does not reside in the definition itself, but in its implications, as will be seen later.

To give a formal definition of Relaxation Problems, some notions are needed.

Definition 3.1.1: A poset $(\mathbf{P}, \geq)$ has the *Bounded Descending Chain* (BDC) property iff, whenever $\{\mathbf{p}_k\}_{k \in \mathbf{N}}$ is an *infinite* descending chain in $\mathbf{P}$ with a lower bound $\mathbf{q} \in \mathbf{P}$, that is,

$$\mathbf{p}_0 \geq \mathbf{p}_1 \geq \mathbf{p}_2 \geq \ldots \geq \mathbf{q},$$

then there exists a $k_0 \in \mathbf{N}$ such that $\mathbf{p}_k = \mathbf{p}_{k_0}$ for all $k \geq k_0$.

Definition 3.1.2: Let $(\mathbf{P}, \geq)$ be a poset. A set $\mathbf{C} \subseteq \mathbf{P}$ is called a *relaxable* subset of $\mathbf{P}$ iff for any $\mathbf{p} \in \mathbf{P}$, the set

$$\mathbf{C} \downarrow \mathbf{p} = \{\mathbf{q} \in \mathbf{C} \mid \mathbf{q} \leq \mathbf{p}\} \tag{3.1.1}$$

either has a *greatest* element, or is empty.

Definition 3.1.2a: Whenever $\mathbf{C}$ is relaxable and $\mathbf{C} \downarrow \mathbf{p} \neq \varnothing$, let

$$\text{RELAX}(\mathbf{C}, \mathbf{p}) = \max \{ \mathbf{C} \downarrow \mathbf{p} \}.$$

Moreover, whenever RELAX(C, p) is used, it is implied that $\mathbf{C} \downarrow \mathbf{p} \neq \varnothing$.

Notice that $\mathbf{p} \in \mathbf{C}$ iff RELAX(C, p) = p. Now we are ready to give the formal definition of Relaxation Problems.

Definition 3.1.3: A *Relaxation Problem* (RP) consis[illegible] a triple $(\mathbf{P}, \Xi, \mathbf{p}_0)$, where

(a) $\mathbf{P} = (\mathbf{P}, \geq)$ is a poset with the Bounded Descendi[illegible]ain property;

(b) $\Xi = \{\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_m\}$ is a finite collection of *rel[illegible]e* subsets of $\mathbf{P}$;

(c) $\mathbf{p}_0$ is an arbitrary element of $\mathbf{P}$.

The task is to find the *greatest* element of $\mathbf{P}$ which is $\leq \mathbf{p}_0$ and belongs to each $\mathbf{C}_j$ $(1 \leq j \leq m)$. In other words, we want to find the greatest element of the set

$$\Xi \downarrow \mathbf{p}_0 = \left[\bigcap_{j=1}^{m} \mathbf{C}_j \right] \downarrow \mathbf{p}_0 . \tag{3.1.2}$$

$\Xi \downarrow \mathbf{p}_0$ is, of course, an abuse of notation.

Intuitively, the subsets $\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_m$ represent the *constraints* that we want to satisfy. For a constraint $\mathbf{C}_j$ and a point $\mathbf{p} \in \mathbf{P}$, we say that $\mathbf{p}$ *satisfies* $\mathbf{C}_j$ or $\mathbf{C}_j$ is *in-kilter* at $\mathbf{p}$, if $\mathbf{p} \in \mathbf{C}_j$, otherwise we say that $\mathbf{p}$ *violates* $\mathbf{C}_j$ or $\mathbf{C}_j$ is *out-of-kilter* at $\mathbf{p}$. Thus, the task of an RP is to find the greatest element which satisfies all constraints and $\leq$ some given element. Moreover, we call RELAX($\mathbf{C}_j$, p) the *relaxation* of constraint $\mathbf{C}_j$ at point $\mathbf{p}$. This terminology stems from the observation that, if $\mathbf{p}$ violates $\mathbf{C}_j$, RELAX($\mathbf{C}_j$, p) is in some sense the minimal adjustment to $\mathbf{p}$ that is necessary to *relax* the out-of-kilter constraint $\mathbf{C}_j$ (that is, bring $\mathbf{C}_j$ back to in-kilter state).

3.2. Examples of Relaxation Problems

In this section we show that our definition of Relaxation Problems is general enough to encompass (LI-formulated) SPP and PI. Before digging into the details of individual problems, we would like to state (without proof) some simple properties of products of posets, which will be very useful in the subsequent discussion.

Let $(P_1, \geq_1), (P_2, \geq_2), \ldots, (P_n, \geq_n)$ be posets, $(\mathbf{P}, \geq)$ their Cartesian product with the natural order. It turns out that both the Bounded Descending Chain property and the relaxability of subsets are inheritable from the component posets to the product poset. More precisely, we have:

Lemma 3.2.1: Poset $\mathbf{P}$ has the BDC property iff each P_i, $1 \leq i \leq n$, has the BDC property.

Lemma 3.2.2: Let $1 \leq i_1 \neq i_2 \leq n$. Suppose $\mathbf{C}$ is a subset of $\mathbf{P}$ defined by

$$\mathbf{C} = \{ (p_1, \ldots, p_n) \in \mathbf{P} \mid (p_{i_1}, p_{i_2}) \in D \},$$

where D is a *relaxable* subset of $P_{i_1} \times P_{i_2}$. Then $\mathbf{C}$ is a relaxable subset of $\mathbf{P}$. Moreover, we have:

$$\text{RELAX}(\mathbf{C}, (p_1, p_2, \ldots, p_n)) = (q_1, q_2, \ldots, q_n)$$

if, and only if,

RELAX$(D, (p_{i_1}, p_{i_2})) = (q_{i_1},$ [illegible] nd $q_i = p_i$ for $i \neq i_1, i_2$, $1 \leq i \leq n$.

Now we are ready to show [illegible] oth (LI-formulated) SPP and PI are special cases of Relaxation Problems.

(LI-Formulated) Shortest-Path Problem

Recall that an LI-formulated SPP is the problem of finding the greatest tuple of extended integers satisfying the Linear Inequalities associated with a (traditional) instance of SPP. Using the same notation as in Section 2.1, let $(\mathbf{G}, L)$ be an instance of SPP, where $\mathbf{G} = (\mathbf{V}, \mathbf{A})$ is a di-graph *without self-loops*, $\mathbf{V} = \{0, 1, 2, \ldots, n\}$ is the set of nodes, $\mathbf{A} = \{1, 2, \ldots, m\}$ is the set of arcs, node 0 is the source node, and $L : \mathbf{A} \rightarrow \mathbf{Z}$ is the length-labelling mapping. Its associated Linear Inequalities are

$$\left.\begin{array}{c} \mathbf{I}_j : x_{from(j)} + L_j \geq x_{to(j)}, \quad 1 \leq j \leq m \\ \\ 0 \geq x_0 \end{array}\right\} \qquad (3.2.1)$$

To show that the LI-formulated SPP corresponding to the above system of LI does specify a Relaxation Problem, it suffices to identify the three components of an RP. With the same notation as in Definition 3.1.3, this is done in the following:

(a) Let $\mathbf{P} = \overline{\mathbf{Z}}^{n+1}$, the $(n+1)$-st power of the poset $\overline{\mathbf{Z}}$ of extended integers, partially ordered by the component-wise generalization of the natural (total) order of extended integers. Since $\overline{\mathbf{Z}}$ has the BDC property, it follows from Lemma 3.2.1 that $\mathbf{P}$ also has the BDC property.

(b) Let $\Xi = \{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_m\}$, where $\mathbf{C}_j$ $(1 \leq j \leq m)$ is the constraint subset associated with the inequality $\mathbf{I}_j$ in (3.2.1) and defined by

$$\mathbf{C}_j = \{ (d_0, d_1, \ldots, d_n) \in \mathbf{P} \mid d_{from(j)} + L_j \geq d_{to(j)} \}. \tag{3.2.2}$$

The relaxability of $\mathbf{C}_j$ will be proved soon (in Lemma 3.2.3).

(c) $\mathbf{p}_0 = (0, \infty, \infty, \ldots, \infty) \in \mathbf{P}$.

Lemma 3.2.3: Each $\mathbf{C}_j$ $(1 \leq j \leq m)$ is a relaxable subset of $\mathbf{P}$.

Proof: By Lemma 3.2.2, to show that $\mathbf{C}_j$ is a relaxable subset of $\mathbf{P}$, it is sufficient to show that the set

$$D = \{ (x, y) \in \bar{\mathbf{Z}} \times \bar{\mathbf{Z}} \mid x + L_j \geq y \}$$

is a relaxable subset of $\bar{\mathbf{Z}} \times \bar{\mathbf{Z}}$. Note that since there is *no* self-loop in graph $\mathbf{G}$, it is always true that $from(j) \neq to(j)$. This is why we need only to consider the 2-dimensional set $\bar{\mathbf{Z}} \times \bar{\mathbf{Z}}$. Given any $(a, b) \in \bar{\mathbf{Z}} \times \bar{\mathbf{Z}}$, consider the set

$$E = D \downarrow (a, b) = \{ (x, y) \in \bar{\mathbf{Z}} \times \bar{\mathbf{Z}} \mid x + L_j \geq y ,\ x \leq a ,\ y \leq b \}.$$

There are two cases:

(i) If $a + L_j \geq b$, then (a, b) is clearly the greatest element of E.

(ii) If $a + L_j < b$, for any $(x, y) \in E$, we have $(x, y) \leq (x, x + L_j) \leq (a, a + L_j)$, so $(a, a + L_j)$ is the greatest element of E.

So, D is a relaxable subset of $\bar{\mathbf{Z}} \times \bar{\mathbf{Z}}$ and

$$\text{RELAX}(D, (a, b)) = \begin{cases} (a, b) & \text{if } a + L_j \geq b \\ (a, a + L_j) & \text{if } a + L_j < b \end{cases} \tag{3.2.3}$$

■

Prefix Inequalities

Using the same notation as in Section 2.3, let us consider an instance of PI, $(\Sigma, \mathbf{G}, \mathbf{I})$, where Σ is an alphabet, $\mathbf{G} = (\mathbf{V}, \mathbf{A})$ is a di-graph without *self-loops*, $\mathbf{V} = \{1, 2, \ldots, n\}$ is the set of nodes, $\mathbf{A} = \{1, 2, \ldots, m\}$ is the set of arcs, and $\mathbf{I}$ assigns to each $j \in \mathbf{A}$ a prefix inequality

$$\mathbf{I}_j : \lambda_j x_{from(j)} \geq \rho_j x_{to(j)}, \tag{3.2.4}$$

with given $\lambda_j, \rho_j \in \Sigma^*$. The problem is to find the *greatest* n-tuple of strings under the partial order $\geq$ ("*is a prefix of*") which satisfies each $\mathbf{I}_j$, $1 \leq j \leq m$. To show that this instance of PI also specifies an RP, we need to identify the three components of a Relaxation Problem, as required by Definition 3.1.3. This is done in the following:

(a) Let $\mathbf{P} = (\Sigma^*)^n$, the n-th power of the poset Σ^*, partially ordered by the component-wise generalization of the partial order "*is a prefix of*" on Σ^*. Since it has been shown in Lemma 2.3.3 that Σ^* has the BDC property, it follows from Lemma 3.2.1 that $\mathbf{P}$ also has BDC property.

(b) Let $\Xi = \{\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_m\}$, where $\mathbf{C}_j$ $(1 \le j \le m)$ is the constraint subset associated with prefix inequality $\mathbf{I}_j$ (3.2.4) and defined by

$$\mathbf{C}_j = \{ (\tau_1, ..., \tau_n) \in \mathbf{P} \mid \lambda_j \tau_{from(j)} \ge \rho_j \tau_{to(j)} \}. \tag{3.2.5}$$

The relaxability of $\mathbf{C}_j$ will be proved soon (in Lemma 3.2.5).

(c) $\mathbf{p}_0 = (\varepsilon, \varepsilon, ..., \varepsilon) \in \mathbf{P}$. Clearly $\mathbf{p}_0 \ge \mathbf{p}$ for any $\mathbf{p} \in \mathbf{P}$.

To prove the relaxability of $\mathbf{C}_j$'s, we need the following notation.

Definition 3.2.4: Let $\sigma, \tau \in \Sigma^*$. Suppose $\sigma \ge \tau$. We define $\sigma^{-1}\tau$ to be the unique string $\rho \in \Sigma^*$ such that $\sigma\rho = \tau$.

Intuitively, $\sigma^{-1}\tau$ is obtained from τ by "cutting away" the prefix σ. Note that $\sigma^{-1}\tau$ is *undefined* if $\sigma \ge \tau$ does not hold.

Lemma 3.2.5: Each $\mathbf{C}_j$ $(1 \le j \le m)$ is a relaxable subset of $\mathbf{P}$.

Proof: By Lemma 3.2.2, to show that $\mathbf{C}_j$ is a relaxable subset of $\mathbf{P}$, it is sufficient to show that the set

$$D = \{ (x, y) \in \Sigma^* \times \Sigma^* \mid \lambda_j x \ge \rho_j y \}$$

is a relaxable subset of $\Sigma^* \times \Sigma^*$. Given any $(\sigma, \tau) \in \Sigma^* \times \Sigma^*$, consider the set

$$E = D \downarrow (\sigma, \tau) = \{ (x, y) \in \Sigma^* \times \Sigma^* \mid \lambda_j x \ge \rho_j y,\ x \le \sigma, y \le \tau \}.$$

There are three possible cases:

(i) If $\lambda_j \sigma \ge \rho_j \tau$, the greatest element of E is clearly (σ, τ).

(ii) If $\lambda_j \sigma < \rho_j \tau$, then for any $(x, y) \in E$, $\lambda_j x \le \lambda_j \sigma < \rho_j \tau \le \rho_j$, so both $\rho_j^{-1}(\lambda_j x)$ and $\rho_j^{-1}(\lambda_j \sigma)$ are well-defined. Further, we have

$$(x, y) \le (x, \rho_j^{-1}(\lambda_j x)) \le (\sigma, \rho_j^{-1}(\lambda_j \sigma)).$$

Since $(\sigma, \rho_j^{-1}(\lambda_j \sigma)) \in E$, it is the greatest element of E.

(iii) If $\lambda_j \sigma$ and $\rho_j \tau$ are *incomparable*, then we note that for any $\sigma' \le \sigma$ and $\tau' \le \tau$, $\lambda_j \sigma'$ and $\rho_j \tau'$ are still incomparable. So $E = \varnothing$.

Therefore, D is a relaxable subset of $\Sigma^* \times \Sigma^*$ and

$$\text{RELAX}(D, (\sigma, \tau)) = \begin{cases} (\sigma, \tau) & \text{if } \lambda_j \sigma \ge \rho_j \tau \\ (\sigma, \rho_j^{-1}(\lambda_j \sigma)) & \text{if } \lambda_j \sigma < \rho_j \tau \\ \text{undefined} & \text{if } \lambda_j \sigma \text{ and } \rho_j \tau \text{ are incomparable} \end{cases} \tag{3.2.6}$$

■

3.3. General Relaxation Semi-Algorithm

In the last section it is shown that both SPP and PI fit into our definition of Relaxation Problems. We will, in this section, formalize the relaxation techniques which are used in Examples 2.1.1 and 2.3.1 to solve SPP and PI, respectively. The result is a semi-algorithm, the *General Relaxation Semi-Algorithm* (GRSA), for solving Relaxation Problems. It is called a *semi*-algorithm because it may never halt on some problem instances. Then we will prove some important properties of GRSA, which justify our claim about the capability of GRSA and also shed some light on the properties of Relaxation Problems themselves.

As having been pointed out in Chapter 1, a typical relaxation process consists of the selection of *initial approximation* to the solution and the (possibly infinite) iteration through *relaxation steps*, which are *local* adjustments to the *intermediate approximations* in attempt to *satisfy* some *constraints*. The goal, that is, the terminating condition, of the iteration is the satisfaction of *all* constraints. Put formally and supplemented with error-handling code, we get the semi-algorithm GRSA shown in Figure 3.3.1,[1] which is supposed to be applied to a Relaxation Problem $(\mathbf{P}, \Xi, \mathbf{p}_0)$ as the one in Definition 3.1.3.

Note that the **if** test at Line (4) of GRSA is both necessary and sufficient for the applicability of the Relaxation Step at Line (5), since each $\mathbf{C}_j$ is assumed to be a *relaxable* subset of **P**.

Properties of GRSA

Given a Relaxation Problem $(\mathbf{P}, \Xi, \mathbf{p}_0)$ as the one in Definition 3.1.3, let us consider the execution of its corresponding GRSA. Suppose the successive values assumed by the variable *Approx* when the control is at Line (2) of GRSA are $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k, \ldots$.. This sequence may or may not be finite, depending on whether or not the semi-algorithm terminates (either through Line (7) or through the **halt** statement). In either case, the following two propositions are true.

Lemma 3.3.1: Each $\mathbf{p}_k$ is an upper bound of the set $\Xi \downarrow \mathbf{p}_0$.

Proof: First note that this lemma is vacuously true when $\Xi \downarrow \mathbf{p} = \varnothing$. So suppose $\Xi \downarrow \mathbf{p} \neq \varnothing$. The proof is by induction on k. Note that the argument is valid whether $\{\mathbf{p}_k\}$ is finite or not. The induction base is simple: p_0 is an upper bound of $\Xi \downarrow \mathbf{p}_0$ simply by definition. For the induction step, suppose that p_k is an upper bound of $\Xi \downarrow \mathbf{p}_0$. Assume that

$$\mathbf{p}_{k+1} = \mathrm{RELAX}(\mathbf{C}_{j_0}, \mathbf{p}_k)$$

[1]All (semi-)algorithms in this Thesis are expressed using an Algol-like syntax. Their semantics should be self-explanatory in most cases. Comments (/*....*/) are added for clarification in spots.

```
/* Approx is a variable whose values are Approximations to the solution. */
var Approx : P;
begin
(1)  Approx ← $p_0$;
(2)  while $\neg(\forall j\,(1 \le j \le m)$ Approx $\in C_j)$ do
(3)      Select arbitrarily a $C_j$, $1 \le j \le m$, such that Approx $\notin C_j$;
(4)      if $C_j \downarrow$ Approx $\neq \varnothing$ then
(5)          Approx ← RELAX($C_j$, Approx)     /* Relaxation Step */
(6)      else
             print("Error: $\Xi \downarrow p_0 = \varnothing$");
             halt
         end if
     end do;
(7)  print("The greatest element of $\Xi \downarrow p_0$ is ", Approx);
end.
```

Figure 3.3.1. General Relaxation Semi-Algorithm (GRSA).

$$= \max\{\, C_{j_0} \downarrow p_k \,\},\ \text{for some } C_{j_0},\ 1 \le j_0 \le m. \qquad (3.3.1)$$

Notice that $\Xi \downarrow p_0 = \left[\bigcap_{j=1}^{m} C_j\right] \downarrow p_0 \subseteq C_{j_0}$. By the induction hypothesis, $\Xi \downarrow p_0 \subseteq C_{j_0} \downarrow p_k$. It follows that p_{k+1} is an upper bound of $\Xi \downarrow p_0$. This completes the induction. ■

Lemma 3.3.2: $\{p_k\}$ is a *strictly* decreasing sequence.

Proof: Note that any two consecutive terms p_k and p_{k+1} in sequence $\{p_k\}$ are related by an equation like Eq. (3.3.1). Hence $p_k \ge p_{k+1}$. Also note that, by the selection at Line (3) of GRSA, constraint C_{j_0} is chosen for relaxing at Line (5) because $\mathbf{p}_k \notin \mathbf{C}_{j_0}$. But $\mathbf{p}_{k+1} \in \mathbf{C}_{j_0}$. So $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ can not be equal. That is, $\mathbf{p}_k > \mathbf{p}_{k+1}$. ■

There are three possible outcomes of the execution of GRSA: (a) it exits through Line (7), (b) it halts by taking the **else** branch at Line (6), or (c) it just runs forever, never stops. If the outcome is (a), then we say that the GRSA *exits successfully.* Below is the main result of this section.

Theorem 3.3.3: Given a Relaxation Problem $(\mathbf{P}, \Xi, \mathbf{p}_0)$, $\Xi \downarrow \mathbf{p}_0 \neq \varnothing$ if, and only if, the corresponding GRSA exits successfully. Moreover, the final value of variable *Approx* when the semi-algorithm exits successfully is exactly the greatest element of $\Xi \downarrow \mathbf{p}_0$.

Proof: Suppose that the semi-algorithm exits successfully with the final value of *Approx* equal to $\mathbf{p}_{k_0}$ for some $k_0 \in \mathbf{N}$. Due to the test at the start of the **while**-loop (Line (2)), $\mathbf{p}_{k_0}$ satisfies all constraints. Besides, $\mathbf{p}_{k_0} \leq \mathbf{p}_0$, since sequence $\{\mathbf{p}_k\}$ is decreasing by Lemma 3.3.2. So, $\mathbf{p}_{k_0} \in \Xi \downarrow \mathbf{p}_0$. But $\mathbf{p}_{k_0}$ is an upper bound of $\Xi \downarrow \mathbf{p}_0$ by Lemma 3.3.1. It follows that $\mathbf{p}_{k_0}$ is, as desired, the greatest element of $\Xi \downarrow \mathbf{p}_0$. This completes the proof of both the *if* and the *moreover* parts.

For the *only-if* part, suppose that $\Xi \downarrow \mathbf{p}_0 \neq \varnothing$, containing (at least) an element $\mathbf{q}$. Note that, by Lemmas 3.3.1 and 3.3.2,

$$\mathbf{p}_0 \geq \mathbf{p}_1 \geq \mathbf{p}_2 \geq \ldots \geq \mathbf{q}.$$

So the semi-algorithm can never halt prematurely by taking the **else** branch of the **if** test at Line (4), since at least $\mathbf{q} \in \mathbf{C}_j \downarrow \mathbf{p}_k$ for any j and k. Now assume that the semi-algorithm never terminates. Then sequence $\{\mathbf{p}_k\}$ is an *infinite* descending chain with a lower bound $\mathbf{q}$. Since $\mathbf{P}$ has the Bounded Descending Chain property, we conclude that there exists a $k_0 \in \mathbf{N}$ such that $\mathbf{p}_k = \mathbf{p}_{k_0}$ for all $k \geq k_0$. But this contradicts the *strict* decreasingness of $\{\mathbf{p}_k\}$ shown in Lemma 3.3.2. Therefore, the semi-algorithm must exit successfully. This completes the proof. ■

Corollary 3.3.4: $\Xi \downarrow \mathbf{p}_0$ always has a *greatest* element as long as it is not empty. (In other words, the property of *relaxability* is closed under set intersections.)

Proof: Just run the General Relaxation Semi-Algorithm, which, by the above Theorem, will give the greatest element of $\Xi \downarrow \mathbf{p}_0$ whenever it is not empty. ■

Notice that the selection of violated constraints at Line (3) of GRSA is entirely *arbitrary*. Theorem 3.3.3 guarantees that, if there is any solution at all, then you will get it no matter how you select constraints to relax. So we may say that GRSA is *nondeterministic*. However, GRSA has a serious weakness: it may never halt when there is no solution. Remedies will be presented in Chapter 4 with respect to SPP and PI. There, it turns out that to detect the potential non-termination of GRSA, violated contraints can *not* be selected to relax in an entirely arbitrary way.

3.4. Applications to SPP and PI

In this section we will apply the results of Section 3.3 to (LI-formulated) SPP and PI, both of which have been shown in Section 3.2 to be special cases of Relaxation Problems. We will first specialize GRSA to solve the individual problem, and then see what consequences this new approach has.

(LI-Formulated) Shortest-Path Problem

Since (LI-formulated) SPP is a special case of RP (Section 3.2), GRSA can (of course) be used to solve it. The specialized version of GRSA is called RSA-SPP (for *Relaxation Semi-Algorithm - SPP*) and shown in Figure 3.4.1.[1] Notice that the Relaxation Step at Line (4) of RSA-SPP uses Eq. (3.2.3). Also note that the error-handling code in GRSA is unnecessary here, since any single linear inequality is always relaxable.

```
        var $x$ : array [0 .. $n$] of $\bar{\bar{Z}}$;   /* The counterpart of Approx in GRSA */
        begin
(1)       $x \leftarrow (0, \infty, \infty, \ldots, \infty)$;
(2)       while $\neg(\forall j (1 \le j \le m)\ x_{from(j)} + L_j \ge x_{to(j)})$ do
(3)          Select arbitrarily a $j$, $1 \le j \le m$, such that $x_{from(j)} + L_j < x_{to(j)}$;
(4)          $x_{to(j)} \leftarrow x_{from(j)} + L_j$      /* Relaxation Step */
          end do;
(5)       print("The solutions are: ", $x_0, x_1, \ldots, x_n$);
        end.
```

Figure 3.4.1.
RSA-SPP: Relaxation Semi-Algorithm for Solving LI-formulated SPP.

The reader can verify that both sequences of relaxation steps shown in Figure 2.1.1 are obtainable by making different selections at Line (3) of

RSA-SPP when it is applied to the instance of SPP in Example 2.1.1.

[1]The notation for arrays in Figure 3.4.1 (and all later figures) is somewhat non-standard. Instead of using $x[i]$, we use x_i for the i-th component of array x.

Now we are able to prove the *only-if* part of Theorem 2.1.3 in the lemma below.

Lemma 3.4.1: Given an instance of SPP and its associated LI, if the shortest distance to each node in the di-graph is well-defined, then there is a greatest tuple of extended integers satisfying every inequality in the associated LI, and RSA-SPP will exit successfully when applied to the associated LI.

Proof: Just note that the tuple of the shortest distances to all nodes in the di-graph always satisfies the associated LI. By Corollary 3.3.4, there is a *greatest* tuple satisfying the associated LI. Moreover, by Theorem 3.3.3, RSA-SPP must exit successfully, because it is just a specialized version of GRSA. ■

Prefix Inequalities

Similarly, we can specialize GRSA to solve PI. What we get is called RSA-PI, shown in Figure 3.4.2. Notice that Lines (4) - (6) of RSA-PI use Eq. (3.2.6).

Theorem 3.4.2: Given any instance of PI with n variables, there is always a *greatest* n-tuple of strings which satisfies every inequality in this instance, as long as there is any solution at all. Moreover, RSA-PI will exit successfully with the greatest solution when applied to this instance.

Proof: Since PI is a special case of Relaxation Problems (Section 3.2), this theorem is just a specialized version of Theorem 3.3.3 and Corollary 3.3.4. ■

```
    var $x$ : array [1 .. $n$] of $\Sigma^*$;   /* The counterpart of Approx in GRSA */
    begin
(1)   $x \leftarrow (\varepsilon, \varepsilon, \ldots, \varepsilon)$;
(2)   while $\neg(\forall j\,(1 \le j \le m)\ \lambda_j x_{from(j)} \ge \rho_j x_{to(j)})$ do
(3)      Select arbitrarily a $j$, $1 \le j \le m$,
              such that $\lambda_j x_{from(j)} \ge \rho_j x_{to(j)}$ does not hold;
(4)      if $\lambda_j x_{from(j)} < \rho_j x_{to(j)}$ then
(5)         $x_{to(j)} \leftarrow \rho_j^{-1}(\lambda_j x_{from(j)})$    /* Relaxation Step */
(6)      else
           print("Error: No solution at all!");
           halt
       end if
    end do;
(7)   print("The solutions are: ", $x_1, \ldots, x_n$);
    end.
```

Figure 3.4.2. RSA-PI: Relaxation Semi-Algorithm for Solving PI.

CHAPTER 4.
Two Case Studies on Relaxation Processes

In the last chapter we described a General Relaxation Semi-Algorithm (GRSA), which can be employed to solve general Relaxation Problems. However, GRSA is far from practical, since it may run *forever* on some input without giving any answer or even warning. Some of this kind of "pathological" situations are presented in Section 1 of this chapter.

In an attempt to improve GRSA, the author decide to concentrate on two special cases of Relaxation Problems: (LI-formulated) SPP and PI. In Section 2 we first prove some interesting and very useful properties of LI-formulated SPP, which are concerned with spanning trees of shortest paths. With these results we are able to modify GRSA to obtain a *polynomial-time algorithm* for solving LI-formulated SPP. As a by-product, we are also able to finish the proof of Theorem 2.1.3.

Analogous results for PI are obtained in Section 3, using arguments almost identical to those in Section 2. Then these results are used to transform GRSA into a *polynomial-time algorithm* for solving PI. Moreover, the reader will observe the striking similarities between LI-formulated SPP and PI.

4.1. Possible Non-Termination of GRSA

The General Relaxation Semi-Algorithm (Figure 3.3.1) and its variations RSA-SPP (Figure 3.4.1) and RSA-PI (Figure 3.4.2) are only *semi*-algorithms: they may never terminate on some inputs. Although it has been shown in Theorem 3.3.3 that, whenever the set $\Xi \downarrow \mathbf{p}_0$ of feasible solutions is non-empty, GRSA must terminate in finitely many steps with the correct answer (i.e., the greatest element of $\Xi \downarrow \mathbf{p}_0$), nothing was said about what might happen if $\Xi \downarrow \mathbf{p}_0 = \emptyset$. It turns out that GRSA may indeed run forever in some of this kind of situations, as the following two examples will show.

Example 4.1.1: Consider the Shortest-Path Problem shown in Figure 4.1.1, which is drawn according to the same notational conventions as Figure 2.1.1.

The Linear Inequalities associated with this SPP are given below:

$$0 \geq x_0$$

$$x_0 + 1 \geq x_1 \qquad \text{(arc 1)}$$

$$x_1 + (-3) \geq x_2 \qquad \text{(arc 2)}$$

$$x_2 + 1 \geq x_1 \qquad \text{(arc 3)}$$

Now let us run RSA-SPP on these Linear Inequalities. An initial segment of the execution trace is also shown in Figure 4.1.1. Though RSA-SPP is nondeterministic, this particular instance of SPP allows only one single sequence of relaxation steps, in contrast to the many in Example 2.1.1. It is easy to see that, after the first relaxation step (relaxing (arc 1)), (arc 2) and (arc 3) are alternately violated, resulting in a relaxation sequence: (arc 2), (arc 3), (arc 2), (arc 3), (arc 2), ... ad infinitum!

The trouble results from the cycle consisting of nodes 1 and 2: it is of a *negative* length (-2). Summing up inequalities (arc 2) and (arc 3) and cancelling the unknowns x_1 and x_2, we get

$$-2 \geq 0,$$

an obvious contradiction! This means that the LI have *no* solution at all. So the non-termination of RSA-SPP does *not* conflict with Theorem 3.3.3. ■

Example 4.1.2: For another example of the possible non-termination of GRSA, we turn our attention to Prefix Inequalities. Consider the following instance of PI:

$$ax_1 \geq x_2 \qquad (1)$$

$$bx_2 \geq x_1 \qquad (2)$$

where a and b are two symbols. Let us run RSA-PI on these Prefix Inequalities. An initial segment of the execution trace is shown in Figure 4.1.2.

It is clear that we can alternate the relaxations of inequalities (1) and (2) indefinitely but never satisfy them simultaneously. Just as in the previous example, the trouble stems from some kind of "negative" cycles. Note that from inequality (1) we have:

$$bax_1 \geq bx_2. \qquad (3)$$

Using the transitivity of $\geq$, we get from (2) and (3) the following inequality:

$$bax_1 \geq x_1,$$

which clearly can never be satisfied by any (finite) string x_1. ■

The above two examples show that it is indeed possible for GRSA never to halt on some inputs. In order to make GRSA practically usable, we would like to modify GRSA in such a way that, whenever $\Xi \downarrow p_0 = \varnothing$, the improved version of GRSA is able to discover and report this fact *in finitely many steps*. We will do this for SPP and PI in the next two sections.

$$ax_1 \geq x_2 \qquad (1)$$

$$bx_2 \geq x_1 \qquad (2)$$

$$(x_1, x_2) =$$

$$\begin{tikzcd}
(\varepsilon, \varepsilon) \arrow[d, "(1)"] \\
(\varepsilon, a) \arrow[d, "(2)"] \\
(ba, a) \arrow[d, "(1)"] \\
(ba, aba) \arrow[d, "(2)"] \\
(baba, aba) \arrow[d, "(1)"] \\
(baba, ababa) \arrow[d] \\
\vdots
\end{tikzcd}$$

Figure 4.1.2. An Instance of PI with a "Negative" Cycle.

4.2. Case Study I: LI-Formulated Shortest-Path Problem

In this section we will concentrate on one particular Relaxation Problem -- the LI-formulated Shortest-Path Problem. The major result of this section can be summarized as follows. Suppose an LI-formulated SPP has a greatest solution. Using the maximality of the greatest solution and some general properties of di-graphs, we will argue that there exists a *spanning forest* in the underlying di-graph with the properties that the Linear Inequalities corresponding to the tree arcs of the forest are satisfied as *equalities* by the greatest solution, and that the final values of the root nodes in the forest are their initial values. This result will enable us not only to design a polynomial-time algorithm for LI-formulated SPP, but also to finish the proof of the equivalence of the two formulations of SPP (Theorem 2.1.3).

Throughout the rest of this section we will work on the following instance of LI-formulated SPP:

$$\left.\begin{array}{c} \mathbf{I}_j : x_{from(j)} + L_j \geq x_{to(j)},\ 1 \leq j \leq m \\ \\ 0 \geq x_0 \end{array}\right\} \qquad (4.2.1)$$

which is associated with an instance $(\mathbf{G}, L)$ of SPP, where $\mathbf{G} = (\mathbf{V}, \mathbf{A})$, $\mathbf{V} = \{0, 1, 2, \ldots, n\}$ (node 0 is the source node), $\mathbf{A} = \{1, 2, \ldots, m\}$, and $L : \mathbf{A} \to \mathbf{Z}$. We also make the following assumption:

Assumption: LI (4.2.1) *have a solution.*

By Corollary 3.3.4, (4.2.1) has a greatest solution, which is denoted by $\mathbf{d} = (d_0, d_1, \ldots, d_n)$. Let us apply RSA-SPP to (4.2.1). The initial approximation is $(x_0, x_1, \ldots, x_n) = (0, \infty, \ldots, \infty)$. By Theorem 3.3.3, RSA-SPP must reach the final approximation $(x_0, x_1, \ldots, x_n) = (d_0, d_1, \ldots, d_n)$ in finitely many steps. So, we can say that the *initial value* of x_i (or, at node i) is ∞ for $1 \leq i \leq n$, or 0 if $i = 0$. Similarly, we say that $d_0, d_1, \ldots, d_n$ are the *final values* of $x_0, x_1, \ldots, x_n$, respectively. Note that, by the monotonicity of RSA-SPP (Lemma 3.3.2), the final value at each node is less than or equal to the initial value.

Our first observation is that there must be some node whose final value is equal to its initial value, as proved below.

Lemma 4.2.1: Let

$$\mathbf{V}_r = \{\, i \in \mathbf{V} \mid d_i \text{ is the initial value of } x_i \,\}.$$

Then $\mathbf{V}_r \neq \varnothing$.

Proof: In fact, we will prove a stronger result: $0 \in \mathbf{V}_r$ (i.e., $d_0 = 0$). Suppose the contrary: $d_0 < 0$. Let's consider the tuple

$$\mathbf{d}' = (d_0', d_1', \ldots, d_n') = (d_0 + 1, d_1 + 1, \ldots, d_n + 1).$$

It is easy to see that each inequality in (4.2.1) is still satisfied by $\mathbf{d}'$. In particular, $0 \geq d_0'$, since $d_0 \leq -1$. But $\mathbf{d}' > \mathbf{d}$, for at least $d_0' > d_0$. This contradicts to the assumption that $\mathbf{d}$ is the *greatest* solution of (4.2.1). ■

Remark: The reasons for which we choose to use the weaker result are: (a) the weaker result is already enough for our purpose, and (b) by using the weaker result, most of the arguements in this section can later be translated almost literally to prove similar results for PI. Remember that in the case of PI, there is no longer any distinguished source node.

Throughout the rest of this chapter "*tree*" really means "*directed tree*". Similarly, a *forest* is a finite collection of disjoint *directed* trees. We say that a tree in $\mathbf{G}$ is *well-rooted* iff its root is in $\mathbf{V}_r$. A forest in $\mathbf{G}$ is said to be well-rooted iff each tree of the forest is well-rooted.

Now we define a subgraph of $\mathbf{G}$, $\mathbf{G}_e = (\mathbf{V}, \mathbf{A}_e)$, as follows. For any arc $j \in \mathbf{A}$, $j \in \mathbf{A}_e$ if and only if

$$d_{from(j)} + L_j = d_{to(j)}. \tag{4.2.2}$$

In other words, an arc j is in $\mathbf{A}_e$ iff inequality $\mathbf{I}_j$ is satisfied as an *equality* by the greatest solution $\mathbf{d}$. Since $\mathbf{G}_e$ is finite and $\mathbf{V}_r \neq \varnothing$, there must exist a *maximal* well-rooted forest $\mathbf{F}_s = (\mathbf{V}_s, \mathbf{A}_s)$ in $\mathbf{G}_e$ (not $\mathbf{G}$!). That is, $\mathbf{F}_s$ is a well-rooted forest in $\mathbf{G}_e$ and for any $j \in \mathbf{A}_e - \mathbf{A}_s$ (or any $i \in \mathbf{V} - \mathbf{V}_s$), $\mathbf{F}_s \cup \{j\}$ (resp. $\mathbf{F}_s \cup \{i\}$) is not well-rooted, if it is a forest at all. It follows immediately that $\mathbf{V}_r \subseteq \mathbf{V}_s$, for $\mathbf{F}_s \cup \mathbf{V}_r$ is still a well-rooted forest in $\mathbf{G}_e$. What we want to show is that $\mathbf{F}_s$ is in fact a *spanning* forest of $\mathbf{G}$, i.e., $\mathbf{V}_s = \mathbf{V}$. To prove this, we need the following simple lemma.

Lemma 4.2.2: For any arc $j \in \mathbf{A}$, if $from(j) \in \mathbf{V}_s$ but $to(j) \in \mathbf{V} - \mathbf{V}_s$, then $j \notin \mathbf{A}_e$, that is,

$$d_{from(j)} + L_j > d_{to(j)}.$$

Proof: Suppose the contrary: $j \in \mathbf{A}_e$. Then clearly $\mathbf{F}_s \cup \{j\}$ is still a well-rooted forest in $\mathbf{G}_e$, contradicting to the maximality of $\mathbf{F}_s$. ■

Now we are prepared to prove the major result of this section, which is stated in the following theorem.

Theorem 4.2.3: $\mathbf{F}_s$ is a well-rooted *spanning* forest of $\mathbf{G}_e$ (hence of $\mathbf{G}$).

Proof: It remains to be shown that $\mathbf{V}_s = \mathbf{V}$. Suppose not, i.e., $\mathbf{V} - \mathbf{V}_s \neq \varnothing$. Define $\mathbf{d}' = (d_0', d_1', \ldots, d_n') \in \mathbf{Z}^{n+1}$ by

$$d_i' = \begin{cases} d_i & \text{if } i \in \mathbf{V}_s \\ d_i + 1 & \text{otherwise.} \end{cases}$$

We claim that $\mathbf{d}'$ satisfies each inequality in (4.2.1). First, notice that, since $\mathbf{V}_r \subseteq \mathbf{V}_s$, $0 \geq d_0'$ even when node $0 \in \mathbf{V}_r$.[1] Consider an arbitrary $j \in \mathbf{A}$. There are four possibilities as to the positions of $from(j)$ and $to(j)$:

(1) If $from(j) \in \mathbf{V}_s$ and $to(j) \in \mathbf{V}_s$, then

$$d'_{from(j)} + L_j = d_{from(j)} + L_j \geq d_{to(j)} = d'_{to(j)}.$$

(2) If $from(j) \notin \mathbf{V}_s$ and $to(j) \notin \mathbf{V}_s$, then

$$d'_{from(j)} + L_j = (d_{from(j)} + L_j) + 1 \geq d_{to(j)} + 1 = d'_{to(j)}.$$

(3) If $from(j) \notin \mathbf{V}_s$ and $to(j) \in \mathbf{V}_s$, then

[1]And it does! But we pretend ignorance. See the proof of Lemma 4.2.1 and the remark there.

$$d'_{from(j)} + L_j = (d_{from(j)} + L_j) + 1 > d_{from(j)} + L_j \geq d_{to(j)} = d'_{to(j)}.$$

(4) If *from* $(j) \in \mathbf{V}_s$ and *to* $(j) \notin \mathbf{V}_s$, then by Lemma 4.2.2

$$d_{from(j)} + L_j \geq d_{to(j)} + 1,$$

which implies that

$$d'_{from(j)} + L_j = d_{from(j)} + L_j \geq d_{to(j)} + 1 = d'_{to(j)}.$$

Therefore, in any one of the above four cases, $\mathbf{I}_j$ is satisfied by $\mathbf{d}'$. This completes the justification of our claim. Note that $\mathbf{V} - \mathbf{V}_s \neq \varnothing$ and $\mathbf{V}_r \subseteq \mathbf{V}_s$. So $\mathbf{d}' > \mathbf{d}$, since $d_i < \infty$ and $d_i' = d_i + 1$ for any $i \in \mathbf{V} - \mathbf{V}_s$. But this contradicts to the maximality of $\mathbf{d}$. So we conclude that $\mathbf{V}_s = \mathbf{V}$. ■

A Polynomial-Time Relaxation Algorithm for LI-Formulated SPP

With the help of the above theorem, we are now able to design a relaxation *algorithm* for solving LI-formulated SPP. This algorithm we call RA-SPP (Relaxation Algorithm - SPP), which is shown in Figure 4.2.1.

To understand RA-SPP, let us reason as follows. Given an LI-formulated SPP with $(n+1)$ nodes, suppose that it has a greatest solution. By Theorem 4.2.3, there exists a *spanning forest* $\mathbf{F}_s$ in the underlying di-graph with two important properties: (1) the final values at the root nodes of $\mathbf{F}_s$ are equal to their initial values; (2) the inequalities corresponding to the tree arcs of $\mathbf{F}_s$ are satisfied as *equalities* by the final values. Therefore, if $\mathbf{F}_s$ were known *in advance*, then we could get the final value at each node in the di-graph by doing relaxations along the branches of $\mathbf{F}_s$ until every leaf node gets its final value. Since $\mathbf{F}_s$ contains no more than n arcs, we could get the desired solution using no more than n relaxation steps.

Of course, we can *not* have known $\mathbf{F}_s$ in advance. But the existence of $\mathbf{F}_s$ does help us design better sequencing of relaxation steps. In RA-SPP, relaxation steps are arranged into *phases*. One phase corresponds to one iteration of the **for**-loop beginning at Line (2) of RA-SPP. During each phase RA-SPP checks all inequalities in turn (Lines (3) & (4)) and relaxes any out-of-kilter inequality whenever one is encountered (Line (5)). A node in $\mathbf{F}_s$ has *depth* k iff the (unique) directed path from a root node of $\mathbf{F}_s$ to this node contains k arcs. Initially, only the root nodes (depth 0) of $\mathbf{F}_s$ have got their final values. During the first phase, any inequality corresponding to a tree arc in $\mathbf{F}_s$ which connects a root node to a node of depth 1 is checked and relaxed if necessary. Recall that any relaxation process is monotonic (Lemma 3.3.2) and that any intermediate approximation is an upper bound of the final solution (Lemma 3.3.1). Hence, after the first phase, each node of depth 0 or 1 in $\mathbf{F}_s$ gets its final value. Similarly, after the second phase, each node of depth 0, 1, or 2 in $\mathbf{F}_s$ acquires its final value. And so on. It can be easily shown by induction on k that, after k phases, each node of depth no more than k in $\mathbf{F}_s$ acquires its final value. Since $\mathbf{F}_s$ is a *spanning* forest of a di-graph with $(n+1)$ nodes, *every* node has a depth no more than n in $\mathbf{F}_s$. Therefore, if the greatest solution does exist, RA-SPP must be able to reach it after n phases. If, after n phases, there is still some out-of-kilter inequality, then there must not be any solution at all! This explains Lines (6) -

```
    var $x$ : array [0 .. $n$] of $\overline{\mathbf{Z}}$;
    begin
(1)  $x \leftarrow (0, \infty, \infty, \ldots, \infty)$;
(2)  for $k \leftarrow 1$ to $n$ do
(3)    for $j \leftarrow 1$ to $m$ do
(4)      if $x_{from(j)} + L_j < x_{to(j)}$ then
(5)        $x_{to(j)} \leftarrow x_{from(j)} + L_j$     /* Relaxation Step */
         end if
       end do
    end do;
(6)  if $\forall j\,(1 \le j \le m)\; x_{from(j)} + L_j \ge x_{to(j)}$ then
(7)    print("The solutions are: ", $x_0, x_1, \ldots, x_n$)
       else
(8)    print("Error: No solution at all!")
       end if
    end.
```

Figure 4.2.1.

RA-SPP: A Polynomial-Time Relaxation Algorithm for Solving LI-formulated SPP.

(8) of RA-SPP.

Time Complexity of RA-SPP

Line (1) takes O(n) time. Each iteration through the part from Line (4) to Line (5) takes only O(1) time. So the **for**-loop beginning at Line (2) takes O(nm) time. The rest of the algorithm takes at most O(n) + O(m) time. So the total time complexity is O(nm).

Notice that instead of executing exactly n phases, RA-SPP could stop as soon as all inequalities are satisfied. This can reduce the number of phases in *some* occasions. But the worst-case time complexity is unaffected anyway. So we have ignored this improvement in Figure 4.2.1 for simplicity's sake. The same remark applies to RA-PI (Figure 4.3.1).

Finishing the Proof of Theorem 2.1.3

The rest of this section is devoted to finishing the proof of Theorem 2.1.3, which asserts the equivalence of the two formulations of SPP we used in this Thesis. The only not-yet-proved part of Theorem 2.1.3 is its *if* part, which follows immediately from the following lemma.

Lemma 4.2.4: Given an instance of SPP and its associated LI, if there exists a tuple of integers satisfying the associated LI, then the shortest distance to each node in the di-graph of the SPP (from the source node) is well-defined.

Proof: Without any loss of generality, consider the LI (4.2.1) and its corresponding SPP. By Theorem 4.2.3 there exists a well-rooted spanning forest $\mathbf{F}_s$ of $\mathbf{G}_e$ and (hence) $\mathbf{G}$. Consider an arbitrary well-rooted tree $\mathbf{T}$ in $\mathbf{F}_s$ whose root node has final value ∞. Since $\infty + x = \infty$ for any $x \in \mathbf{Z}$ and $\mathbf{T}$ is a tree in $\mathbf{G}_e$, it follows that for any node i in $\mathbf{T}$, $d_i = \infty$. Recall that the initial value at any node *except the source node* (node 0) is ∞. So, if there is any node with a final value $< \infty$, then it must belong to a tree in $\mathbf{F}_s$ rooted at node 0. In particular, we know from the proof of Lemma 4.2.1 that $d_0 = 0 < \infty$. Hence node 0 is the root of a tree in $\mathbf{F}_s$, which we denote by $\mathbf{T}_0$.

Now consider an arbitrary node i in $\mathbf{G}$. If i is not reachable from the source node, then the shortest distance to i is trivially well-defined, namely ∞. So, assume node i is reachable from node 0. Let P be an arbitrary path from node 0 to node i. Since $L_j < \infty$ for any $j \in \mathbf{A}$, $L(P) < \infty$. Observe that we can run RSA-SPP on this instance of SPP in the following manner: we start with the initial value at each node in the di-graph, and then do relaxations from node 0 to node i along path P. When we get to node i, $x_i = L(P)$. By Lemma 3.3.1, $L(P) \geq d_i$. Hence node i is in tree $\mathbf{T}_0$. Note that there is a unique path P^* from node 0 to node i which is entirely in tree $\mathbf{T}_0$. Since P^* is in $\mathbf{G}_e$, $L(P^*) = d_i$. Since we have shown that no path from node 0 to node i can be shorter than d_i, P^* is a *shortest path* from node 0 to node i. So the shortest distance to node i is well-defined. The proof is now complete. ■

4.3. Case Study II: Prefix Inequalities

In this section our attention is turned to the study of Prefix Inequalities. The results and arguments are extremely analogous to those in the preceding section. More specifically, we will also argue the existence of some kind of well-rooted spanning forest, and then use the result to design a polynomial-time relaxation algorithm for solving PI.

Throughout the rest of this section we will work on the following instance of PI:

$$\mathbf{I}_j : \lambda_j x_{from(j)} \geq \rho_j x_{to(j)}, \quad j \in \mathbf{A} \tag{4.3.1}$$

where mappings *from* and *to* are defined in terms of a graph $\mathbf{G} = (\mathbf{V}, \mathbf{A})$, $\mathbf{V} = \{1, 2, \ldots, n\}$, $\mathbf{A} = \{1, 2, \ldots, m\}$, and λ_j and ρ_j ($j \in \mathbf{A}$) are (finite) strings over an alphabet Σ. We also make the following assumption:

Assumption: PI (4.3.1) *have a solution.*

By Corollary 3.3.4, (4.3.1) has a *greatest* solution, which is denoted by $\mathbf{d} = (d_1, \ldots, d_n)$. Let us apply RSA-PI (Figure 3.4.2) to (4.3.1). The initial approximation is $(x_1, \ldots, x_n) = (\varepsilon, \ldots, \varepsilon)$. By Theorem 3.3.3, RSA-PI must reach the final approximation $(x_1, \ldots, x_n) = (d_1, \ldots, d_n)$ in finitely many steps. So, we can say that ε is the *initial value* of x_i (or, at node i), for any $1 \leq i \leq n$. Similarly, we say that $d_1, \ldots, d_n$ are the *final values* of $x_1, \ldots, x_n$, respectively. Note that by the monotonicity of RSA-PI (Lemma 3.3.2), the initial value at each node is a prefix of ($\geq$) the final value.

Before proving any results, we need to define a unary operation on strings. For any string $\sigma \in \Sigma^*$ and $\sigma \neq \varepsilon$, let $\sigma\rceil$ denote the string obtained from σ by stripping off the last (right-most) symbol of σ. For example, $(aba)\rceil = ab$, $(a)\rceil = \varepsilon$, where $a, b \in \Sigma$.

Our first result asserts that there must be some node whose final value is equal to its initial value.

Lemma 4.3.1: Let

$$\mathbf{V}_r = \{ i \in \mathbf{V} \mid d_i = \varepsilon \}.$$

Then $\mathbf{V}_r \neq \varnothing$.

Proof: Suppose the contrary: $d_i \neq \varepsilon$ for each $i \in \mathbf{V}$. Let

$$\mathbf{d}' = (d_1', \ldots, d_n') = (d_1\rceil, \ldots, d_n\rceil).$$

Consider an arbitrary inequality $\mathbf{I}_j$ ($j \in \mathbf{A}$) in (4.3.1). Note that

$$\lambda_j d'_{from(j)} = (\lambda_j d_{from(j)})\rceil \geq (\rho_j d_{to(j)})\rceil = \rho_j d'_{to(j)}.$$

Hence each inequality in (4.3.1) is still satisfied by $\mathbf{d}'$. But $\mathbf{d}' > \mathbf{d}$, contradicting to the assumption that $\mathbf{d}$ is the *greatest* solution to (4.3.1). ■

We say that a (directed) tree in $\mathbf{G}$ is *well-rooted* iff its root belongs to $\mathbf{V}_r$. A forest in $\mathbf{G}$ is said to be well-rooted iff each tree of the forest is well-rooted.

Now we define a subgraph of $\mathbf{G}$, $\mathbf{G}_e = (\mathbf{V}, \mathbf{A}_e)$, as follows. For any arc $j \in \mathbf{A}$, $j \in \mathbf{A}_e$ if and only if

$$\lambda_j d_{from(j)} = \rho_j d_{to(j)}. \tag{4.3.2}$$

In other words, an arc j belongs to $\mathbf{A}_e$ iff inequality $\mathbf{I}_j$ is satisfied as an *equality* by the greatest solution $\mathbf{d}$. Since $\mathbf{G}_e$ is finite and $\mathbf{V}_r \neq \emptyset$, there must exist a *maximal* well-rooted forest $\mathbf{F}_s = (\mathbf{V}_s, \mathbf{A}_s)$ in $\mathbf{G}_e$ (not $\mathbf{G}$!). That is, $\mathbf{F}_s$ is a well-rooted forest in $\mathbf{G}_e$ and for any $j \in \mathbf{A}_e - \mathbf{A}_s$ (or any $i \in \mathbf{V} - \mathbf{V}_s$), $\mathbf{F}_s \cup \{j\}$ (resp. $\mathbf{F}_s \cup \{i\}$) is not well-rooted, if it is a forest at all. It follows that $\mathbf{V}_r \subseteq \mathbf{V}_s$, for $\mathbf{F}_s \cup \mathbf{V}_r$ is still a well-rooted forest in $\mathbf{G}_e$. What we want to show is that $\mathbf{F}_s$ is in fact a *spanning* forest of $\mathbf{G}$, i.e., $\mathbf{V}_s = \mathbf{V}$. To prove this, we need the following simple lemma.

Lemma 4.3.2: For any arc $j \in \mathbf{A}$, if $from(j) \in \mathbf{V}_s$ but $to(j) \in \mathbf{V} - \mathbf{V}_s$, then $j \notin \mathbf{A}_e$, that is,

$$\lambda_j d_{from(j)} > \rho_j d_{to(j)}.$$

Proof: Suppose the contrary: $j \in \mathbf{A}_e$. Then clearly $\mathbf{F}_s \cup \{j\}$ is still a well-rooted forest in $\mathbf{G}_e$, contradicting to the maximality of $\mathbf{F}_s$. ■

Now we are prepared to prove the major result of this section, which is stated in the following theorem.

Theorem 4.3.3: $\mathbf{F}_s$ is a well-rooted *spanning* forest of $\mathbf{G}_e$ (hence of $\mathbf{G}$).

Proof: It remains to be shown that $\mathbf{V}_s = \mathbf{V}$. Suppose the contrary, i.e., $\mathbf{V} - \mathbf{V}_s \neq \emptyset$. Define $\mathbf{d}' = (d_1', \ldots, d_n') \in (\Sigma^*)^n$ by

$$d_i' = \begin{cases} d_i & \text{if } i \in \mathbf{V}_s \\ d_i \rceil & \text{otherwise.} \end{cases}$$

Note that $\mathbf{d}'$ is well-defined: since $\mathbf{V}_r \subseteq \mathbf{V}_s$, so $d_i \neq \varepsilon$ for any $i \in \mathbf{V} - \mathbf{V}_s$. We claim that $\mathbf{d}'$ satisfies each inequality in (4.3.1). Consider an arbitrary $j \in \mathbf{A}$. There are four possibilities as to the positions of $from(j)$ and $to(j)$:

(1) If $from(j) \in \mathbf{V}_s$ and $to(j) \in \mathbf{V}_s$, then

$$\lambda_j d'_{from(j)} = \lambda_j d_{from(j)} \geq \rho_j d_{to(j)} = \rho_j d'_{to(j)}.$$

(2) If $from(j) \notin \mathbf{V}_s$ and $to(j) \notin \mathbf{V}_s$, then

$$\lambda_j d'_{from(j)} = (\lambda_j d_{from(j)})\rceil \geq (\rho_j d_{to(j)})\rceil = \rho_j d'_{to(j)}.$$

(3) If $from(j) \notin \mathbf{V}_s$ and $to(j) \in \mathbf{V}_s$, then

$$\lambda_j d'_{from(j)} = (\lambda_j d_{from(j)})\rceil > \lambda_j d_{from(j)} \geq \rho_j d_{to(j)} = \rho_j d'_{to(j)}.$$

(4) If $from(j) \in \mathbf{V}_s$ and $to(j) \notin \mathbf{V}_s$, then by Lemma 4.3.2

$$\lambda_j d_{from(j)} \geq (\rho_j d_{to(j)})\rceil,$$

which implies that

$$\lambda_j d'_{from(j)} = \lambda_j d_{from(j)} \geq (\rho_j d_{to(j)})\rceil = \rho_j d'_{to(j)}.$$

Therefore, in any one of the above four cases, $\mathbf{I}_j$ is satisfied by $\mathbf{d}'$. This completes the justification of our claim. Since $\mathbf{V} - \mathbf{V}_s \neq \emptyset$, $\mathbf{d}' > \mathbf{d}$. But this contradicts to the maximality of $\mathbf{d}$. So we conclude that $\mathbf{V}_s = \mathbf{V}$. ■

A Polynomial-Time Relaxation Algorithm for Solving PI

With the help of the above theorem, we are now able to design a relaxation *algorithm* for solving PI. This algorithm we call RA-PI (Relaxation Algorithm - PI), which is shown in Figure 4.3.1. The justification of the correctness of RA-PI is almost identical to that of RA-SPP and hence is omitted. Here we just note that, since there are now only n nodes in the di-graph, at most $(n-1)$ phases of relaxation steps are needed.

```
    var $x$ : array [1 .. $n$] of $\Sigma^*$ ;
    begin
(1)   $x \leftarrow (\varepsilon, \varepsilon, \ldots, \varepsilon)$;
(2)   for $k \leftarrow 1$ to $(n-1)$ do
(3)     for $j \leftarrow 1$ to $m$ do
(4)       if $\lambda_j x_{from(j)} < \rho_j x_{to(j)}$ then
(5)         $x_{to(j)} \leftarrow \rho_j^{-1}(\lambda_j x_{from(j)})$     /* Relaxation Step */
          end if
        end do
      end do;
(6)   if $\forall j\ (1 \le j \le m)\ \lambda_j x_{from(j)} \ge \rho_j x_{to(j)}$ then
(7)     print("The solutions are: ", $x_1, \ldots, x_n$)
      else
(8)     print("Error: No solution at all!")
      end if
    end.
```

Figure 4.3.1.
RA-PI: A Polynomial-Time Relaxation Algorithm for Solving PI.

Time Complexity of RA-PI

First we need to estimate the lengths of the final values $d_1, d_2, ..., d_n$ (if they exist). Note that all of them can be obtained by doing relaxations along the branches of a well-rooted spanning forest $\mathbf{F}_s$. Since the maximum depth of a node in $\mathbf{F}_s$ is $(n-1)$, the length of any d_i is of the order of $O(ln)$, where

$$l = \max \{ \, |\lambda_j|, \, |\rho_j| \;\; | \;\; 1 \le j \le m \, \}.$$

Therefore, each pattern-match operation (in Lines (4) and (6)) and each relaxation step (Line (5)) take at most $O(ln)$ time.

Now we can estimate the total running time of RA-PI. Line (1) takes $O(n)$ time. Each iteration through the part from Line (4) to Line (5) takes $O(ln)$ time. So the **for**-loop beginning at Line (2) takes $O(lmn^2)$ time. The **if**-test at Line (6) takes $O(lmn)$ time. The output of answers at Line (7) takes $O(ln^2)$ time. Line (8) takes only $O(1)$ time. So the total time complexity is $O(lmn^2)$.

CHAPTER 5.
An Application of Relaxation Techniques: Type Inference for Prolog

After doing much theoretical work in the last two chapters, we now turn to the investigation of the more practical aspects of relaxation techniques. In this chapter we demonstrate the power of relaxation techniques on a real-world application: polymorphic type inference for Prolog. An extensive knowledge of Prolog, however, is not required in order to understand the materials in this chapter. An outline of the syntax of Prolog is given in Section 5.1. Those who do not know Prolog at all are referred to the excellent introductory textbook by Clocksin and Mellish [ClMe].

We begin with introducing a polymorphic type system for Prolog and briefly discussing why we need a type system and the corresponding type inference. Then we show that the task of type inference is reducible to that of solving a system of Term Inequalities and, hence, can be automated, as illustrated in Chapter 2. The automation of type inference results in a program called **TYPEINF** which, given a small amount of type information, is able to analyze a set of Prolog clauses and generate most general types for the predicates defined by those clauses. The type inference strategy adopted by **TYPEINF** is described. Finally, we demonstrate the capabilities of **TYPEINF** by applying it to some test programs including itself, since it is also coded in Prolog.

A complete listing of **TYPEINF** and some of its test data can be found in Appendix II.

5.1. A Polymorphic Type System for Prolog

Prolog, like Lisp, was designed as a typeless programming language -- every data object in Prolog is a *term*, and vice versa. No static (i.e., compile-time) type checking is done. The typeless approach of programming, though flexible, is highly error-prone, as the usefulness of strong typing in detecting errors at compile-time has been well-known through such languages as Pascal. If we want to use Prolog as a serious tool for building large software systems, a type system for Prolog which enables static type checking is highly desirable.

Furthermore, any type system for Prolog should be *polymorphic*. *Polymorphism* means that types may contain *parameters* which can be *instantiated* to other types, thus giving a good degree of flexibility to the type system. For instance, we will give the type $(list(\alpha), list(\alpha), list(\alpha))$ to predicate **append/3** (Example 5.1.1) which defines the relation that its third argument is the list concatenation of its first and second arguments. The type parameter α may be instantiated to *integer*, *real*, *list*(β), or any other types. Without polymorphism, we need a separate type (and hence a separate copy of code) of **append/3** for *each* of the above types of data objects.

In [MyOK] Mycroft and O'Keefe introduced such a polymorphic type system for Prolog. They also described a compile-time type checker based on their type sytem, which can check the well-typedness of a Prolog program provided with sufficient type specifications. In this chapter we would like to go still further: is it possible to implement a program that, given only *partial* type specifications of a Prolog program, can automatically infer the types of those unspecified objects? The answer is positive, as the reader will see.

Before we can describe the polymorphic type system of Mycroft and O'Keefe, we need to briefly review the syntax of Prolog.

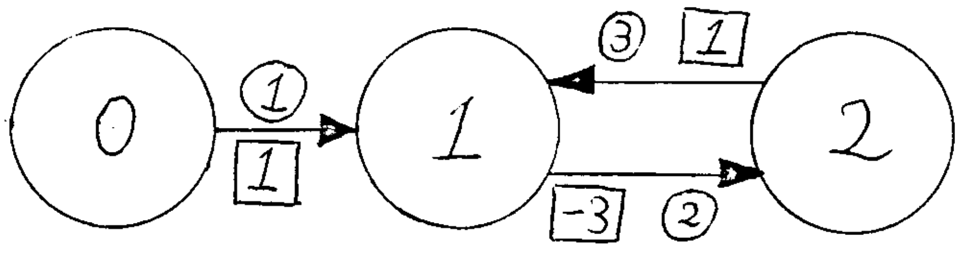
Outline of the Syntax of Prolog

A Prolog program is just a (finite) set of *clauses*. A clause is the disjunction of a number of positive and/or negative *literals*. Prolog allows only the so-called *Horn clauses*, which are clauses containing at most one positive literal. The Horn clauses containing one positive literal are called *positive clauses*, which correspond to the procedure definitions in a conventional programming language. The other Horn clauses are called *negative clauses*, which are queries to the database defined by the positive clauses. We are only interested in positive clauses, which we shall simply call *clauses*. The positive literal in a clause is written on the left-hand side of a **:-** sign[1] (read as "..., *if* ..."); all other (negative) literals are written on the right-hand side of **:-**, if there are any. A literal is a *predicate* applied to a tuple of *terms*, the number of which is called the *arity* of that predicate. Terms have been defined in Section 2.2. We notice that a variable in a Prolog program is local to the clause in which it occurs. (The skeleton of) the syntax of Prolog can be summarized in the following BNF-like notation:

```
Program ::= Clause*
Clause ::= Literal ':-' Literal*
Literal ::= Predicate '(' Term* ')'
Term ::= Variable | Functor '(' Term* ')'
```

In this chapter, unless stated otherwise, we will write fragments of Prolog codes in **bold face**. We also follow the common practice of capitalizing the first letter in the name of a Prolog variable to distinguish it from Prolog functors and predicates, whose names begin with lower-case letters. Underscores also denote Prolog variables.

[1]Some dialects of Prolog use **<-** instead of **:-**.



$(x_0, x_1, x_2) =$

$$\begin{tikzcd}
(0,\infty,\infty) \arrow[d, "\text{(arc 1)}"] \\
(0,1,\infty) \arrow[d, "\text{(arc 2)}"] \\
(0,1,-2) \arrow[d, "\text{(arc 3)}"] \\
(0,-1,-2) \arrow[d, "\text{(arc 2)}"] \\
(0,-1,-4) \arrow[d, "\text{(arc 3)}"] \\
(0,-3,-4) \arrow[d, "\text{(arc 2)}"] \\
(0,-3,-6) \arrow[d] \\
\vdots
\end{tikzcd}$$

Figure 4.1.1. A Shortest-Path Problem with a Negative Cycle.

Type Algebra

The scheme of *types* of [MyOK] is simple: a type is simply a term made of *type constructors* (ranged over by *Italic* words) and *type variables* (ranged over by Greek letters near the beginning of the alphabet, such as α, β, γ). Types are ranged over by (possibly subscripted) σ and τ. The scope of a type variable is the extended type (see later) in which it occurs. Some examples of types are:

$$integer,\ list(list(integer)),\ list(\alpha),$$

which are, respectively, the types of integers, lists of lists of integers, and general lists.

Therefore, we have actually two kinds of *terms*: type terms and Prolog terms. In order to avoid any conceptual ambiguities, we assume that the sets of (Prolog) functors, (Prolog) variables, type constructors and type variables are mutually disjoint. The set of (Prolog) terms is ranged over by ***Bold-Italic*** words.

An *extended type* is a type, or a tuple of types:

$$(\tau_1, \tau_2, \ldots, \tau_n),$$

or a tuple of types paired with another type:

$$(\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \sigma,$$

where $n \geq 0$. Extended types are ranged over by ρ (possibly with subscripts).

Extended types, like types, can also be viewed as terms by considering the tupling operations and $\rightarrow$ as new type constructors.[1] Therefore, the relations "is more general than" ($\geq$), "is an instance of" ($\leq$), and "is obtainable by variable renaming from" ($\approx$) are also defined over extended types. (See Section 2.2 for the definitions of these relations.) For example,

$$list(\alpha) \geq list(list(integer)),$$

$$((\beta, list(\beta)) \rightarrow list(\beta)) \geq ((real, list(real)) \rightarrow list(real)).$$

Now we are prepared to describe the polymorphic type system proposed by Mycroft and O'Keefe, which is the subject of next subsection.

Well-Typing of Prolog

Let $\boldsymbol{P}$ be a Prolog program. Since a Prolog variable is local to the clause in which it appears, we may assume without loss of generality that variables in different clauses of $\boldsymbol{P}$ have different names. A *typing* $t(\boldsymbol{P})$ of $\boldsymbol{P}$ consists of two sets of type associations:

(i) *type premises*, which associate an extended type ρ with each symbol s (predicate, functor or variable) occurring in $\boldsymbol{P}$, denoted by:

[1]But they are *not* type constructors in the strict sense, because they can not be used recursively in constructing extended types.

$$s : \rho$$

(ii) *type attachments*, which associate a (non-extended) type τ with each (sub)term t in $\boldsymbol{P}$, denoted by attaching the type to the (sub)term as superscript:

$$t^{\tau}$$

We say that $t(\boldsymbol{P})$ *well-types* $\boldsymbol{P}$, or $t(\boldsymbol{P})$ *is a well-typing of* $\boldsymbol{P}$, if and only if all of the following conditions hold:

(0) The type premises of $t(\boldsymbol{P})$ associate with each symbol in $\boldsymbol{P}$ an extended type of the correct form and arity. More specifically, for each variable $\mathbf{X}$ in $\boldsymbol{P}$, each predicate $\mathbf{p}$ of arity n in $\boldsymbol{P}$, and each functor f of arity n in $\boldsymbol{P}$, $t(\boldsymbol{P})$ contains type premises of the following forms:

$$\mathbf{X} : \tau,$$

$$\mathbf{p} : (\tau_1, \tau_2, ..., \tau_n),$$

$$\mathbf{f} : (\tau_1, \tau_2, ..., \tau_n) \rightarrow \sigma,$$

where $n \geq 0$.

(1) For each literal at the head of a clause in $\boldsymbol{P}$ of the form

$$\mathrm{p}(t_1, t_2, ..., t_n) \text{ :- } ....$$

where $t_1, t_2, ..., t_n$ are (Prolog) terms, if $t(\boldsymbol{P})$ contains the type premise

$$\mathbf{p} : (\tau_1, \tau_2, ..., \tau_n)$$

and the type attachments

$$t_1^{\sigma_1}, t_2^{\sigma_2}, ..., t_n^{\sigma_n},$$

then

$$(\sigma_1, \sigma_2, ..., \sigma_n) \approx (\tau_1, \tau_2, ..., \tau_n).$$

(Recall that $\sigma \approx \tau$ iff both $\sigma \geq \tau$ and $\sigma \leq \tau$.)

(2) For each literal in the body of a clause in $\boldsymbol{P}$ of the form

$$... \text{ :- } ..., \mathrm{p}(t_1, t_2, ..., t_n), ...$$

where $t_1, t_2, ..., t_n$ are (Prolog) terms, if $t(\boldsymbol{P})$ contains the type premise

$$\mathbf{p} : (\tau_1, \tau_2, ..., \tau_n)$$

and the type attachments

$$t_1^{\sigma_1}, t_2^{\sigma_2}, ..., t_n^{\sigma_n},$$

then

$$(\tau_1, \tau_2, ..., \tau_n) \geq (\sigma_1, \sigma_2, ..., \sigma_n).$$

(3) For each (sub)term t in $\boldsymbol{P}$ of the form

$$\mathbf{f}(t_1, t_2, \ldots, t_n)$$

where $t_1, t_2, \ldots, t_n$ are (Prolog) terms, if $t(P)$ contains the type premise

$$\mathbf{f} : (\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau$$

and the type attachments

$$t^{\sigma}, t_1^{\sigma_1}, t_2^{\sigma_2}, \ldots, t_n^{\sigma_n},$$

then

$$((\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau) \geq ((\sigma_1, \sigma_2, \ldots, \sigma_n) \rightarrow \sigma).$$

(4) For each variable **X** in P, if $t(P)$ contains the type premise

$$\mathbf{X} : \tau,$$

then it must also contain the type attachment

$$\mathbf{X}^{\tau}.$$

Example 5.1.1: Consider the Prolog program defining the concatenation of lists:

```
append(nil, L1, L1).
append(cons(X, T1), L2, cons(X, T2)) :-
     append(T1, L2, T2).
```

A well-typing of this program contains the type premises

$$\mathbf{nil} : () \rightarrow list(\alpha),$$

$$\mathbf{cons} : (\beta, list(\beta)) \rightarrow list(\beta),$$

$$\mathbf{append} : (list(\gamma), list(\gamma), list(\gamma)),$$

$$\mathbf{L1} : list(\eta_1),\ \mathbf{X} : \eta_2,\ \mathbf{T1} : list(\eta_2),\ \mathbf{L2} : list(\eta_2),\ \mathbf{T2} : list(\eta_2),$$

and the type attachments

$$\mathbf{append}(\mathbf{nil}^{list(\eta_1)}, \mathbf{L1}^{list(\eta_1)}, \mathbf{L1}^{list(\eta_1)}).$$

$$\mathbf{append}(\mathbf{cons}(\mathbf{X}^{\eta_2}, \mathbf{T1}^{list(\eta_2)})^{list(\eta_2)}, \mathbf{L2}^{list(\eta_2)}, \mathbf{cons}(\mathbf{X}^{\eta_2}, \mathbf{T2}^{list(\eta_2)})^{list(\eta_2)}) :-$$

$$\mathbf{append}(\mathbf{T1}^{list(\eta_2)}, \mathbf{L2}^{list(\eta_2)}, \mathbf{T2}^{list(\eta_2)}).$$

The reader can verify that all of the conditions (0) - (4) above are satisfied by these type associations. ■

It is quite clear from the above example that the effort of well-typing even a trivial Prolog program would be prohibitive to do entirely *by hand*. Some degree of automation is definitely needed to make the type system practical. This raises our interest in *type inference*, which is discussed in the next section.

5.2. Type Inference and Term Inequalities

By *type inference* we mean, in the context of the type system described in the last section, the completion of a partial typing (i.e., a subset of the necessary type associations) supplied by the user so that it satisfies the five conditions of well-typing. The power of a type inference system is inversely proportional to the amount of type associations the user must supply. The less type information the user must write down explicitly, the easier it is to use the type inference system.

Ideally we hope for a system in which the user writes programs without giving any type information. The system will do all the rest automatically. This is the goal taken by Mishra [Mi]. Unfortunately, he was not very successful in dealing with *polymorphism*, which is a key feature of our type system. Moreover, it is unclear how to choose appropriate type constructors without human intervention.

Mycroft and O'Keefe took another approach. The user is required to supply the type premises for all functors and predicates used in a Prolog program. It is not too difficult to infer everything else. This approach is quite practical, but it gives us more of a type *checker* than a type *inferencer*. Consider the fact that a typical Prolog program often contains the definitions of hundreds of predicates[1], each of which need be given a type specification. It seems that this approach still requires too much of the user.

Therefore, we propose a simpler approach: the user must supply only the type premises for functors; all other types, including the type premises for predicates, are inferred automatically. Since typical Prolog programs do not use too many functors, the labor needed to well-type a program can be greatly reduced.

Of course, we still need to supply the type premises for certain predicates in advance, namely, the types of built-in predicates. But this can be done once and for all. Also, when some kind of data abstraction is desired, it is necessary to give explicitly the type premises for the abstract data type access predicates, thus hiding the actual implementation of those abstract data types.

Now we need to address the problem of *how* the proposed type inference is done. It turns out that the task of type inference can be reduced to that of solving Term Inequalities. Let us first use an example to illustrate this point.

Example 5.2.1: Consider the program for **append/3** in Example 5.1.1 again. Suppose we are given only the type premises for the two functors used:

[1]One of the reasons for which Prolog programs typically contain a large number of predicate definitions is that the structure of Prolog programs is "flat" -- the body of a procedure (predicate) definition is in most cases just a *linear* list of procedure calls. A subproblem which is solved in a conventional programming language by one more level of nesting (e.g., a conditional branching or an iterative loop) usually incurs the definition of a new predicate in a Prolog program.

$$\textbf{nil} : () \to list(\alpha),$$

$$\textbf{cons} : (\beta, list(\beta)) \to list(\beta).$$

We would like to find the type premises for predicate **append/3** and all the variables, and the type attached to each (sub)term in the two clauses above, such that, when all put together, they form a well-typing of the program.

To begin, let us assign a type variable to each unknown type, as follows.

$$\textbf{append} : (\gamma_1, \gamma_2, \gamma_3),$$

$$\textbf{L1} : \delta_2,\ \textbf{X} : \delta_3,\ \textbf{T1} : \delta_4,\ \textbf{L2} : \delta_6,\ \textbf{T2} : \delta_7,$$

$$\textbf{append}(\textbf{nil}^{\delta_1}, \textbf{L1}^{\delta_2}, \textbf{L1}^{\delta_2}).$$

$$\textbf{append}(\textbf{cons}(\textbf{X}^{\delta_3},\textbf{T1}^{\delta_4})^{\delta_5}, \textbf{L2}^{\delta_6}, \textbf{cons}(\textbf{X}^{\delta_3},\textbf{T2}^{\delta_7})^{\delta_8}) \textbf{ :-}$$

$$\textbf{append}(\textbf{T1}^{\delta_4}, \textbf{L2}^{\delta_6}, \textbf{T2}^{\delta_7}).$$

Notice that conditions (0) and (4) of well-typing have been used in assigning these type variables. Conditions (1) - (3) impose more constraints, which are listed below with each constraint labelled by the number of the well-typing condition used.

$(\delta_1,\delta_2,\delta_2) \approx (\gamma_1,\gamma_2,\gamma_3)$	Cond. (1)
$(() \to list(\alpha)) \geq (() \to \delta_1)$	Cond. (3)
$(\delta_5,\delta_6,\delta_8) \approx (\gamma_1,\gamma_2,\gamma_3)$	Cond. (1)
$((\beta,list(\beta)) \to list(\beta)) \geq ((\delta_3,\delta_4) \to \delta_5)$	Cond. (3)
$((\beta,list(\beta)) \to list(\beta)) \geq ((\delta_3,\delta_7) \to \delta_8)$	Cond. (3)
$(\gamma_1,\gamma_2,\gamma_3) \geq (\delta_4,\delta_6,\delta_7)$	Cond. (2)

Thus the task of inferring the types of **append/3** and other symbols are reduced to that of solving a set of eight Term Inequalities. (Recall that $\sigma \approx \tau$ is equivalent to $\sigma \geq \tau$ and $\sigma \leq \tau$.) ■

It is generally true that type inference for Prolog is reducible to solution of Term Inequalities, in which the *terms* are type terms, not Prolog terms. A naive strategy of type inference is given below.

Naive Type Inference Strategy

Given a Prolog program $\boldsymbol{P}$, and the type premises for the functors used in $\boldsymbol{P}$ and for the predicates invoked but not defined in $\boldsymbol{P}$, execute the following steps:

Step 1: Assign an n-tuple of type variables to each predicate of arity n defined in $\boldsymbol{P}$, a type variable to each (Prolog) variable occurring in $\boldsymbol{P}$, and a type variable to each (sub)term in ***P***. Except the requirement that the last assignment should coincide with the second one on (Prolog) variables, all type variables assigned should be distinct. This step takes care of conditions (0) and (4) of well-typing.

Step 2: Analyze ***P*** and collect all constraints (Term Inequalities) imposed by conditions (1) - (3) of well-typing among the assigned type variables.

Step 3: Solve the set of Term Inequalities collected in *Step 2* using relaxation technique, as explained in Section 2.2. After every Term Inequality is satisfied (relaxed), the composition of all type variable substitutions used in the relaxation process gives most general types to the predicates defined in ***P***. ■

The above type inference strategy is called "naive" because not all inequalities collected in *Step 2* are necessary. This point will be made clearer in the next section.

Remark: It is now clear that the type premises for functors are the least we need to conduct "meaningful" type inference in our type system. Since the type premises for functors always appear on the left-hand side of $\geq$ in a Term Inequality, they will never get further instantiated during the whole relaxation process. So, if they were not already given and tuples of new type variables were assigned in their places, the constraints they impose would be effectively null!

5.3. TYPEINF: A Polymorphic Type Inferencer for Prolog

TYPEINF is a polymorphic type inferencer for Prolog. The input to **TYPEINF** is a Prolog program (i.e., a set of clauses), together with the type premises for the functors used in the program and for the predicates which are invoked but not defined in the program. The output is, if the input program is well-typeable (that is, there exists a well-typing of the input program which contains the user-supplied type premises), the type premises for the predicates defined by the program, or, if the input program is not well-typeable, failure of execution. The type inference strategy of **TYPEINF** is based on the naive strategy described in the preceding section, with the following improvements made:

(i) We need only the $\geq$-half of the bi-directional term inequality in condition (1) of well-typing, *provided that* we are only interested in inferring the type premises for predicates. This simplification is based on the following observation. The $\leq$-half of the term inequality only affects the instantiation states of the types of Prolog variables occurring in the head of the clause under consideration. But the scope of a Prolog variable is merely the clause in which it occurs. Also notice that the type of a Prolog variable never appears on the left-hand side of $\geq$ in any term inequality arising from the *body* of a clause (due to condition (2) or (3) of well-typing). Hence **TYPEINF** discards the $\leq$-half of any term inequality incurred by condition (1) of well-typing.

(ii) **TYPEINF** relaxes the term inequalities resulting from condition (3) of well-typing *immediately* after they are found, and then discards these term inequalities. The rationale is that, as pointed out in the remark at the end of the last section, the type premises of functors are (in fact, must be) always given and never changed (further instantiated) during the whole relaxation process. Hence the constraint on type variables created by the type premises of functors (i.e., condition (3)) can be immediately relaxed and never worried about again later. Note that the same argument does *not* hold for conditions (1) and (2) of well-typing.

Example 5.3.1: Continue Example 5.2.1. We first relax all inequalities resulting from condition (3) of well-typing. Relaxing the second inequality, we get:

$$\{ \delta_1 / list(\eta_1) \}.$$

Relaxing the fourth inequality, we get:

$$\{ \delta_3 / \eta_2, \delta_4 / list(\eta_2), \delta_5 / list(\eta_2) \}.$$

Relaxing the fifth inequality, we get:

$$\{ \eta_2 / \eta_3, \delta_7 / list(\eta_3), \delta_8 / list(\eta_3) \}.$$

The composition of these substitutions is:

$$\{ \delta_1 / list(\eta_1), \delta_3 / \eta_3, \delta_4 / list(\eta_3),$$

$$\delta_5 / list(\eta_3), \delta_7 / list(\eta_3), \delta_8 / list(\eta_3) \}.$$

After discarding the unnecessary and already-relaxed ones, the following inequalities are left:

$$(list(\eta_1),\delta_2,\delta_2) \geq (\gamma_1,\gamma_2,\gamma_3) \qquad (1)$$

$$(list(\eta_3),\delta_6,list(\eta_3)) \geq (\gamma_1,\gamma_2,\gamma_3) \qquad (2)$$

$$(\gamma_1,\gamma_2,\gamma_3) \geq (list(\eta_3),\delta_6,list(\eta_3)) \qquad (3)$$

Thus the number of Term Inequalities are reduced from the 8 in Example 5.2.1 to the current 3. The above set of Term Inequalities is exactly the one solved in Example 2.2.1, which through a relaxation process generates the following substitutions:

$$\{ \gamma_1 / list(\eta_6), \gamma_2 / list(\eta_6), \gamma_3 / list(\eta_6), \delta_6 / list(\eta_7), \eta_3 / \eta_7 \}.$$

Composed with the previous set of substitutions, they indeed provide a correct type premise for **append/3**:

$$\textbf{append} : (list(\eta_6), list(\eta_6), list(\eta_6)).$$

■

Further improvements of the efficiency of type inference are possible. For instance, consider the *calling graph* of a Prolog program, which is a di-graph with nodes representing predicates and arcs representing the calling relations between predicates. That is, there is an arc from node **p** to node **q** iff **p** calls **q** in the program. A type inferencer can first break the calling graph of the analyzed Prolog program into strongly connected components, and then perform type inference on each component separately. However, according to the author's experience, this would make the type inferencer much more complicated than it is now, thus obscuring the role of relaxation in the inferencer. Since the main theme of this Thesis is relaxation, not efficient type inference for Prolog, it seems appropriate to avoid this unnecessary complication.

TYPEINF is itself coded in Prolog. A complete listing of the documented codes can be found in Appendix II. The main routine of **TYPEINF** is contained in file infer.pl. File ineq.pl contains the routines for handling a single Term Inequality. File misc.pl contains some miscellaneous routines, including those accessing abstract data types. File work.pl contains some I/O routines. File op.pl contains operator declarations. File load, when consulted, loads all of the above files into the Prolog database.

5.4. Demonstrations of the Capabilities of TYPEINF

We first demonstrate **TYPEINF** on our familiar example: **append/3**. The following is obtained from the script of a Prolog session:

```
  cprolog

Cprolog version 1.4d.edai

| ?- [load].
op.pl consulted 0 bytes 0.083335 sec.
infer.pl consulted 3852 bytes 1.7 sec.
ineq.pl consulted 388 bytes 0.15 sec.
misc.pl consulted 1444 bytes 0.75 sec.
work.pl consulted 2724 bytes 1.25 sec.
load consulted 8408 bytes 4.1667 sec.

yes

| ?- type_inference(
|             [ ( append(nil, '$VAR'('L1'), '$VAR'('L1'))
|               ),
|               ( append(cons('$VAR'('X'), '$VAR'('T1')),
|                        '$VAR'('L2'),
|                        cons('$VAR'('X'), '$VAR'('T2')))
|                :- append('$VAR'('T1'), '$VAR'('L2'), '$VAR'('T2'))
|               ) ],
|             [ type(nil/0,   ('' -> list(Alpha))),
|               type(cons/2,  (''(Beta, list(Beta)) -> list(Beta)))
|             ],
|             InferredTypes).

InferredTypes = [type(append/3,
                      ''(list(_1327), list(_1327), list(_1327)))]
Beta = _82
Alpha = _66

yes
```

The result is the same as in Example 5.3.1. The first argument to **type_inference/3** is the Prolog program to be well-typed, represented as a list of clauses. The second argument is a list of user-supplied type premises for functors and predicates. When the goal succeeds, the third argument is instantiated to the inferred type premises for the predicates defined by the well-typed Prolog program. Notice that a variable in the well-typed program is represented in the (meta-)program **TYPEINF** as a (Prolog) term of the form $VAR(*Name*), where *Name* is the name of the variable represented by an atom. Type variables are represented by Prolog variables in **TYPEINF**.

Obviously, it is very inconvenient to key in all input data interactively. A better idea is to put the input data in a file and just refer to that file when the user is interacting with the Prolog interpreter. This is what will be done next. The content of file test is:

```
:- type list(T) => [] | [T | list(T)].
:- type pair(T, U) => (T - U).
:- type tree(T) => leaf(T) | branch(T, tree(T), tree(T)).


member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).

append([], L, L).
append([X | T1], L, [X | T2]) :- append(T1, L, T2).

reverse([], []).
reverse([X | T], R) :- reverse(T, S), append(S, [X], R).

pairing([], [], []).
pairing([X1 | T1], [X2 | T2], [(X1-X2) | T]) :-
      pairing(T1, T2, T).

append_many([], []).
append_many([L | LL], M) :-
      append_many(LL, N), append(L, N, M).

% In-order flattening of binary trees:

inorder(leaf(X), [X]).
inorder(branch(X, T1, T2), L) :-
      inorder(T1, L1), inorder(T2, L2),
      append_many([L1, [X], L2], L).
```

Note that we resumed the usual Prolog practice of writing null list as [ ] and list constructor as [<*head*> | <*tail*>]. The ":- **type ...**" commands are shorthands for specifying the type premises of functors. For example, the first **type** command in file test specifies exactly the same thing as the second argument to **type_inference/3** does in the last demonstration. In the following **work/1** is a predicate which, given the name of a file such as test, will read that file, prepare input for and invoke **type_inference/3**, and finally print out the inferred types.

```
| ?- work(test).

:- pred append(list(A), list(A), list(A)).

:- pred append_many(list(list(A)), list(A)).

:- pred inorder(tree(A), list(A)).

:- pred member(A, list(A)).

:- pred pairing(list(A), list(B), list(pair(A, B))).

:- pred reverse(list(A), list(A)).

yes
```

The results are correct, as the reader can easily verify. Note that "**:- pred ...**" is a shorthand for specifying the type premises of predicates.

The last, also the most interesting, demonstration is to apply TYPEINF to (a part of) itself. The file self contains the Prolog codes from file infer.pl (the main routine of TYPEINF) plus some necessary type informations. (A copy of self can also be found in Appendix II.) Applying TYPEINF to self, we get:

```
| ?- work(self).

:- pred arg_types_of_literal(type_tuple, literal,
             list(type_spec), list(type_spec)).

:- pred collect_ineqs(list(clause),
             list(type_spec), list(type_spec),
             list(term_ineq), list(term_ineq)).

:- pred collect_ineqs_in_body(clause,
             list(type_spec), list(type_spec), list(type_spec),
             list(term_ineq), list(term_ineq)).

:- pred collect_ineqs_in_clause(clause,
             list(type_spec), list(type_spec), list(type_spec),
             list(term_ineq), list(term_ineq)).

:- pred collect_ineqs_in_goal(predicate, literal,
             list(type_spec), list(type_spec), list(type_spec),
             list(term_ineq), list(term_ineq)).
```

```
:- pred collect_ineqs_in_head(literal,
             list(type_spec), list(type_spec), list(type_spec),
             list(term_ineq), list(term_ineq)).

:- pred init_pred_types(list(predicate), list(type_spec)).

:- pred init_var_types(list(variable), list(type_spec)).

:- pred member(A, list(A)).

:- pred relax_ineqs(list(term_ineq), flag, list(term_ineq)).

:- pred type_inference(list(clause), list(type_spec),
                       list(type_spec)).

:- pred welltype_term(term, list(type_spec), list(type_spec),
                      type).

:- pred welltype_term_list(list(term), list(type_spec),
                           list(type_spec), list(type)).

yes
```

CHAPTER 6.
Directions of Future Research

In this chapter we indicate some possible directions in which the work of this Thesis can be refined and/or extended, thus concluding our quest of the nature of relaxation processes in this Thesis. The reader is warned that this chapter is much more speculative and much less rigorous than the preceding chapters.

Fitting Term Inequalities into the Formal Framework

It was pointed out in Section 2.3 that Term Inequalities, due to their complexities, had resisted the development of a formal theory. This resistance forced us to consider a restricted version of Term Inequalities, namely, Prefix Inequalities, which were studied extensively in Chapters 3 and 4. Though Prefix Inequalities do preserve most of the "flavors" of Term Inequalities, many characteristics of the latter are lost in simplification into the former. For instance, the significance of the identities of *variables* in Term Inequalities, which can be readily seen from the examples of Chapter 5, is entirely lost in Prefix Inequalities. Therefore it is still desirable to fit Term Inequalities into a formal framework similar to that in Chapter 3, so that we can study their properties in a more rigorous way.

A Stronger Definition of Relaxation Problems

The major defect of the formal theory in Chapter 3 is that there exists merely a *semi*-algorithm, viz. GRSA, for solving general Relaxation Problems. However, there are indeed polynomial-time algorithms for solving the two Relaxation Problems (SPP and PI) we studied in this Thesis. This implies that our definition of Relaxation Problems (Definition 3.1.3) is too weak, in the sense that it lacks structural properties which enables the design of general relaxation algorithms.

One hopeful alternative is to define Relaxation Problems in terms of graphs or hypergraphs. Then it may be possible to design a general relaxation algorithm using some kind of "spanning forest" arguments, just as we did in Chapter 4 for SPP and PI. Other alternatives are also possible.

More Concrete Examples of Relaxation Processes

It is quite obvious that our development of the formal theory in Chapter 3 was motivated by the concrete examples in Chapter 2. Similarly, if we want to develop more powerful and more sophisticated theories of relaxation processes, it is imperative for us to discover more concrete examples of relaxation processes. The author believes that only from concrete examples can we draw the inspiration needed in developing good abstract theories.

Bibliography

[Bi] Birkhoff, G., *Lattice Theory* (3rd Ed.), American Mathematical Society Colloquium Publications, vol. 25 (1967).

[ClMe] Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, Springer-Verlag, 1981.

[En] Enderton, H. B., *A Mathematical Introduction to Logic* Academic Press, 1972.

[Ev] Even, S., *Graph Algorithm*, Computer Science Press, 1979.

[Mi] Mishra, P., "Towards a Theory of Types in Prolog", *Proc. IEEE Internat. Symp. Logic Programming*, Atlantic City, 1984.

[MyOK] Mycroft, A. and O'Keefe, R. A., "A Polymorphic Type System for Prolog", *Artificial Intelligence* 23 (1984).

APPENDIX I.
Some Mathematical Notions and Terminologies

In this appendix we summarize some mathematical notions and terminologies used in this Thesis. For the convenience of reference, we group terminologies by subject.

I.1. General Terminologies

We use *iff* as an abbreviation for *if and only if.*

Given n arbitrary sets $S_1, S_2, \ldots, S_n$ $(n \geq 1)$, their *Cartesian product* is denoted by

$$S_1 \times S_2 \times \ldots \times S_n = \{ (x_1, x_2, \ldots, x_n) \mid x_i \in S_i,\ 1 \leq i \leq n \}.$$

The set of *natural numbers* (i.e., nonnegative integers) is denoted by $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$. The set of *integers* is_denoted by $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The set of *extended integers* is denoted by $\overline{\mathbf{Z}} = \mathbf{Z} \cup \{\infty\}$, where ∞ is a new, distinct symbol denoting *positive infinity*. Note that we have

$$\infty + z = \infty - z = \infty, \quad \infty > z$$

for any $z \in \overline{\mathbf{Z}}$.

I.2. Graphs

A general reference on the theory of graphs is [Ev].

In this Thesis *di-graphs* mean directed multi-graphs *without self-loop.* More precisely, a di-graph $\mathbf{G}$ is a tuple, $\mathbf{G} = (\mathbf{V}, \mathbf{A}, \mathit{from}, \mathit{to})$, where $\mathbf{V}$ is a finite set of *nodes* (or, *vertices*), $\mathbf{A}$ is a finite set of *arcs*, *from* and *to* are two mappings from $\mathbf{A}$ into $\mathbf{V}$. For each arc $e \in \mathbf{A}$, $\mathit{from}(e)$ is the *from-node* of e and $\mathit{to}(e)$ is the *to-node* of e. For some minor technical reasons (see Section 3.2), we require that $\mathit{from}(e) \neq \mathit{to}(e)$ for any $e \in \mathbf{A}$. In a pictorial diagram of $\mathbf{G}$ an arc e is depicted as an arrow starting from its from-node and ending at its to-node. Note that we usually omit the *from* and *to* mappings and just write $\mathbf{G} = (\mathbf{V}, \mathbf{A})$.

A *path* P from node u to node v in $\mathbf{G}$ is a finite sequence of arcs in $\mathbf{G}$, $P = e_1 e_2 \ldots e_k$ $(k \geq 0)$ such that $u = \mathit{from}(e_1)$, $v = \mathit{to}(e_k)$, and $\mathit{to}(e_{j-1}) = \mathit{from}(e_j)$ for all $1 < j \leq k$. We can extend the notation and write $u = \mathit{from}(P)$ and $v = \mathit{to}(P)$. When $k = 0$, P is called an *empty path* and we require that $u = v$. A *cycle* is a path with identical from-node and to-node. We say that node v is *reachable* from node u in G iff there exists a path from node u to node v in G. So each node is reachable from itself by means of an empty cycle.

A di-graph $G' = (V', A')$ is called a *subgraph* of $G = (V, A)$ iff $V' \subseteq V$, $A' \subseteq A$, and whenever $e \in A'$, both *from*(e) and *to*(e) are in V'. Let $e \in A$ be an arc. By $G' \cup \{e\}$ we mean the subgraph of G whose arc set is $A' \cup \{e\}$ and whose node set is $V' \cup \{$*from*(e), *to*$(e)\}$.

A subgraph $T = (V', A')$ of $G = (V, A)$ is said to have a *root* r iff $r \in V'$ and every node $v \in V'$ is reachable from r by a path entirely in T. Furthermore, if that path is *unique* for every node $v \in V'$, then T is called a *(directed) tree* in G. Clearly a tree has exactly one root. A *(directed) forest* F in G is a subgraph of G which is the union of a number of disjoint trees in G. F is called a *spanning* forest of G iff the node set of F is the entire V, i.e., F contains every node of G.

I.3. Partial Orders and Posets

An general reference on the theory of posets is [Bi].

Given a set P, a *pre-order* $\geq$ on P is a relation on P with the following two properties:

(a) *Reflexivity*: $p \geq p$ for all $p \in P$.

(b) *Transitivity*: $p \geq q$ and $q \geq r$ imply $p \geq r$, for all $p, q, r \in P$.

If $\geq$ satisfy the additional property:

(c) *Anti-symmetry*: $p \geq q$ and $q \geq p$ imply $p = q$, for all $p, q \in P$.

then we say that $\geq$ is a *partial order* on P, $\geq$ *partially orders* P, or P is a *poset* (partially ordered set) with partial order $\geq$. We also say that $(P, \geq)$ is a poset. We write $p \leq q$ iff $q \geq p$, $p > q$ iff $p \geq q$ but $p \neq q$, and $p < q$ iff $q > p$. We say p and q are *incomparable* iff neither $p \geq q$ nor $p \leq q$ holds.

Let P be a poset with partial order $\geq$, S a subset of P. An *upper bound* q of S is an element in P such that $q \geq p$ for all $p \in S$. The *greatest element* of S is an element in S which is also an upper bound of S. It is easy to see (by anti-symmetry) that the greatest element of S is *unique* as long as it exists. (So we are justified in using article *the*.) We denote the greatest element of S, if it exists, by $\max S$. The *lower bounds* and *least element* of S are defined symmetrically with $\geq$ replaced by $\leq$ and max by min.

The *top* $\top$ and *bottom* $\bot$ of P are, respectively, the greatest element and the least element of the whole P. Note that either one may not exist. By convention, $\max \varnothing = \bot$ and $\min \varnothing = \top$, as long as the elements involved exist.

Some examples of posets are $\mathbf{N}$, $\mathbf{Z}$ and $\bar{\mathbf{Z}}$ with their natural orders, and Σ^* with the "is a prefix of" relation defined in Definition 2.3.1. Actually, the former three examples are *totally* ordered sets, i.e., there is no pair of incomparable elements. Note that $\bar{\mathbf{Z}}$ and Σ^* have top elements ∞ and ε, respectively, while neither $\mathbf{N}$ nor $\mathbf{Z}$ has a top element.

Let $(P_1, \geq_1), (P_2, \geq_2), \ldots, (P_n, \geq_n)$ be n posets. Their Cartesian product $\mathbf{P} = P_1 \times P_2 \times \ldots \times P_n$ can be partially ordered by the component-wise generalization of $\geq_1, \geq_2, \ldots, \geq_n$, that is, for any $\mathbf{p} = (p_1, p_2, \ldots, p_n)$, $\mathbf{q} = (q_1, q_2, \ldots, q_n) \in \mathbf{P}$,

$$\mathbf{p} \geq \mathbf{q} \text{ iff } p_i \geq_i q_i, \text{ for } 1 \leq i \leq n.$$

We say that $\geq$ is the *natural order* of the product $\mathbf{P}$ of posets $P_1, P_2, \ldots, P_n$.

APPENDIX II.
Program Listing and Test Data for **TYPEINF**

This appendix contains a complete listing of the program **TYPEINF** and some of its test data. The following files are listed (in that order):

infer.pl
ineq.pl
misc.pl
work.pl
op.pl
load
self

```
%
% FILE:         infer.pl
% AUTHOR:       Ching-Tsun Chou
% SPEC:         Main program of TYPEINF
%

% type_inference(+Program, +TypeEnv, -PredTypeSpecs)
% Program:
%       A list of clauses, in which each (Prolog) variable is represented
%       by a term of the form: '$VAR'(var-name), where var-name is an atom
%       or a number denoting the name of the variable.  This is the Prolog
%       program to be well-typed.
% TypeEnv:
%       A list of type associations, each of which is of the form:
%       type(symbol, extended-type), where symbol is a predicate, a functor,
%       or a variable.  This is the user-supplied type information.
% PredTypeSpecs:
%       Also a list of type associations, which specifies the types of the
%       predicates defined in Program.  This is the output.

type_inference(Program, TypeEnv, PredTypeSpecs) :-
        collect_predicates(Program, Predicates),
        init_pred_types(Predicates, PredTypeSpecs),
        collect_ineqs(Program, PredTypeSpecs, TypeEnv, [], Inequalities),
        relax_ineqs(Inequalities, yes, Inequalities).

% init_pred_types(+Predicates, -PredTypeSpecs)
% Predicates:
%       A list of predicates.
% PredTypeSpecs:
%       A list of type associations, which gives a most general type to
%       each predicate in Predicates.

init_pred_types([Pred | OtherPreds],
                [type(Pred, MGTypeTuple) | OtherPredTypeSpecs]) :- !,
        assign_mg_pred_type(Pred, MGTypeTuple),
        init_pred_types(OtherPreds, OtherPredTypeSpecs).
init_pred_types([], []).

% collect_ineqs(+Program, +PredTypeSpecs, +TypeEnv, +OldIneqs, -NewIneqs)
% collects the (type) Term Inequalities arising from the conditions of
% well-typing (see Section 5.1), except condition (3) (see Section 5.3),
% from Program, under the type associations PredTypeSpecs and TypeEnv.
% NewIneqs is OldIneqs plus the newly found inequalities.

collect_ineqs([Clause | OtherClauses], PredTypeSpecs, TypeEnv,
              OldIneqs, NewIneqs) :- !,
        collect_variables(Clause, Variables),
        init_var_types(Variables, VarTypeSpecs),
        collect_ineqs_in_clause(Clause, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                                OldIneqs, MidIneqs),
        collect_ineqs(OtherClauses, PredTypeSpecs, TypeEnv,
                      MidIneqs, NewIneqs).
collect_ineqs([], _, _, Ineqs, Ineqs).

% init_var_types(+Variables, -VarTypeSpecs)
% Variables:
%       A list of variables.
% VarTypeSpecs:
%       A list of type associations, which gives a most general type to
%       each variable in Variables.

init_var_types([Var | OtherVars],
               [type(Var, MGVarType) | OtherVarTypeSpecs]) :- !,
        assign_mg_var_type(Var, MGVarType),
        init_var_types(OtherVars, OtherVarTypeSpecs).
init_var_types([], []).

% collect_ineqs_in_clause(+Clause, +PredTypeSpecs, +TypeEnv, +VarTypeSpecs,
%                         +OldIneqs, -NewIneqs)
% does the same work as collect_ineqs/5, except that it works on a Prolog
% Clause.

collect_ineqs_in_clause((Head :- Body), PredTypeSpecs, TypeEnv, VarTypeSpecs,
                        OldIneqs, NewIneqs) :- !,
        collect_ineqs_in_head(Head, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, MidIneqs),
        collect_ineqs_in_body(Body, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              MidIneqs, NewIneqs).
collect_ineqs_in_clause(Clause, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                        OldIneqs, NewIneqs) :-
        literal_of_unit_clause(Literal, Clause),
        collect_ineqs_in_head(Literal, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, NewIneqs).

% collect_ineqs_in_head(+Head, +PredTypeSpecs, +TypeEnv, +VarTypeSpecs,
%                       +OldIneqs, -NewIneqs)
% does the same work as collect_ineqs/5, except that it works on a Prolog
% clause's Head.

collect_ineqs_in_head(Head, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      Ineqs, [(TypeTuple1 >== TypeTuple2) | Ineqs]) :-
        arg_types_of_literal(TypeTuple1, Head, TypeEnv, VarTypeSpecs),
        predicate_of_literal(Pred, Head),
        member(type(Pred, TypeTuple2), PredTypeSpecs).

% collect_ineqs_in_body(+Body, +PredTypeSpecs, +TypeEnv, +VarTypeSpecs,
%                       +OldIneqs, -NewIneqs)
% does the same work as collect_ineqs/5, except that it works on a Prolog
% clause's Body.

collect_ineqs_in_body((Goal1 , Goal2), PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      OldIneqs, NewIneqs) :- !,
        collect_ineqs_in_body(Goal1, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, MidIneqs),
        collect_ineqs_in_body(Goal2, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              MidIneqs, NewIneqs).
collect_ineqs_in_body((Goal1 ; Goal2), PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      OldIneqs, NewIneqs) :- !,
        collect_ineqs_in_body(Goal1, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, MidIneqs),
        collect_ineqs_in_body(Goal2, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              MidIneqs, NewIneqs).
collect_ineqs_in_body((!), _, _, _, Ineqs, Ineqs) :- !.
collect_ineqs_in_body(Goal, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      OldIneqs, NewIneqs) :-
        literal_of_unit_clause(Literal, Goal),
        predicate_of_literal(Pred, Literal),
        collect_ineqs_in_goal(Pred, Literal, PredTypeSpecs, TypeEnv,
                              VarTypeSpecs, OldIneqs, NewIneqs).

% collect_ineqs_in_goal(+Pred, +Goal, +PredTypeSpecs, +TypeEnv, +VarTypeSpecs,
%                       +OldIneqs, -NewIneqs)
% does the same work as collect_ineqs/5, except that it works on a Prolog
% goal literal Goal with predicate Pred.

collect_ineqs_in_goal(Pred, Goal, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      Ineqs, [(TypeTuple1 >== TypeTuple2) | Ineqs]) :-
        member(type(Pred, TypeTuple1), PredTypeSpecs),
        !,
        arg_types_of_literal(TypeTuple2, Goal, TypeEnv, VarTypeSpecs).
collect_ineqs_in_goal(Pred, Goal, _, TypeEnv, VarTypeSpecs,
                      Ineqs, Ineqs) :-
        member(type(Pred, TypeTuple1), TypeEnv),
        arg_types_of_literal(TypeTuple2, Goal, TypeEnv, VarTypeSpecs),
        relax_one_ineq((TypeTuple1 >== TypeTuple2)).

% arg_types_of_literal(-TypeTuple, +Literal, +TypeEnv, +VarTypeSpecs)
% computes TypeTuple, which is the tuple of the most general types associated
% with the arguments of Literal as superscripts (see Section 5.1), under the
% given type associations TypeEnv and VarTypeSpecs.
% Note that we don't need PredTypeSpecs, since the arguments of Literal
% are Prolog terms which contain only functors and variables.

arg_types_of_literal(TypeTuple, Literal, TypeEnv, VarTypeSpecs) :-
        args_of_literal(Args, Literal),
        welltype_term_list(Args, TypeEnv, VarTypeSpecs, ArgTypes),
        args_of_type_tuple(ArgTypes, TypeTuple).

% welltype_term_list(+TermList, +TypeEnv, +VarTypeSpecs, -TypeList)
% applies welltype_term to each member of TermList to generate TypeList.
% See welltype_term/4.
```

```
welltype_term_list([Term | OtherTerms], TypeEnv, VarTypeSpecs,
                   [Type | OtherTypes]) :- !,
        welltype_term(Term, TypeEnv, VarTypeSpecs, Type),
        welltype_term_list(OtherTerms, TypeEnv, VarTypeSpecs, OtherTypes).
welltype_term_list([], _, _, []).

% welltype_term(+Term, +TypeEnv, +VarTypeSpecs, -Type)
% computes Type, which is the most general type associated with (Prolog) Term
% as superscript (see Section 5.1), under the given type associations TypeEnv
% and VarTypeSpecs.

welltype_term('$VAR'(Var), _, VarTypeSpecs, Type) :- !,
        member(type(Var, Type), VarTypeSpecs).
welltype_term(Term, TypeEnv, VarTypeSpecs, Type) :-
        args_of_term(Args, Term),
        welltype_term_list(Args, TypeEnv, VarTypeSpecs, ArgTypes),
        args_of_type_tuple(ArgTypes, TypeTuple),
        functor_of_term(Func, Term),
        member(type(Func, (TypeTuple1 -> Type1)), TypeEnv),
        relax_one_ineq(((TypeTuple1 -> Type1) >== (TypeTuple -> Type))).

% relax_ineqs(+Ineqs, +Flag, +OldIneqs)
% checks and relaxes, if it is unsatisfied, each (type) Term Inequality in
% OldIneqs.  Flag = yes if all inequalities checked so far are satisfied,
% otherwise Flag = no.  After all inequalities are checked (Ineqs = []),
% exit if Flag = yes, otherwise re-run the whole process.

relax_ineqs([Ineq | OtherIneqs], Flag, OldIneqs) :-
        ineq_satisfied(Ineq),
        !,
        relax_ineqs(OtherIneqs, Flag, OldIneqs).
relax_ineqs([Ineq | OtherIneqs], _, OldIneqs) :- !,
        relax_one_ineq(Ineq),
        relax_ineqs(OtherIneqs, no, OldIneqs).
relax_ineqs([], no, OldIneqs) :- !,
        relax_ineqs(OldIneqs, yes, OldIneqs).
relax_ineqs([], yes, _).

% member(?Element, +List)
% succeeds if Element is a member of List, fails otherwise.

member(Element, [Element | _]).
member(Element, [_ | ListTail]) :- member(Element, ListTail).
```

```
%
% FILE:         ineq.pl
% AUTHOR:       Ching-Tsun Chou
% SPEC:         Predicates handling Term Inequalities
%

% ineq_satisfied((Type1 >== Type2))
% succeeds iff Term Inequality (Type1 >== Type2) is satisfied.
% Note that unification with occurs-check is needed here.

ineq_satisfied((Type1 >== Type2)) :-
        numbervars(Type2, 0, _),
        + unify(Type1, Type2),
        !,
        fail.
ineq_satisfied(_).

% relax_one_ineq((Type1 >== Type2))
% relaxes Term Inequality (Type1 >== Type2), if it is relaxable.
% Note that unification with occurs-check is needed here.

relax_one_ineq((Type1 >== Type2)) :-
        copy_term(Type1, Type11),
        unify(Type11, Type2).

% copy_term(+Term1, -Term2)
% copies Term1 to Term2 with all variables renamed.
% Note: This is a hack!

copy_term(Term1, Term2) :-
        assert('$copy_term$'(Term1)),
        '$copy_term$'(Term2),
        retractall('$copy_term$'(_)).
```

```
%
% FILE:         misc.pl
% AUTHOR:       Ching-Tsun Chou
% SPEC:         Miscellaneous predicates used by TYPEINF
%

literal_of_unit_clause(Unit, Unit).

predicate_of_literal((Name/Arity), Literal) :- functor(Literal, Name, Arity).

functor_of_term((Name/Arity), Term) :- functor(Term, Name, Arity).

args_of_literal(Args, Literal) :- Literal =.. [_ | Args].

args_of_term(Args, Term) :- Term =.. [_ | Args].

args_of_type_tuple(ArgTypes, TypeTuple) :- TypeTuple =.. ['' | ArgTypes].

assign_mg_pred_type((_/Arity), MGTypeTuple) :-
        functor(MGTypeTuple, '', Arity).

assign_mg_var_type(_, _).

% collect_variables(+Term, -Vars)
% collects variable names (Vars) occurring in (Prolog) Term.
% Note that variables in Term are marked by functor '$VAR'/1.

collect_variables(Term, Vars) :-
        setof(Var, occurs_in(Var, Term), Vars),
        !.
collect_variables(_, []).

occurs_in(Var, '$VAR'(Var)) :- !.
occurs_in(_, Atom) :- atomic(Atom), !, fail.
occurs_in(Var, Term) :-
        functor(Term, _, Arity),
        occurs_in_args(Arity, Var, Term).

occurs_in_args(0, _, _) :- !, fail.
occurs_in_args(I, Var, Term) :-
        arg(I, Term, Arg),
        occurs_in(Var, Arg).
occurs_in_args(I, Var, Term) :-
        succ(J, I),
        occurs_in_args(J, Var, Term).

% collect_predicates(+Program, -Predicates)
% collects Predicates defined in (Prolog) Program.

collect_predicates(Program, Predicates) :-
        setof(Pred, pred_defined_in_prog(Pred, Program), Predicates),
        !.
collect_predicates(_, []).

pred_defined_in_prog(Pred, [Clause | _]) :-
        pred_defined_by_clause(Pred, Clause).
pred_defined_in_prog(Pred, [_ | OtherClauses]) :-
        pred_defined_in_prog(Pred, OtherClauses).

pred_defined_by_clause(Pred, (Head :- _)) :-
        predicate_of_literal(Pred, Head),
        !.
pred_defined_by_clause(Pred, UnitClause) :-
        predicate_of_literal(Pred, UnitClause).
```

```
%
% FILE:         work.pl
% AUTHOR:       Ching-Tsun Chou
% SPEC:         I/O routines of TYPEINF
%

% work(File) :-
% reads File, which contains Prolog clauses and type declarations, performs
% type inference on the Prolog program in File, and then prints out the
% results (types of predicates defined in File).

work(File) :-
        read_file(File, Program, TypeEnv),
        type_inference(Program, TypeEnv, PredTypeSpecs),
        write_results(PredTypeSpecs).

read_file(File, Program, TypeEnv) :-
        seeing(OldInput, File),
        read_file1(Program, TypeEnv),
        see(OldInput).

read_file1(Program, TypeEnv) :-
        read(Term, NameVarPairs),
        read_terms(Term, NameVarPairs, [], Program, [], TypeEnv).

read_terms(end_of_file, [], Prog, Prog, Env, Env) :- !.
read_terms(Term, NameVarPairs, OldProg, NewProg, OldEnv, NewEnv) :-
        process_term(Term, NameVarPairs, OldProg, MidProg, OldEnv, MidEnv),
        read(NewTerm, NewNameVarPairs),
        read_terms(NewTerm, NewNameVarPairs, MidProg, NewProg, MidEnv, NewEnv).

process_term((:- op(Prec, Type, Name)), _, Prog, Prog, Env, Env) :- !,
        op(Prec, Type, Name).
process_term((:- pred PredDcl), _, Prog, Prog, OldEnv, NewEnv) :- !,
        process_pred_dcl(PredDcl, OldEnv, NewEnv).
process_term((:- type TypeDcl), _, Prog, Prog, OldEnv, NewEnv) :- !,
        process_type_dcl(TypeDcl, OldEnv, NewEnv).
process_term(Clause, NameVarPairs, Prog, [Clause | Prog], Env, Env) :- !,
        make_ground(Clause, NameVarPairs).

process_pred_dcl((PredType , OtherPredTypes), OldEnv, NewEnv) :- !,
        process_pred_dcl1(PredType, OldEnv, MidEnv),
        process_pred_dcl(OtherPredTypes, MidEnv, NewEnv).
process_pred_dcl(PredType, OldEnv, NewEnv) :-
        process_pred_dcl1(PredType, OldEnv, NewEnv).

process_pred_dcl1(PredType, Env, [type(Pred, TypeTuple) | Env]) :-
        predicate_of_literal(Pred, PredType),
        args_of_literal(Args, PredType),
        args_of_type_tuple(Args, TypeTuple).
```

```
process_type_dcl((Type => (FuncType | OtherFuncTypes)), OldEnv, NewEnv) :- !,
        process_type_dcl1(FuncType, Type, OldEnv, MidEnv),
        process_type_dcl((Type => OtherFuncTypes), MidEnv, NewEnv).
process_type_dcl((Type => FuncType), OldEnv, NewEnv) :-
        process_type_dcl1(FuncType, Type, OldEnv, NewEnv).

process_type_dcl1(FuncType, Type,
                  Env, [type(Func, (TypeTuple -> Type)) | Env]) :-
        functor_of_term(Func, FuncType),
        args_of_term(Args, FuncType),
        args_of_type_tuple(Args, TypeTuple).

make_ground(Clause, NameVarPairs) :-
        make_ground1(NameVarPairs),
        numbervars(Clause, 0, _).

make_ground1([(Name = Var) | OtherPairs]) :- !,
        Var = '$VAR'(Name),
        make_ground1(OtherPairs).
make_ground1([]).

write_results(PredTypeSpecs) :-
        result_file(NewOutput),
        !,
        telling(OldOutput, NewOutput),
        write_results1(PredTypeSpecs),
        tell(OldOutput).
write_results(PredTypeSpecs) :-
        write_results1(PredTypeSpecs).

write_results1([PredTypeSpec | OtherPredTypeSpecs]) :- !,
        write_pred_type_spec(PredTypeSpec),
        write_results1(OtherPredTypeSpecs).
write_results1([]).

write_pred_type_spec(type(Pred, TypeTuple)) :-
        predicate_of_literal(Pred, PredType),
        args_of_type_tuple(Args, TypeTuple),
        args_of_literal(Args, PredType),
        numbervars(PredType, 0, _),
        print((:- pred PredType)), write('.'), nl, nl.
```

```
%
% FILE:         op.pl
% AUTHOR:       Ching-Tsun Chou
% SPEC:         Operator declarations for TYPEINF
%

:- op( 800, xfx, '>=='). 	% Used in Term Inequalities.
:- op(1125, xfx, '=>'). 		% Used in type declarations.
```

```
%
% FILE:         load
% AUTHOR:       Ching-Tsun Chou
% SPEC:         Load TYPEINF into memory
%

:- [
       'op.pl'        ,
       'infer.pl'     ,
       'ineq.pl'      ,
       'misc.pl'      ,
       'work.pl'            ].
```

```
%
% FILE:          self
% AUTHOR:        Ching-Tsun Chou
% SPEC:          infer.pl with type declarations added
%                (Test data for TYPEINF)
%

% Type declarations for functors:

:- type list(T) => [] | [T | list(T)].
:- type flag => yes | no.

:- type clause => (literal :- clause)
                | (clause , clause)
                | (clause ; clause)
                | (!).
:- type term => '$VAR'(variable).

:- type type_spec => type(predicate, type_tuple)
                   | type(functor,   functor_type)
                  | type(variable,  type).
:- type functor_type => (type_tuple -> type).

:- type term_ineq => (T >== U).


% Type declarations for predicates which are invoked but not defined in
% infer.pl:

:- pred literal_of_unit_clause(literal, clause).
:- pred predicate_of_literal(predicate, literal).
:- pred functor_of_term(functor, term).
:- pred args_of_literal(list(term), literal).
:- pred args_of_term(list(term), term).
:- pred args_of_type_tuple(list(type), type_tuple).
:- pred assign_mg_pred_type(predicate, type_tuple).
:- pred assign_mg_var_type(variable, type).
:- pred collect_variables(clause, list(variable)).
:- pred collect_predicates(list(clause), list(predicate)).
:- pred ineq_satisfied(term_ineq).
:- pred relax_one_ineq(term_ineq).


% Prolog clauses in infer.pl:

type_inference(Program, TypeEnv, PredTypeSpecs) :-
        collect_predicates(Program, Predicates),
        init_pred_types(Predicates, PredTypeSpecs),
        collect_ineqs(Program, PredTypeSpecs, TypeEnv, [], Inequalities),
        relax_ineqs(Inequalities, yes, Inequalities).
```

```
init_pred_types([Pred | OtherPreds],
                [type(Pred, MGTypeTuple) | OtherPredTypeSpecs]) :- !,
        assign_mg_pred_type(Pred, MGTypeTuple),
        init_pred_types(OtherPreds, OtherPredTypeSpecs).
init_pred_types([], []).

collect_ineqs([Clause | OtherClauses], PredTypeSpecs, TypeEnv,
              OldIneqs, NewIneqs) :- !,
        collect_variables(Clause, Variables),
        init_var_types(Variables, VarTypeSpecs),
        collect_ineqs_in_clause(Clause, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                                OldIneqs, MidIneqs),
        collect_ineqs(OtherClauses, PredTypeSpecs, TypeEnv,
                      MidIneqs, NewIneqs).
collect_ineqs([], _, _, Ineqs, Ineqs).

init_var_types([Var | OtherVars],
               [type(Var, MGVarType) | OtherVarTypeSpecs]) :- !,
        assign_mg_var_type(Var, MGVarType),
        init_var_types(OtherVars, OtherVarTypeSpecs).
init_var_types([], []).

collect_ineqs_in_clause((Head :- Body), PredTypeSpecs, TypeEnv, VarTypeSpecs,
                        OldIneqs, NewIneqs) :- !,
        collect_ineqs_in_head(Head, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, MidIneqs),
        collect_ineqs_in_body(Body, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              MidIneqs, NewIneqs).
collect_ineqs_in_clause(Clause, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                        OldIneqs, NewIneqs) :-
        literal_of_unit_clause(Literal, Clause),
        collect_ineqs_in_head(Literal, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, NewIneqs).

collect_ineqs_in_head(Head, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      Ineqs, [(TypeTuple1 >== TypeTuple2) | Ineqs]) :-
        arg_types_of_literal(TypeTuple1, Head, TypeEnv, VarTypeSpecs),
        predicate_of_literal(Pred, Head),
        member(type(Pred, TypeTuple2), PredTypeSpecs).

collect_ineqs_in_body((Goal1 , Goal2), PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      OldIneqs, NewIneqs) :- !,
        collect_ineqs_in_body(Goal1, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, MidIneqs),
        collect_ineqs_in_body(Goal2, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              MidIneqs, NewIneqs).
collect_ineqs_in_body((Goal1 ; Goal2), PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      OldIneqs, NewIneqs) :- !,
        collect_ineqs_in_body(Goal1, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              OldIneqs, MidIneqs),
        collect_ineqs_in_body(Goal2, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                              MidIneqs, NewIneqs).
collect_ineqs_in_body((!), _, _, _, Ineqs, Ineqs) :- !.
collect_ineqs_in_body(Goal, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      OldIneqs, NewIneqs) :-
        literal_of_unit_clause(Literal, Goal),
        predicate_of_literal(Pred, Literal),
        collect_ineqs_in_goal(Pred, Literal, PredTypeSpecs, TypeEnv,
                              VarTypeSpecs, OldIneqs, NewIneqs).

collect_ineqs_in_goal(Pred, Goal, PredTypeSpecs, TypeEnv, VarTypeSpecs,
                      Ineqs, [(TypeTuple1 >== TypeTuple2) | Ineqs]) :-
        member(type(Pred, TypeTuple1), PredTypeSpecs),
        !,
        arg_types_of_literal(TypeTuple2, Goal, TypeEnv, VarTypeSpecs).
collect_ineqs_in_goal(Pred, Goal, _, TypeEnv, VarTypeSpecs,
                      Ineqs, Ineqs) :-
        member(type(Pred, TypeTuple1), TypeEnv),
        arg_types_of_literal(TypeTuple2, Goal, TypeEnv, VarTypeSpecs),
        relax_one_ineq((TypeTuple1 >== TypeTuple2)).

arg_types_of_literal(TypeTuple, Literal, TypeEnv, VarTypeSpecs) :-
        args_of_literal(Args, Literal),
        welltype_term_list(Args, TypeEnv, VarTypeSpecs, ArgTypes),
        args_of_type_tuple(ArgTypes, TypeTuple).

welltype_term_list([Term | OtherTerms], TypeEnv, VarTypeSpecs,
                   [Type | OtherTypes]) :- !,
        welltype_term(Term, TypeEnv, VarTypeSpecs, Type),
        welltype_term_list(OtherTerms, TypeEnv, VarTypeSpecs, OtherTypes).
welltype_term_list([], _, _, []).

welltype_term('$VAR'(Var), _, VarTypeSpecs, Type) :- !,
        member(type(Var, Type), VarTypeSpecs).
welltype_term(Term, TypeEnv, VarTypeSpecs, Type) :-
        args_of_term(Args, Term),
        welltype_term_list(Args, TypeEnv, VarTypeSpecs, ArgTypes),
        args_of_type_tuple(ArgTypes, TypeTuple),
        functor_of_term(Func, Term),
        member(type(Func, (TypeTuple1 -> Type1)), TypeEnv),
        relax_one_ineq(((TypeTuple1 -> Type1) >== (TypeTuple -> Type))).

relax_ineqs([Ineq | OtherIneqs], Flag, OldIneqs) :-
        ineq_satisfied(Ineq),
        !,
        relax_ineqs(OtherIneqs, Flag, OldIneqs).
relax_ineqs([Ineq | OtherIneqs], _, OldIneqs) :- !,
        relax_one_ineq(Ineq),
        relax_ineqs(OtherIneqs, no, OldIneqs).
relax_ineqs([], no, OldIneqs) :- !,
        relax_ineqs(OldIneqs, yes, OldIneqs).
relax_ineqs([], yes, _).
```

```
member(Element, [Element | _]).
member(Element, [_ | ListTail]) :- member(Element, ListTail).
```