PARTIAL ORDER PROGRAMMING

D. Stott Parker

December 1987
CSD-870067

Partial Order Programming

D. Stott Parker

Computer Science Department
University of California
Los Angeles, CA 90024-1596

ABSTRACT

We introduce a programming paradigm in which statements are constraints over partial orders. A *partial order programming problem* has the form

$$\begin{array}{ll} \text{minimize} & u \\ \text{subject to} & u_1 \sqsupseteq v_1,\ u_2 \sqsupseteq v_2,\ \cdots \end{array}$$

where u is the *goal,* and $u_1 \sqsupseteq v_1,\ u_2 \sqsupseteq v_2,\ \cdots$ is a collection of constraints called the *program.* A solution of the problem is a minimal value for u determined by values for u_1, v_1, etc. satisfying the constraints. The domain of values here is a *partial order,* a domain D with ordering relation $\sqsupseteq$.

The partial order programming paradigm has interesting properties:

(1) It generalizes mathematical programming, dynamic programming, and computer programming paradigms (logic, functional, and others) cleanly, and offers a foundation both for studying and combining paradigms.

(2) It takes thorough advantage of known results for continuous functionals on complete partial orders, when the constraints involve expressions using only continuous and monotone operators. These programs have an elegant semantics coinciding with recent results on the relaxation solution method for constraint problems.

(3) It presents a framework that may be effective in modeling of complex systems, and in knowledge representation for cognitive computation problems.

This work supported by a State of California – IBM Los Angeles Scientific Center MICRO grant, "RAPPORT: Relaxation and Pattern-Oriented Rule Programming", and DARPA contracts MDA-903-82-C-0189 and F29601-87-K-0072.

Table of Contents

Partial Order Programming

D. Stott Parker

Computer Science Department
University of California
Los Angeles, CA 90024-1596

1. Introduction

This work began as an attempt to design a formal framework for modeling complex real-world systems such as software environments and computer systems. Over time part of the design has grown into a theory of its own. This monograph summarizes the theory in its current state. To provide some perspective on the philosophy behind the theory, we begin with some speculation on the nature of modeling.

1.1. Paradigms for Modeling Complex Systems

The basic issue in modeling complex systems is the identification of an appropriate *paradigm* for expressing the objects one knows about and how they behave. A paradigm is a language with well-specified semantics, along with conventions for effective usage. There are many paradigms today, including knowledge representation paradigms (frames and rules, constraint satisfaction, production systems), computer programming paradigms (functional, logic, object-oriented), and mathematical programming paradigms (linear and nonlinear programming). None of these is ideal for modeling truly complex systems.

Recently a number of ingenious paradigms have been introduced that offer alternatives here. They include the type-oriented extensions of logic programming LOGIN, LeFun, and LIFE of Ait-Kaci, Nasr, and Lincoln [2, 3, 4, 5] with type hierarchies and functions; the Constraint Logic Programming scheme of Jaffar et al [47, 48], extending logic programming with general mechanisms for delayed satisfaction of constraints, particularly linear equality and inequality constraints in *CLP*(R); and the iterative UNITY paradigm of Chandy and Misra [22] which generalizes spreadsheet calculations into collections of assignment statements executed in parallel. Each will have different strengths in different modeling applications.

In our view, "modeling" and "knowledge representation" are essentially synonymous, and much of the initial motivation behind our design came from problems that are commonly recognized as knowledge representation problems. Currently the field of knowledge representation lacks a theoretical foundation. Different issues in knowledge representation apparently require very different solutions, and few important issues have definitive solutions. Important issues include: which paradigms are appropriate for which problems, how paradigms can be combined, how reasoning can be controlled, how parallelism can be exploited, and how multi-level models can be managed.

The impetus behind this work comes from conviction in the following thesis:

> A paradigm for stating and solving *ordering constraints* is effective in modeling complex systems.

By ordering constraints we mean here relationships of comparison or inequality – binary relationships that are transitive and reflexive. These relations are known as *preorders,* and when further required to be antisymmetric (acyclic), they are called *partial orders.*

Partial order programming is a paradigm in which information is expressed as ordering constraints over a predefined partial order. It is the use of partial orders that permits this paradigm to model the rich structure of complex systems. Real-valued (totally ordered) parameters used in many models are less flexible than parameters that range over partial orders such as type hierarchies and intervals of values, not to mention vectors, functions, sets, lists, algebraic expressions, or even complex numbers. After justifying our belief in the importance of ordering and the related concept of monotonicity, we will provide a more careful definition of this paradigm.

1.2. Ordering as a Basis for Modeling

Ordering is basic in human knowledge representation. First of all, ordering underlies type hierarchies and generalization or abstraction, such as with the following incomplete taxonomy of animals:

vertebrate	$\subseteq$	animal
gnathostomata	$\subseteq$	vertebrate
pisces	$\subseteq$	gnathostomata
shark	$\subseteq$	pisces
requiem shark	$\subseteq$	shark
hammerhead shark	$\subseteq$	shark
whale shark	$\subseteq$	shark.

In this taxonomy $\subseteq$ is the partial order of inclusion or containment. Ordering also underlies composition, part-of relationships and aggregation, spatial relationships, temporal relationships, dependencies, causal relationships, transfers of possession, strength of conviction, preference, utility, planning, procedures, reductive problem-solving, many modes of inference, chains of reasoning, and heuristics. In short, humans are very good at reasoning about ordering.

Some have argued that the role of set theory in mathematics suggests that ordering is not important, since sets are unordered. This is not true: set theory immediately introduces the partial ordering of containment ($\subseteq$) just mentioned. Also, consider the following remark of the eminent logician Abraham Fraenkel:

> From a psychological viewpoint, there can be no doubt that somehow the *ordered* set is the primary notion, yielding the plain notion of set or aggregate by an *act of abstraction*, as though one jumbled together the elements which originally appear in a definite succession. As a matter of fact, our senses offer the various objects or ideas in a certain spatial order or temporal succession. When we want

> to represent the elements of an originally non-ordered set, say the inhabitants of Washington, D.C., by script or language, it cannot be done but in a definite order [38].

In fact, the notation $\overline{\overline{S}}$ representing the cardinal number of the set S was developed by Cantor as a double abstraction, "the general concept which with the aid of our active intelligence results from a set when we abstract from the nature of its various elements and from the order of their being given" [53].

In his recent book [67], Minsky identifies many useful orderings used by the mind – towers, bridges, chains, levels, etc. He also points out that "much of ordinary thought is based on recognizing differences ... our senses react mainly to how things change in time." Noticing differences, and differences in differences, invites comparisons and ordering depending on what kinds of difference are important in context (§23).

Partial ordering necessarily arises when two objects differ in more than one aspect; clearly if one object does not dominate the other completely, the two cannot be directly compared or ordered except by individual aspects. In fact taxonomies like the shark taxonomy above arise as an organized record of differences – as "discrimination nets".

Further evidence of the importance of ordering is its role in natural language. At least western languages make basic use of partial ordering in descriptions and ordinary statements. Common nouns denote types, and adjectives typically are restrictions on types. A description is a noun phrase denoting a *subtype* of the type of its noun. Furthermore, prepositions usually define spatial relationships that entail ordering, and ordering is implicit in most uses of the verbs "to be" (is a, generalization) and "to have" (has a, part of, of, role chains, aggregation). It is not surprising that type hierarchies are natural to western programmers.

Systems providing type hierarchies with the orderings suggested here can be used for many central knowledge representation problems. For example, consider the following inference from such a system:

shark & non(whale shark)	$\subseteq$	carnivore
maneater	=	big & (eats flesh)
non(eats flesh)	$\subseteq$	non(carnivore)
big & (shark & non(whale shark))	$\subseteq$	maneater.

This inference derives that big sharks are maneaters, except for whale sharks, which it turns out subsist on plankton. Keenan and Faltz [50] give a semantics for natural language in terms of Boolean algebras, Ait-Kaci and Nasr [3] use a distributive lattice of types with attributes to generalize first-order terms, and Attardi and Simi [10] use a lattice of descriptions in the Omega knowledge representation system. Lattices and Boolean algebras are, of course, well-behaved partial orders.

1.3. Monotonicity

Given the importance of ordering, it is natural to study operators that preserve ordering. We say a function f is *order-preserving,* or *monotone,* if $x \leq y$ implies $f(x) \leq f(y)$. Given a set S with some ordering relationships, the image $f(S)$ will respect these ordering relationships, and possibly others. Viewed differently, f is monotone if giving it a larger input cannot get a smaller output, i.e., f does not give out less if we put in more.

Monotone operators appears frequently, for example, in natural language. Determiners (a, all, every, most, several, some) are often monotone on types. That is, determiners preserve implication when applied to two phrases one of whose denotations implies the other's [80]. Because "sharks eat quickly" implies "sharks eat", "several sharks eat quickly" implies "several sharks eat".

A deep topic that we will touch upon later in this work is the connection between monotonicity, ordering, and computation. Given the ordering

$$\begin{array}{lll} x & \geq & y \\ y & \geq & z \\ z & \geq & 3 \end{array}$$

it is easy to see that both x and y are least 3. The computational percolation required by this inference also underlies procedural concepts such as spreading activation, constraint propagation, hill-climbing, etc. All these methods iteratively propagate information along "chains" of ordering constraints. Percolation is a monotone process: percolation through constraints results in a monotone increase in information about variable values.

This percolation can be generalized to what is called *relaxation* by some researchers. Relaxation takes a set of constraints like the above and an arbitrary initial set of assignments of values to variables. Some of the constraints will usually be out-of-kilter, i.e., unsatisfied. Relaxation then repeatedly "relaxes the tension" of any out-of-kilter constraints by updating value assignments to satisfy these constraints. Although some satisfied constraints may become unsatisfied by this updating, in many cases the process will eventually arrive at an assignment under which all constraints are in-kilter. We will discuss relaxation later, but among other things it has also been applied in natural language processing. Mellish describes a constraint satisfaction framework for natural language processing that he calls 'incremental semantic interpretation' [65]. Essentially his system uses relaxation in resolving constraints that arise in semantic analysis of noun phrases.

Monotonicity is at the heart of of iterative, hill-climbing approaches for solving inequality constraints like relaxation. Suppose we are given the constraint

$$u \geq f(u)$$

when f is monotone, and wish to find the least u that satisfies this inequality. This sort of feedback control relationship arises frequently in survival, such as with

$$income \geq lifestyle\_requirements(\ income\ ).$$

If we have an initial value u_0 such that $u_0 \leq f(u_0)$, applying monotonicity repeatedly tells us that

$$\begin{array}{ccc} u_0 & \leq & f(u_0) \\ f(u_0) & \leq & f(f(u_0)) \\ f(f(u_0)) & \leq & f(f(f(u_0))) \end{array}$$

so the sequence $u_0, f(u_0), f(f(u_0)), \ldots$ will increase, hopefuly reaching a solution of the constraint $u \geq f(u)$.

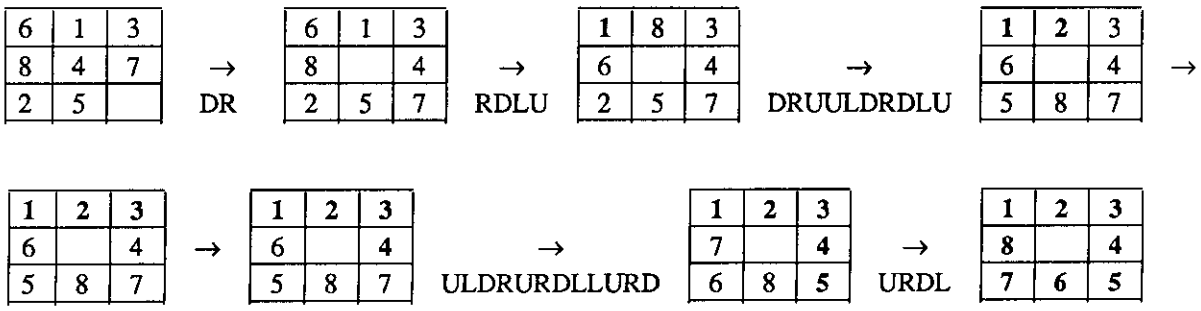
We must know more about f to *prove* that this sequence will reach a solution. One common condition that turns out to be sufficient is that f also be *continuous,* in the sense that it preserves limits of ascending sequences. That is, $\lim_k f(u_k) = f(\lim_k u_k)$, so that a graph of the function can be drawn without lifting pencil from the paper. We will see later that if f is continuous and monotone, the sequence above must have a limit – a *fixed point* u such that $u = f(u)$. With continuous monotone constraints, then, we are not only no worse off by iteratively replacing our current value u with $f(u)$, but also guaranteed eventually to satisfy our constraints. Moreover, if we stop iteration prematurely we still have an *approximation* to a solution.

Humans *like* monotonicity, and sometimes go out of their way to make life monotone even though this behavior may be inefficient. To back up this generality, let us consider the well-known eight puzzle. The problem is to take an initial configuration such as

6	1	3
8	4	7
2	5	

and change it with moves of the "hole" shown here in the lower right corner to a configuration in which the tiles are placed in order around the periphery.

Although many heuristics are possible for the eight puzzle, a common human heuristic is to impose a macro-structuring in which the tiles are placed in the right positions in order. This heuristic is monotone. It gradually improves the configuration above as follows [54]:



When viewed at the level of the macro-steps shown here, the puzzle improves monotonically in the ordering of "number of tiles in the right place". In general, when given a problem goal it is not obvious that we can find a collection of easily-stated

subgoals along with operators that monotonically achieve these subgoals, but for the eight puzzle we can. In [54], Korf proves this assertion by producing a complete set of *macro operators* which permit monotone solution of the puzzle. In GPS terminology, the operators reduce of the goal of reordering the eight puzzle to "serializable subgoals" for individual tiles. As the example here suggests, the solutions obtained with these operators are not necessarily efficient, but the simplification they provide make them very useful. Korf also gives macro operators for Rubik's cube and other puzzles where their existence is not at all obvious.

Monotone operators have important properties. First, their composition is monotone. As a consequence sums, products, etc. of monotone operators are monotone. Second, to minimize (resp. maximize) the result of a monotone operator, one must minimize (resp. maximize) the inputs to the operator. This is Bellman's famous "principle of optimality", the cornerstone of dynamic programming [39]. Thus monotone operators are also easy to reason about, and an agent possessing only monotone operators and monotone models of the world is in a consistent position. In fact, we can show that monotonicity and ordering lie at the heart of logic and inference.

1.4. Logic and Ordering

Logic and ordering are directly connected: *every logical implication is a statement of ordering*. The implication $P \rightarrow Q$ is identical to the constraint "truth(P)" $\leq$ "truth(Q)", an assertion about ordering on truth values or validity. Thus simple logical implications are inequalities over the partial order with domain {**true**,**false**}. The ordering relationship $\rightarrow$ on this domain is given by the familiar truth table with **false** $\rightarrow$ **true**. Furthermore, the chain of logical implications

$$\begin{array}{lcl} P & \rightarrow & Q \\ \underline{Q} & \underline{\rightarrow} & \underline{R} \\ P & \rightarrow & R \end{array}$$

can be seen as just a result of transitivity on inequalities.

Analogously, first-order (clausal or nonclausal) inference systems that use implication as their basis are, in a very concrete sense, systems for chain reasoning about inequalities. Inference rules used in automated deduction systems are then applications of partial ordering axioms and properties of monotonicity. The resolution rule

$$\begin{array}{r} G \vee P \\ \underline{\neg P \vee H} \\ G \vee H \end{array}$$

(if G or P, and either not P or H, then G or H) is equivalent to

$$\begin{array}{lcl} G \vee P & \sqsupseteq & \textbf{true} \\ \underline{H} & \underline{\sqsupseteq} & \underline{P} \\ G \vee H & \sqsupseteq & \textbf{true}, \end{array}$$

where we write $\leftarrow$ as $\sqsupseteq$. In other words, we can substitute H for P since it is at least as valid. The rule is sound because $\vee$ is monotone in its arguments.

Induction is also just the combination of logical ordering with monotonicity of predicates. The principle of mathematical induction

$$\forall P\ [\ P(0)\ \wedge\ [\ \forall n\ \ P(n) \rightarrow P(n{+}1)\ ]\ ]\ \ \rightarrow\ \ [\ \forall n\ P(n)\ ]$$

can be reexpressed as

$$\forall P\ [\ [\ P(0) \sqsupseteq \mathbf{true}\ ]\ \wedge\ [\ P\ \textit{monotone}\ ]\ ]\ \ \rightarrow\ \ [\ \forall n\ P(n) \sqsupseteq \mathbf{true}\ ].$$

For computational induction and structural induction [60], the same observation holds.

Readers interested in automated reasoning may be surprised to find some very involved inference rules result from ordering and monotonicity. Manna and Waldinger's various replacement rules [64] make direct use of ordering and monotonicity. Also, consider the very general nonclausal *nested resolution* rule recently developed by Traugott [91]: Let $G[P]$ be a ground first-order formula G that contains one or more occurrences of the subformula P, and $F[P^+, P^-]$ be a ground formula with zero or more occurrences of P, where P^+ and P^- represent respectively the occurrences under an even and odd number of negations. Therefore F is monotone increasing in P^+, and monotone decreasing in P^-. Nested resolution is the following (surprising) rule:

$$\frac{\begin{array}{l} F[\ P^+, P^-\ ] \\ G[\ P\ ] \end{array}}{F[\ G[\ \mathbf{true}\ ], \neg G[\ \mathbf{false}\ ]\ ].}$$

Soundness of the rule follows from ordering and monotonicity. To see this, first rewrite it as:

$$\frac{\begin{array}{rcl} F[\ P^+, P^-\ ] & \sqsupseteq & \mathbf{true} \\ G[\ P\ ] & \sqsupseteq & \mathbf{true} \end{array}}{F[\ G[\ \mathbf{true}\ ], \neg G[\ \mathbf{false}\ ]\ ] \ \sqsupseteq\ \mathbf{true}.}$$

Note that $G[P] \sqsupseteq \mathbf{true}$ implies $G[\mathbf{true}] \sqsupseteq P \sqsupseteq \neg G[\mathbf{false}]$ whether P is **true** or **false**. Therefore, the rule can be proved by using monotonicity twice and then the first antecedent:

$$F[G[\mathbf{true}],\neg G[\mathbf{false}]]\ \sqsupseteq\ F[P,\neg G[\mathbf{false}]]\ \sqsupseteq\ F[P,P]\ =\ F[P^+,P^-]\ \sqsupseteq\ \mathbf{true}.$$

1.5. Partial Order Programming

The speculation above about ordering, monotonicity, and logic can be put together to give a formal framework for structuring complex models such as expert systems. Clancey's *classification problem solving* methodology [24] is used to classify objects with certain features in a type hierarchy. For example, we identify a large cartilaginous animal with jaws and a backbone in a water habitat as a shark, using constraints on members of the hierarchy (e.g., gnathostomatae have jaws). Clancey argues that "a broad range of heuristic programs – embracing forms of diagnosis, catalog selection, and skeletal planning – ... have a characteristic inference structure that systematically

relates data to a preenumerated set of solutions by abstraction, heuristic association, and refinement.'' This structure is represented by ten significant expert systems, including MYCIN, SACON, The Drilling Advisor, GRUNDY, and SOPHIE III. Classification is a process of assigning a value to an object from a well-established partial order of stereotypes, in a way that satisfies constraints.

Consistent with this perspective on expert systems, we feel diagnostic problems can be put in the form

find the most specific	*cause*
subject to	*cause* $\rightarrow$ *observed effects*

and more generally classification problems can be written as

find the most specific	*classification*
subject to	*classification* $\rightarrow$ *properties*
	properties $\rightarrow$ *observed properties.*

We formalize this format for stating problems as *partial order programming*. Partial order programming is a computational modeling paradigm that expresses both computation and relationships declaratively as statements of ordering. Every partial order programming problem has the form

minimize	G
subject to	P

where G is the *goal,* and P is a collection of constraints called the *program.* A *solution* of the problem is a value for G that satisfies the constraints P.

This much looks like ordinary mathematical programming. The new twist of partial order programming is that *the domain of values is a partial order,* a domain D with ordering relation $\sqsupseteq$. By contrast, mathematical programming requires D to be totally ordered. We require each constraint to be of the form

$$u \sqsupseteq v$$

where u and v are *objects*. The goal G is also an object.

Thus, in addition to the goal G the program P specifies:

(1) a domain D of *values* with partial order $\sqsupseteq$;

(2) a collection of ordering constraints C;

(3) a collection B of *objects*.

Semantics of the program are assignments of values in D to the objects u in B that satisfy the constraints of C. Specifically, we are interested in those semantics giving the *least* possible values to the object G.

Consider the following example of a partial order program, where the domain of values D is the collection of all subsets of {1,2,3} and is partially ordered by inclusion ($\supseteq$) with least element $\emptyset$:

minimize	s		
subject to	s	$\supseteq$	t
	s	$\supseteq$	{3}
	t	$\supseteq$	{1,2}.

In this problem the objects B are $\{s,t\} \cup 2^D$, and the inequalities listed here give the ordering constraints C. The unique solution of the problem is determined by the semantics

$$\begin{aligned} s &= \{1,2,3\} \\ t &= \{1,2\}, \end{aligned}$$

which gives the smallest possible value to s that is consistent with the constraints.

We will make this definition more formal later. To help build intuition about how useful partial order programming can be, the next section first presents a number of examples illustrating its application. However, we should assert here that the definition is very general. For example, we can let the objects B be *expressions* using a fixed set of operators and variables:

minimize	s		
subject to	s	$\supseteq$	$t \cup \{3\}$
	t	$\supseteq$	{1,2}.

When B contains expressions as it does in this problem, we obtain what we call an *algebraic* partial order programming problem (the operators and variables determine an algebra). When, furthermore, all the operators are continuous and monotone (such as $\cup$ is), we obtain a *continuous monotone partial order programming problem.* As we suggested above, these problems arise not infrequently, and have many nice properties.

The example above suggests partial order programming is like mathematical programming. Actually, it covers certain forms of computer programming as well. For example, with the problem

minimize	$f(1)$		
subject to	$f(X)$	$\sqsupseteq$	$2 * g(X)$
	$g(Y)$	$\sqsupseteq$	$Y + 1$

we expect a solution that assigns the goal object $f(1)$ the value $2 * (1 + 1)$. Later we will see that this expectation can be met.

1.6. Foreword

This monograph covers a lot of varied terrain at a quick pace. Each part of the terrain contributes part of the whole picture, and all parts of the picture are, at least as of this writing, important. Before we begin, let us take a few moments to summarize in a large way what the picture is.

Partial order programming is interesting for several reasons:

- It connects diverse fields in a natural way, including fields of both symbolic and numeric computation. Partial order programming generalizes a variety of computational paradigms: logic, functional, and others. We will explain how later, but basically established computational notions such as logical derivation, lambda reduction, and so forth can all be expressed with partial orders (or preorders). In addition it resembles constraint-satisfaction paradigms from operations research, particularly mathematical programming.
- There is a natural class of partial order programming problems that is easily solved by relaxation iterations: the *continuous monotone* partial order programming problems. Semantics for continuous monotone programs are especially elegant, since the programs take thorough advantage of known results for continuous functionals on complete partial orders. In fact, these programs give different perspectives on the use of these results. Where this theory has been applied to the problem of defining semantics of programming languages, the partial order typically used is that of approximation, where one program approximates another if its denotation is subsumed by the other's [69, 92]. Continuous monotone partial order programming encourages the use of other partial orders: numerical comparison, type inclusion, etc. Also, the connection with relaxation and the fundamental role of partial orders in concurrency (e.g., in formalizing data and control dependencies) suggest these programs may have parallel computing applications.
- Partial order programming offers a computational framework that has desirable properties for dealing with cognitive problems, including natural language processing and knowledge representation. Its paradigm of relating constrained structures (objects) to well-understood structures (values) via a process of constraint satisfaction appears to coincide with concepts at the center of the knowledge representation problem: typing, approximation (belief, certainty, confidence, utility, etc.), and more generally abstraction.

Our concern here is to present basic results, in order to set a foundation for implementations of partial order programming. Throughout we have tried to keep the presentation simple and self-contained, in the belief that the only successful programming paradigms are simple ones. After we define partial order programming rigorously, we study a number of examples of the paradigm. We then investigate its relationship with linearity, showing how continuous monotone programming problems sometimes can be cast in the format of linear algebraic systems, and then solved by relaxation or elimination methods. Finally we relate continuous monotone partial order programming to existing computer programming paradigms, specifically logic and functional programming.

2. Complete Partial Orders and Fixed Point Theorems

This section reviews basic results in the theory of partial orders. A mild warning! The presentation here essentially follows current conventions in denotational semantics [82]. However the reader should be aware that published treatments of this material, even in the literature on denotational semantics, usually differ in some way. First, some references base their presentation on complete lattices, while others use (differing definitions of) complete partial orders. Second, older results are attributed to different people by different authors. Third, conventions are sometimes tacit. For example, a complete partial order sometimes tacitly contains a least element $\perp$, and in denotational semantics "continuity" tacitly implies "monotonicity".

For the sake of clarity, we use the term "*continuous monotone*" below. (Readers comfortable with denotational semantics should read this as "continuous", since there monotonicity is implicit in the definition of continuity.) This perhaps cumbersome notation states both requirements on functions sufficient to achieve fixed points iteratively as in Theorem 1. In general, however, continuity does *not* imply monotonicity: real-valued continuous functions are not necessarily monotone, for example. Some of the applications we illustrate later involve real-valued functions.

The fixed point results here are only the most basic. For example, a vast literature exists on fixed point theory of functions that are continuous but not monotone. Brouwer's famous theorem proves the existence of a fixed point for such functions, but does not show how to produce one; consequently many numerical algorithms for calculating fixed points have been developed [81, 85]. Partial order programming should be extended eventually to draw on other fixed point results.

In view of the potential for confusion, we have tried to keep the presentation simple and self-contained. Proofs of theorems are included in the appendix. See [12, 82, 87, 88] for clear alternative presentations.

2.1. Complete Partial Orders and Complete Lattices

Definition

A *partial order* is a pair $<D,\sqsupseteq>$ where $\sqsupseteq$ is a binary relation on D such that

(P1) For all x in D, $x \sqsupseteq x$.
(P2) For all x,y,z in D, if $x \sqsupseteq y$ and $y \sqsupseteq z$, then $x \sqsupseteq z$.
(P3) For all x,y in D, if $x \sqsupseteq y$ and $y \sqsupseteq x$, then $x = y$.

A *preorder* is a pair $<D,\sqsupseteq>$ satisfying (P1) and (P2).

The least element of a partial order, if one exists, is written $\perp$; the greatest element, if one exists, is written $\top$. When they exist, we have:

(C1) For all x in D, $x \sqsupseteq \perp$.
(C2) For all x in D, $\top \sqsupseteq x$.

Definition

A *total order* $<D,\sqsupseteq>$ is a partial order in which for every pair of elements x,y either $x \sqsupseteq y$ or $y \sqsupseteq x$ holds.

Definition
ω is the set of natural numbers.

Definition
An *ascending chain* S in a partial order $<D,\sqsupseteq>$ is a sequence $S = \{ x_k \mid k \in \omega \}$ with

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$$

It is a totally ordered subset of D.

Definition
A *directed set* S in a partial order $<D,\sqsupseteq>$ is a nonempty subset of D with the property that if x,y are arbitrary elements in S, then there is another element z in S such that both $z \sqsupseteq x$ and $z \sqsupseteq y$.

Directed sets include ascending chains as a special case.

Definition
In a partial order $<D,\sqsupseteq>$, an *upper bound* of a subset S of D is an element z in D such that for every x in S, $z \sqsupseteq x$.

The *least upper bound* of a set S, written

$$\bigsqcup S,$$

is the least z such that z is an upper bound of S. By P3, it is unique if it exists.

Definition
A *complete partial order* (*cpo*) is a partial order $<D,\sqsupseteq>$ in which every directed set S of D has a least upper bound $\bigsqcup S$. A *pointed cpo* is a cpo with $\bot$.

This definition is interesting because it requires infinite directed sets to have least upper bounds; every finite directed set must already contain its own least upper bound. As an example, the partial order of real numbers in the open interval $D = (0,1)$ ordered by $\geq$ is *not* a complete partial order, since the infinite directed set $S = \{1 - 2^k \mid k > 0\}$ has no least upper bound in D. This partial order is also not pointed, since it has no least element in D. Changing the domain D to the closed interval $[0,1]$ ordered by $\geq$ makes it a pointed cpo.

Definition
A *complete lattice* is a partial order $<D,\sqsupseteq>$ such that *every* subset S of D has a least upper bound $\bigsqcup S$. In particular $\bot \in D$, since $\bot = \bigsqcup \varnothing$.

Complete lattices are sometimes defined as partial orders in which $\bigsqcup S$ and $\bigsqcap S$ exist for every set S. Because $\bigsqcap S = \bigsqcup \{ x \mid y \sqsupseteq x$, for every y in $S \}$, the definition here is equivalent.

Given a collection of cpo s $<D_1,\sqsubseteq_1>$, ..., $<D_n,\sqsubseteq_n>$ we can construct further cpo s:

(1) if $D_i \cap D_j = \{\bot\}$ for all $i \neq j$, the *direct sum*

$$<D,\sqsubseteq> \quad = \quad <D_1,\sqsubseteq_1> + \ ... \ + <D_n,\sqsubseteq_n>$$

is a cpo, in which $D = D_1 \cup \cdots \cup D_n$, and $x \sqsubseteq y$ iff $x \sqsubseteq_i y$ for some i.

(2) the *direct product*

$$<D,\sqsubseteq> \quad = \quad <D_1,\sqsubseteq_1> \times \ ... \ \times <D_n,\sqsubseteq_n>$$

is a cpo, in which $D = D_1 \times \cdots \times D_n$, and $<x_1, \ldots ,x_n> \sqsubseteq <y_1, \ldots ,y_n>$ iff $x_i \sqsubseteq y_i$ for $1 \leq i \leq n$. The least element of D is written

$$\bot = <\bot, \ldots ,\bot>.$$

2.2. Continuous Monotone Functions and Fixed Points

Definition

A function on a partial order $f : D \rightarrow D$ is *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

Definition

The *composition* $f \bullet g$ of two functions f and g is defined by $(f \bullet g)(x) = f(\ g(x)\ )$.

Thus $\bullet : ((D \rightarrow D) \times (D \rightarrow D)) \rightarrow (D \rightarrow D)$. We also permit $\bullet$ to serve as functional application, where $\bullet : ((D \rightarrow D) \times D) \rightarrow D$. In this case, $f \bullet x = f(\ x\ )$.

We introduce this notation instead of using the more common $f \circ g = g(f)$ so that functional composition can be written left-to-right. This will simplify notation later when we view functional composition as a kind of "multiplication" operator.

Note that any composition of monotone functions is monotone.

If S is a directed set and f is monotone, then $f(S) = \{\ f(x) \mid x \in S\ \}$ is a directed set. In particular, f maps ascending chains into ascending chains.

When f is monotone and we have the converse property that $f(x) \sqsubseteq f(y)$ implies $x \sqsubseteq y$, f is called an *order embedding*, and S and $f(S)$ are *order isomorphic*.

Definition

A function $f : D \rightarrow D$ on a cpo $<D,\sqsubseteq>$ is called *continuous monotone* if for every directed set S in D,

$$f(\bigsqcup S) = \bigsqcup f(S) .$$

This definition says that for an infinite ascending chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ we have the equality

$$f(\bigsqcup_{k \in \omega} x_k) = f(\bigsqcup \{x_k \mid k \in \omega\}) = \bigsqcup \{f(x_k) \mid k \in \omega\} = \bigsqcup_{k \in \omega} f(x_k).$$

In other words f preserves the limits of chains. This parallels the definition of continuity of real-valued functions, but differs in considering only sequences that are ascending chains.

Note that every continuous monotone function must also be monotone, since if $x \sqsupseteq y$,

$$f(x) \;=\; f(\bigsqcup\{x,y\}) \;=\; \bigsqcup f(\{x,y\}) \;=\; f(x) \bigsqcup f(y) \;\sqsupseteq\; f(y).$$

Conversely, every monotone function is also continuous monotone when every directed set in the cpo is finite.

Continuous monotonicity is sufficient for functions to have a fixed point on a pointed cpo:

Theorem 1
Let $f : D \to D$ be a continuous monotone map on the pointed cpo $<D,\sqsupseteq>$. Then f has a least fixed point equal to

$$\mathbf{fix}\, f \;=\; \bigsqcup\{\, f^k(\bot) \mid k \in \omega \,\}$$

where f^k is f iterated k times, i.e., $f^k = f \bullet f^{k-1}$, and f^0 is the identity.

Although it did not use this terminology, Kleene's first recursion theorem [52] presented Theorem 1 for the situation where f is a partial recursive functional over the pointed cpo D of partial functions ordered by approximation, where $\bot$ is the completely undefined function. See [61]. Hence Theorem 1 is usually attributed to Kleene.

Theorem 1 is false if we drop the continuity requirement. For example, the discontinuous function

$$f(x) \;=\; \begin{cases} x/2 + 1 & 0 \le x < 2 \\ x + 1 & 2 \le x \end{cases}$$

is monotone on the pointed cpo $[0,\infty]$ ordered by $\geq$ with $\bot = 0$. However

$$\mathbf{fix}\, f \;=\; \bigsqcup\{\, f^k(\bot) \mid k \in \omega \,\} \;=\; 2,$$

and $f(2) \neq 2$, i.e., 2 is not a fixed point of f.

On a complete lattice, we can adapt the result of Knaster-Tarski [90] to obtain a result like Theorem 1 that omits the continuity requirement:

Theorem 2
Let $f : D \to D$ be a monotone function on the complete lattice $<D,\sqsupseteq>$. Then f has a fixed point in D.

Moreover, the set of fixed points of f forms a complete lattice, so f has a least fixed point again. Unfortunately this theorem does not provide an effective means for finding fixed points.

We have avoided introducing ordinals in this presentation, since transfinite computations are not of use here. However, Hitchcock and Park [42] have shown that Theorem 1 can be extended for monotone functions by using ordinals: if f is monotone on the pointed cpo D, then $f^{\alpha}(\perp)$ reaches a least fixed point for some ordinal α, of cardinality less than or equal to that of D. Nelson's presentation of this result in [72] is well-written. Fitting [35] presents a proof that does not use ordinals directly, but Zorn's lemma.

3. Partial Order Programming

This section defines partial order programming, and studies two important subclasses of programs called *reductive partial order programs* and *continuous monotone partial order programs.* Because the definition is abstract, both model-theoretic and procedural semantics for these classes of programs are straightforward to develop. The procedural semantics reflect the fact that there are basically two ways to go about solving systems of inequalities: elimination and relaxation. Elimination corresponds to standard reductive or proof-theoretic semantics, while relaxation semantics are more novel.

The purpose of this section is to state *formally* what is meant by partial order programming. Some readers will find themselves wishing for clarification along the way. An example is provided at the end of the section to help make the abstraction more understandable. Also, a number of well-known classes of problems are expressed with partial order programming in the following section. Later sections also relate partial order programming to relaxation solution of inequalities and to computer programming.

3.1. Formal Definition

Definition

A *partial order program* P is a 4-tuple $<B,C,D,\sqsupseteq>$ where:

$<D,\sqsupseteq>$ is a pointed complete partial order of *values,*

B is a set of *objects,*

$C \subseteq B \times B$ is a set of *constraints.*

We require that $D \subseteq B$, so every value is also an object. Each member $<u,v>$ of C represents the constraint $u \sqsupseteq v$, and is normally written in this inequality form. In the special case where both u and v are values in D, we require that $u = v$, i.e., the only constraints in C involving two values are of the form $v \sqsupseteq v$.

Definition

Let v be a value, and u be an object that is not a value. A constraint $u \sqsupseteq v$ is $\sqsupseteq$-*oriented.* A constraint $v \sqsupseteq u$ is $\sqsubseteq$-*oriented.*

A partial order program is $\sqsupseteq$-oriented it has no $\sqsubseteq$-oriented constraint. Analogously, a program is $\sqsubseteq$-oriented it has no $\sqsupseteq$-oriented constraint. The program is *oriented* if it is either $\sqsupseteq$-oriented or $\sqsubseteq$-oriented.

Definition

A *valuation* is a function $\rho : B \rightarrow D$ that is the identity on D, and maps each object in B–D to a value in D.

Definition

The *trivial valuation* $\perp$ is the valuation $\perp : B \rightarrow D$ defined by

$$\perp(u) = \perp$$

for every u in B–D.

Definition
For two valuations ρ_1, ρ_2, we write $\rho_2 \sqsupseteq \rho_1$ if for every u in B, $\rho_2(u) \sqsupseteq \rho_1(u)$. Thus valuations are partially ordered pointwise. Under this ordering, $\rho \sqsupseteq \perp$ for every valuation ρ.

Furthermore we say a set S of valuations is *directed* if for every ρ_1, ρ_2 in S, there is an element ρ in S such that $\rho \sqsupseteq \rho_1$ and $\rho \sqsupseteq \rho_2$.

The space of valuations is then a pointed cpo, since the trivial valuation is a least element, and for every directed set S of valuations the least upper bound $\bigsqcup S$ is defined.

In particular since S is directed, $\{ \rho(u) \mid \rho \in S \}$ is directed for every u in B, so its least upper bound is defined, and

$$(\bigsqcup S)(u) = (\bigsqcup_{\rho \in S} \rho)(u) = \bigsqcup_{\rho \in S} \rho(u)$$

is well-defined for every u in B.

Definition
A constraint $u \sqsupseteq v$ is *satisfied* by the valuation ρ if $\rho(u) \sqsupseteq \rho(v)$.

Definition
A *model* of a partial order program $P = \langle B,C,D,\sqsupseteq \rangle$ is a valuation $\rho : B \rightarrow D$ such that, for each object u in B, every constraint $u \sqsupseteq v$ of C is satisfied by ρ.

A *minimal model* ρ of P is a model for which there is no other model ρ' such that $\rho \sqsupseteq \rho'$. Minimal models are not necessarily unique.

A *least model* ρ of P is a minimal model that is smallest, i.e., for which every other model ρ' satisfies $\rho' \sqsupseteq \rho$. Least models are therefore unique.

A *semantics* of a program is a model of the program.

Definition
A *partial order programming problem* is a statement of the form

minimize	G
subject to	P

where P is a partial order program, and G is an object of P, called the *goal* of the problem. A *solution* of this problem is a value $\rho(G)$ of G that is minimal over all models ρ of P.

Briefly, a model of a partial order program $P = \langle B,C,D,\sqsupseteq \rangle$ is an order-preserving map

from the preorder $<B,\sqsupseteq>$ defined by C to the partial order $<D,\sqsupseteq>$. We can think of finding a model for a partial order program as a process of mapping a preorder of "unknowns" into a partial order that is "understood", in a way that respects the preorder's constraints.

For example, consider the partial order programming problem

minimize	x		
subject to	x	$\geq$	y
	y	$\geq$	z
	z	$\geq$	3

and the three valuations below:

	Valuation		
Object	ρ_0	ρ_1	ρ_2
x	5	5	3
y	2	4	3
z	3	3	3

Here ρ_0 is not a model, ρ_1 is a model but not a minimal model, and ρ_2 is a minimal model. In fact, ρ_2 is the *least* model. The unique solution of the problem is the value $\rho_2(x) = 3$.

3.2. Solvability of Partial Order Programming Problems

There are several important issues concerning the definition of partial order programs above, and how it affects our ability to find a model for a partial order program.

First, we currently have no reasonable procedure for solving partial order programming problems. This is very like the situation for mathematical programming, and in fact is just as bad: for sufficiently interesting problems finding a solution is undecidable. This undecidability follows from the expressibility of either functional or logic programming in partial order programming demonstrated below. However functional and logic programming do have evaluation procedures, and we will need to produce some solution procedure if partial order programming is to be of use.

Second, partial order programs may have no model. For example, if $\sqsupseteq$ is a partial order on D for which the values a,b have no common upper bound, then the partial order program

$$u \sqsupseteq a$$
$$u \sqsupseteq b$$

has no model.

Third, the definition of partial order programs permits non-oriented programs like

$$\bot \sqsupseteq u$$
$$u \sqsupseteq \top$$

which has a model only if the domain D is trivial. Note also that when the domain of values is a total order, the constraint $v \sqsupseteq u$ is the same as the negation of $u \sqsupset v$. It will be interesting to extend the definition of partial order programs to include *strict inequalities*. However, strict inequalities allow us to write inconsistent programs like

$$u \sqsupset u$$

and lead to many thorny issues of negation.

Summarizing, the definition of partial order programming above does not give us a procedure for solving problems, or any kind of guarantee that a solution even exists. We must therefore define subclasses of partial order programs to deal with these issues. In the remainder of this monograph we will limit ourselves to one specific class that seems to cover many interesting problems: reductive partial order programs.

3.3. Reductive Partial Order Programs

Definition

A *reductive partial order program* P is a partial order program $<B,C,D,\sqsupseteq>$ whose constraint set C meets the following restriction:

$C : B \rightarrow B$ is a *constraint function* that is the identity on D.

For every object u in B, P is said to contain the inequality constraint

$$u \sqsupseteq C(u).$$

A model of a reductive partial order program is thus a valuation $\rho : B \rightarrow D$ such that, for each object u in B, $\rho(u) \sqsupseteq \rho(C(u))$.

Clearly every reductive partial order program is $\sqsupseteq$-oriented. This restriction eliminates the possibility of programs such as

$$\bot \sqsupseteq u$$
$$u \sqsupseteq \top$$

which simplifies our analysis considerably, and has the beneficial effect that every reductive partial order program has a model.

Above we noted that if $\sqsupseteq$ is a partial order on D for which the values a,b have no common upper bound, then the partial order program

$$u \sqsupseteq a$$
$$u \sqsupseteq b$$

has no model. This problem cannot arise with reductive partial order programs, since $C(u)$ must be single-valued.

When the value cpo D is a complete lattice, we can say still more about reductive partial order programs. In this case upper bounds always exist, and the partial order program with a,b above can be converted to the reductive partial order program

$$u \sqsupseteq (a \sqcup b).$$

This second program is 'equivalent' in that any model of the first will also be a model of the second. In fact, we can convert any $\sqsupseteq$-oriented partial order program over a complete lattice to an equivalent reductive partial order program:

Theorem 3
Let $P = <B,C,D,\sqsupseteq>$ be a $\sqsupseteq$-oriented partial order program, where $<D,\sqsupseteq>$ is a complete lattice. Then there is a reductive partial order program $P' = <2^B,C',D,\sqsupseteq>$ that is equivalent to P, where by equivalence we mean there is a natural isomorphism between models of P and models of P'.

Proof
Define $C' : 2^B \to 2^B$ for $S \subseteq B$ by

$$C'(S) = \{ v \mid u \in S,\ (u \sqsupseteq v) \in C \}.$$

In particular when u is in B, $C'(\{u\}) = \{ v \mid (u \sqsupseteq v) \in C \}$, so for a fixed u the set of constraints $\{ u \sqsupseteq v \}$ in C is converted to the single constraint $\{u\} \sqsupseteq C'(\{u\})$. Because of the restrictions on C this function C' is the 'identity' on D, in the sense that $C'(\{d\}) = \{d\}$ for all d in D.

Now, we can extend any valuation $\rho : B \to D$ to a valuation $\rho' : 2^B \to D$ in the obvious way:

$$\rho'(S) = \bigsqcup_{u \in S} \rho(u).$$

We claim that ρ is a model of P if and only if ρ' is a model of P':

First, if ρ is a model of P, then $\rho(u) \sqsupseteq \rho(v)$ for every $(u \sqsupseteq v) \in C$, so

$$\rho(u) \sqsupseteq \bigsqcup_{(u \sqsupseteq v) \in C} \rho(v).$$

But $\rho'(\{u\}) = \rho(u)$, and $\rho'(C'(\{u\})) = \bigsqcup_{(u \sqsupseteq v) \in C} \rho(v)$. Thus $\rho'(\{u\}) \sqsupseteq \rho'(C'(\{u\}))$, and ρ' is a model of P'.

Conversely, if ρ' is a model of P', then for every u in B, $\rho'(\{u\}) \sqsupseteq \rho'(C'(\{u\}))$. From this we can infer that

$$\rho(u) \sqsupseteq \rho(v) \text{ for every } (u \sqsupseteq v) \in C,$$

so ρ is a model of P. □

The significance of Theorem 3 is that restricting attention to reductive partial order programs does not cause much loss of generality. Because of Theorem 3, from this point on we will investigate only reductive partial order programs.

3.4. Model-Theoretic Semantics

Model-theoretic semantics for reductive partial order programs are simple. At this point they are even simpler than the elegant semantics for logic programs [8, 58], because partial order programs are defined more abstractly. Naturally we can say more about the inequalities and their solution for specific domains B and D (and we will do so shortly), but the point is that we can avoid details of the semantics neatly through abstraction.

Definition
For a reductive partial order program $P = \langle B,C,D,\sqsupseteq \rangle$, we define the function $T_P : (B \to D) \to (B \to D)$ by $T_P(\rho) = \rho \bullet C$.

Theorem 4
T_P is a continuous monotone map on valuations.

Proof
T_P maps valuations to valuations since $C : B \to B$. T_P is also monotone, since if $\rho_2 \sqsupseteq \rho_1$, then $\rho_2 \bullet C \sqsupseteq \rho_1 \bullet C$. Thus T_P maps directed sets to directed sets. Similarly T_P is continuous. To see this, recall that for (pointwise) directed sets of valuations S,

$$(\bigsqcup S)(u) = \bigsqcup_{\rho \in S} \rho(u),$$

so

$$\begin{aligned} &\left(\bigsqcup_{\rho \in S} T_P(\rho) \right)(u) \\ &= \left(\bigsqcup_{\rho \in S} \rho \bullet C \right)(u) \\ &= \left(\bigsqcup_{\rho \in S} (\rho \bullet C)(u) \right) \\ &= \left(\bigsqcup_{\rho \in S} (\rho(C(u))) \right) \\ &= \left(\bigsqcup_{\rho \in S} \rho \right)(C(u)) \\ &= T_P\left(\bigsqcup_{\rho \in S} \rho \right)(u). \end{aligned}$$

□

Theorem 5
$\rho \sqsupseteq T_P(\rho)$ if and only if ρ is a model of P.

Proof
ρ is a model iff for every u in B, $\rho(u) \sqsupseteq \rho(C(u)) = \rho \bullet C(u) = T_P(\rho)(u)$. □

Definition
An object u of a reductive partial order program is *definite* if $\{ C^n(u) \mid n \geq 1 \}$ includes some element in D. Otherwise it is *indefinite*.

Theorem 6 (Model-Theoretic Semantics of Reductive Partial Order Programs)
Let P be a reductive partial order program. The least fixed point of T_P is the valuation

$\rho = \bigsqcup_{k \in \omega} T_P^k(\perp)$, and is the least model of P.

Proof

Follows directly from Theorems 1 and 5, since T_P is continuous monotone. The least fixed point is a minimal model of the program, since all inequality constraints $u \sqsupseteq C(u)$ of P are satisfied by ρ with equality. Theorem 5 shows that any other model ρ' must be a *prefixed point* of T_P, i.e., $\rho' \sqsupseteq T_P(\rho')$. But then $\rho' \sqsupseteq \rho$ since ρ is the least prefixed point of T_P, so ρ is the least model.

More precisely, since $\rho' \sqsupseteq \perp$ and T_P is monotone, $T_P(\rho') \sqsupseteq T_P(\perp)$, so $\rho' \sqsupseteq T_P(\perp)$. Inductively then $\rho' \sqsupseteq T_P^k(\perp)$ for all $k \geq 0$, and $\rho' \sqsupseteq \rho$. □

Theorem 6 shows that reductive partial order programs always have models, and also a least model. It also shows that the least model of a program assigns $\perp$ to indefinite objects.

Since valuations are unrestricted here, Theorem 6 also gives a procedure for solving a partial order programming problem. The problem with goal G and program P has the solution $\rho(G)$, where ρ is the least model of P. No other model can give a smaller value to G.

3.5. Procedural Semantics

There are a number of ways to solve a system of inequalities, but two very basic methods are *elimination* and *relaxation.*

By elimination, we mean repeated simplification of the set of constraints through elimination of variables. The Gaussian and Gauss-Jordan elimination methods are classical methods for solving linear constraint systems. The subfield of algebraic geometry known as *elimination theory* studies such techniques for systems of polynomial equalities and inequalites [46].

Relaxation methods begin with an approximate solution and iteratively modify the solution to satisfy currently unsatisfied constraints. While relaxation methods are less well-understood, for many constraint systems (real-valued and otherwise) iterative relaxation is the *only* option for finding a solution.

We can therefore find the value $\rho(G)$ of an object G under the least model ρ of a reductive partial order program P in different ways. These ways give different procedural semantics for the program.

First, we can treat the program P 'proof-theoretically'. Given two constraints $u \sqsupseteq v$, $v \sqsupseteq w$ of P, we can *derive* $u \sqsupseteq w$. This is a form of elimination, where the use of v has been eliminated. The transitive closure of all constraints of P that this derivation process produces will eventually contain a minimal value g lower bounding G, if any such value exists. That is, if G is definite we will eventually derive a constraint $G \sqsupseteq g$ where g is minimal.

We can view this process as *evaluating* or *reducing* G to the value g. Repeated reduction obtains a sequence of objects $G^{(0)}$, $G^{(1)}$, ..., $G^{(M)}$ such that

$$G = G^{(0)} \sqsupseteq G^{(1)} \sqsupseteq \cdots \sqsupseteq G^{(M)} = g.$$

For this approach to be well-defined, all reduction sequences should ultimately give the same value g. This is guaranteed with partial order programs.

We can also obtain values for G via *relaxation,* and use relaxation as our procedural semantics. Relaxation is a simple paradigm for iteratively finding values that satisfy a set of constraints. The name "relaxation" was used by Southwell [86], after the application he pioneered in stress analysis.

In this monograph we identify relaxation with fixed point iteration. That is, by relaxation we mean an iterative technique which, given an initial value (namely: $\perp$) for the objects u in B–D, repeatedly selects an unsatisfied constraint $u \sqsupseteq v$ and updates u with the current value of v to enforce the constraint. ('Out of kilter' constraints are brought 'into kilter'.) By repeatedly enforcing constraints on G, we obtain an ascending chain $g^{(0)}$, $g^{(1)}$, ..., $g^{(N)}$ of values in D:

$$\perp = g^{(0)} \sqsubseteq g^{(1)} \sqsubseteq \cdots \sqsubseteq g^{(N)} = g.$$

Again, on reductive partial order programs enforcement of constraints in any 'fair' order (any order that eventually enforces all unsatisfied constraints) will ultimately obtain minimal model values for the objects if such a value exists. That relaxation works is then a consequence of Theorem 1.

Single-assignment semantics in programming languages have recently grown in importance. Relaxation semantics can be called *assignment refinement*: an assignment $G := g^{(k)}$ made for a partial order program can be replaced by $G := g^{(k+1)}$ provided that $g^{(k+1)} \sqsupseteq g^{(k)}$. In other words, we can replace the value of an object with better and better "approximations" as they are obtained.

3.6. Continuous Monotone Partial Order Programs

In this section we will consider an important subclass of reductive partial order programs for which these procedural semantics become more explicit: *continuous monotone partial order programs.* These programs are sufficiently general to cover many practical problems and the functional and logic programming paradigms.

Up to this point, we have assumed very little about the value domain D: it is a partial order with ordering relation $\sqsupseteq$. Frequently we know more about D. In particular, a set of *operators* on D may be useful, and we would like B to include expressions using these operators. Many computational paradigms define atoms ("variables") that can appear as arguments of operators ("functions").

For example, if we have $D = \omega$, the set of natural numbers, we might include a binary operator '*plus*' representing addition on D, and permit B to include expressions using it. (The interpretation of *Op* is fixed *a priori; plus* represents addition, and only addition, on D.) This can be accomplished by letting B include the set of *terms* involving

plus and some other domain of objects.

More exactly, we want to extend the definition of partial order programs by identifying a set of *atoms* A, letting B be a set of *expressions* over A and the values in D, and letting C represent both *constraints* on A and *reduction* or evaluation among expressions in B. Thus C blends both declarative and procedural semantics. The partial order program definition given earlier is general enough to encompass this extension.

Because C has additional structure, however, models of programs are more constrained. For example, suppose $u,v \in A$ and $plus(u,v)$ is an expression in B. If it is to be "reasonable", a valuation ρ should behave like a *substitution* acting on the variables in A, and must satisfy something like $\rho(plus(u,v)) = \rho(u)+\rho(v)$. We formalize this basic understanding as follows.

Definition
An *operator domain Op* is a set of *function symbols f*, each of which has an *arity* $\alpha(f)$. Given a fixed domain D, each function symbol f in Op denotes *a priori* a specific function $F: D^{\alpha(f)} \rightarrow D$, that, for our purposes, we require to be total and effectively computable in finite time. The set of these functions F paired with D is called a *D-algebra.*

Definition
If S is a set and Op is an operator domain, the set $Term(Op,S)$ of *first-order terms* is inductively defined as the terms obtained from the union of S with the expressions constructed from function symbols in Op acting on elements in $Term(Op,S)$. For example, if f has arity $\alpha(f) = 2$,

$$Term(\{f\},\{a,b\}) = \{ a, b, f(a,a), f(a,b), f(b,a), f(b,b), f(a,f(a,a)), \ldots \}.$$

Thus the set of expressions over D using function symbols in Op is $Term(Op,D)$.

Definition
An *algebraic valuation* over Op is a valuation $\rho : B \rightarrow D$ defined recursively for u in B:

(1) If u is in D, then $\rho(u) = u$.

(2) If u is in A, then $\rho(u)$ is a value in D.

(3) If otherwise $u = f(u_1, \ldots, u_n)$, and $F : D^n \rightarrow D$ is the function corresponding to f in Op, then

$$\rho(f(u_1, \ldots, u_n)) = F(\rho(u_1), \ldots, \rho(u_n)).$$

Thus ρ is expression evaluation on $Term(Op,D)$.

Definition
The *trivial algebraic valuation* $\perp_{Op,D}$ is the valuation $\perp_{Op,D} : B \rightarrow D$ defined by expression evaluation on $Term(Op,D)$, and

$$\perp_{Op,D}(u) = \perp$$

for every u in $B - Term(Op,D)$.

Thus $\perp_{Op,D}$ is the identity on D, and is the least algebraic valuation: $\rho \sqsupseteq \perp_{Op,D}$ for any algebraic valuation ρ, since all algebraic valuations must agree on $Term(Op,D)$.

Definition
Op is *continuous monotone,* if every function symbol f in *Op* denotes a function F on the pointed cpo $<D,\sqsupseteq>$ that is continuous monotone in each of its arguments. In this case the set of functions F and D is sometimes called a *continuous algebra.*

Theorem 7
Suppose *Op* is continuous monotone, and ρ_1, ρ_2 are algebraic valuations over *Op* on B. If $\rho_2 \sqsupseteq \rho_1$ on A, then $\rho_2 \sqsupseteq \rho_1$ on B.

Proof
Any composition of continuous monotone functions is continuous monotone. Given every function F denoted by symbols f in *Op* is continuous monotone on $<D,\sqsupseteq>$, then every term in $B = Term(Op,A\cup D)$ will denote a continuous monotone function on A. Thus $\rho_2 \sqsupseteq \rho_1$ on A implies $\rho_2 \sqsupseteq \rho_1$ on B. □

Definition
Given two terms e_1,e_2 in B, we write $e_1 \equiv_{Op,D} e_2$ (e_1 and e_2 are algebraically equivalent over *Op*) if for every algebraic valuation ρ, $\rho(e_1) = \rho(e_2)$.

Definition
An *algebraic partial order program* P is a reductive partial order program $<Term(Op,A\cup D),C,D,\sqsupseteq>$ where

$<D,\sqsupseteq>$ is a pointed cpo of values,
Op is an operator domain of functions on D,
A is a set of atoms,
$C : B \rightarrow B$ is a constraint function, where $B = Term(Op,A\cup D)$.

For every object u in B, P is said to contain the inequality constraint $u \sqsupseteq C(u)$.
C must meet two requirements:

(1) C is the identity on D, and more generally $C : Term(Op,D) \rightarrow D$.

(2) C is a *computation rule* [62] for reduction of expressions. Specifically, if f is a n-ary function symbol in *Op* so $f(u_1, \ldots, u_n)$ is an object in B, then for some $m \geq 1$ there exist argument indices $i_1, \ldots, i_m$ such that

$$C(f(\cdots, u_{i_1-1}, u_{i_1}, u_{i_1+1}, \cdots, u_{i_m-1}, u_{i_m}, u_{i_m+1}, \cdots))$$

$$\equiv_{Op,D} \quad f(\cdots, u_{i_1-1}, C(u_{i_1}), u_{i_1+1}, \cdots, u_{i_m-1}, C(u_{i_m}), u_{i_m+1}, \cdots).$$

These restrictions on C have several ramifications:

- The second restriction requires C to reduce $f(u_1, \ldots, u_n)$ either to $f(..., C(u_{i_1}), ..., C(u_{i_m}), ...)$ or to some equivalent expression. Thus C can perform algebraic simplifications that preserve equivalence over D. For example, if D is the natural numbers, Op is $\{plus\}$ representing addition on D, $A = \{p\}$, and $B = Term(Op,A \cup D)$, C could include the following definitions:

$$\begin{aligned} C(\, plus(p,0)\,) &= p \\ C(\, plus(0,p)\,) &= p \\ C(\, plus(0,0)\,) &= 0. \end{aligned}$$

- The first and second restrictions together require C to be expression evaluation on $Term(Op,D)$, so if f is a function symbol in Op denoting the function $F : D^n \rightarrow D$ and $v_1, \ldots, v_n$ are in $Term(Op,D)$, then

$$C(f(v_1, \ldots, v_n)) = F(C(v_1), \ldots, C(v_n)).$$

- The restrictions also essentially require C to be expression evaluation on $B - A$. We will clarify this momentarily.
- For objects u in A, $C(u)$ is not restricted. Thus the real "program" of an algebraic partial order program comes from the constraints on A.

Definition
A *continuous monotone partial order program* is an algebraic partial order program $<Term(Op,A \cup D),C,D,\sqsupseteq>$ where Op is continuous monotone.

The problem of finding a model for algebraic partial order programs is, in general, undecidable. This is evident from the undecidability of Hilbert's tenth problem [28]. Also, algebraic partial order programs may have many models but no least model, such as with the program $x \geq -y$ over the value domain D of the integers. The following theorem shows that continuous monotone programs resolve these problems, and have least models that are algebraic, modulo indefiniteness.

Definition
A valuation ρ_1 *approximates* a valuation ρ_2 if for every object $u \in B$, either $\rho_1(u) = \rho_2(u)$, or $\rho_1(u) = \perp$.

Theorem 8
Let P be a continuous monotone partial order program. Then P has a least model, which approximates an algebraic valuation.

Proof
Let $P = <Term(Op,A \cup D),C,D,\sqsupseteq>$, and consider the map $T_P(\rho) = \rho \bullet C$. Since every continuous monotone partial order program is also a reductive partial order

program, Theorem 4 shows that T_P is a continuous and monotone map from valuations to valuations. Theorem 6 then implies

$$\rho^* \;=\; \bigsqcup_{k\in\omega} T_P{}^k(\bot_{Op,D})$$

is the least model of P, and is a least fixed point solution to the inequality $\rho \sqsupseteq T_P(\rho)$. Thus $\rho^* = \rho^* \bullet C$.

We must now show that this least model ρ^* approximates an algebraic valuation. Let u be in $B = Term(Op,A\cup D)$. If u is indefinite, $\rho^*(u)$ is $\bot$. If u is definite, then for some integer $k > 0$, which we can call the *stage* of u, $C^k(u)$ is in D. We prove by induction on k that ρ^* is algebraic for all definite objects u in B of stage at most k. Since ρ^* is $\bot$ for indefinite objects, this induction establishes the theorem.

If $k = 0$, then u is in D and ρ^* trivially meets the conditions on algebraic valuations. Assume then $k \geq 1$, and that ρ^* is algebraic for all objects u of stage at most k–1. If $u = f(\, u_1\, ,\, \ldots\, ,\, u_n\,)$ has stage k, $C(\, f(\, u_1\, ,\, \ldots\, ,\, u_n\,)\,)$ has stage k–1, and the restrictions on C imply there is some $m \geq 1$ and arguments $i_1, \ldots, i_m$ such that

$$\begin{aligned}
&\rho^*(\, f(\, u_1\, ,\ldots,\, u_n\,)\,)\\
&\quad= \rho^* \bullet C(\, f(\, u_1\, ,\ldots,\, u_n\,)\,)\\
&\quad= \rho^*(\, C(\, f(\, u_1\, ,\ldots,\, u_n\,)\,)\,)\\
&\quad= \rho^*(\, f(\ldots,\, u_{i_1-1},\, C(u_{i_1}),\, u_{i_1+1},\, \ldots,\, u_{i_m-1},\, C(u_{i_m}),\, u_{i_m+1},\, \ldots)\,)\\
&\quad= F(\ldots,\, \rho^*(u_{i_1-1}),\, \rho^* \bullet C(u_{i_1}),\, \rho^*(u_{i_1+1}),\, \ldots,\, \rho^*(u_{i_m-1}),\, \rho^* \bullet C(u_{i_m}),\, \rho^*(u_{i_m+1}),\, \ldots)\\
&\quad= F(\ldots,\, \rho^*(u_{i_1-1}),\, \rho^*(u_{i_1}),\, \rho^*(u_{i_1+1}),\, \ldots,\, \rho^*(u_{i_m-1}),\, \rho^*(u_{i_m}),\, \rho^*(u_{i_m+1}),\, \ldots).
\end{aligned}$$

Thus the induction holds for objects of stage k, and ρ^* approximates an algebraic valuation. □

It is important to understand how ρ^* can fail to be algebraic. Consider the following definition for C:

$$\begin{aligned}
C(\, plus(p,q)\,) &= plus(plus(p,0),q)\\
C(\, plus(plus(p,0),q)\,) &= plus(p,q)\\
C(\, p\,) &= 2\\
C(\, q\,) &= 3.
\end{aligned}$$

With this definition $\rho^*(plus(p,q)) = \bot$, but $\rho^*(p) + \rho^*(q) = 5$. Thus ρ^* only approximates an algebraic valuation. It is algebraic modulo the indefiniteness of the object $plus(p,q)$ under C. Nevertheless, it *is* the least model, since it satisfies the constraints of C.

In some situations we will need a computation rule C that produces an algebraic valuation. In the context of recursive programming languages, Manna [62] shows that what is needed is a *fixpoint computation rule*. For our purposes, however, the requirements on C above are sufficient.

Combining Theorems 7 and 8, we see that models of continuous monotone partial

order programs are determined on B by their values on A. Thus values of the least model on A gives a semantics for the program.

We can find values of the least model on A in two ways:

(1) A valuation on A be determined iteratively (by relaxation) yielding a sequence of approximations to a model that increases in a continuous monotone fashion.

(2) When A is finite, continuous monotone partial order programs often can be *linearized* (converted to systems of linear inequalities over some semiring). Values for objects in A can then be obtained with elimination methods.

We will clarify the second statement later. The first statement says that if $\mathbf{u}$ denotes the vector of atoms in the finite set A, and $\mathbf{f}$ is a vector-valued continuous monotone function on A, then the continuous monotone partial order programming problem

minimize	$\mathbf{u}$
subject to	$\mathbf{u} \sqsupseteq \mathbf{f}(\mathbf{u})$

over the pointed cpo $<D,\sqsupseteq>$, is solved by the relaxation iteration

$$\mathbf{u} := \mathbf{f}(\mathbf{u})$$

with initial value $\perp$ for $\mathbf{u}$. This follows from Theorem 1, since the cross-product of n copies of the pointed cpo is again a pointed cpo, and since the vector function $\mathbf{f}$ is continuous monotone on this cpo. The fixed point obtained is the least fixed point, and is the least solution to the inequalities in the program.

This shows that a relaxation procedure which satisfies all constraints *in parallel* converges to a solution of the constraints. The same statement can be made for *serial* constraint satisfaction. Since the vector functions

$$\mathbf{F}_i(u_1, \ldots, u_n) = <u_1, \ldots, u_{i-1}, f_i(u_1,\ldots,u_n), u_{i+1}, \ldots, u_n>, \quad 1 \le i \le n,$$

are each continuous monotone, the vector function

$$\mathbf{F} = \mathbf{F}_n \bullet \cdots \bullet \mathbf{F}_1$$

is also continuous monotone. Thus relaxation with $\mathbf{F}$ will converge to the least fixed point of $\mathbf{F}$. But every fixed point of $\mathbf{F}$ will also be a fixed point of $\mathbf{f}$, and vice-versa. So, relaxation with either $\mathbf{F}$ or $\mathbf{f}$ must yield the same (least) solution of the inequalities $\mathbf{u} \sqsupseteq \mathbf{f}(\mathbf{u})$. Serial and parallel relaxation will yield the same result.

3.7. A Sample Continuous Monotone Partial Order Program

Consider the partial order programming problem

minimize	min(a, max(c, e))		
subject to	a	≥	0.50 * min(b, f)
	a	≥	0.50 * min(c, d)
	b	≥	0.20
	c	≥	0.45
	c	≥	0.30
	d	≥	1.00
	e	≥	0.50
	f	≥	0.90 * e.

Formally we specify the program of the problem to be $<B,C,D,\geq>$, where D is the set of real numbers between 0 and 1 ordered by value, A is the set $\{a,b,c,d,e,f\}$, B is the set of expressions on A and D using the operators Op = {min, max, *}, and C is the collection of constraints above. This program is reductive, since all constraints are ≥-oriented.

Note that Op is continuous monotone on D, and max serves as $\bigsqcup$ on D. Using Theorem 3, therefore, we can convert the program to the following equivalent continuous monotone partial order program (where we have labeled constraints for reference later):

(C1):	a	≥	max(0.50 * min(b, f), 0.50 * min(c, d))
(C2):	b	≥	0.20
(C3):	c	≥	max(0.45, 0.30)
(C4):	d	≥	1.00
(C5):	e	≥	0.50
(C6):	f	≥	0.90 * e.

The algebraic valuation ρ defined by

ρ(a)	=	0.225
ρ(b)	=	0.20
ρ(c)	=	0.45
ρ(d)	=	1.00
ρ(e)	=	0.50
ρ(f)	=	0.45

is a model for this program. With this model, the goal

min(a, max(c, e))

is assigned the value 0.225. In fact, ρ is a least model: it always gives the smallest possible values under the order ≥.

Procedural semantics yield the same result. First, reduction of the goal min(a,max(c,e)) might first produce the object

min(a, max(max(0.45, 0.30), e))

using the inequality (C3), and then reduced to

$$\min(\ a,\ \max(\ 0.45,\ e\ )\ ).$$

This process will eventually give the value 0.225 again.

Second, relaxation produces a sequence of valuations culminating in the model ρ given above. For example, we might derive the sequence of valuations on A below. The goal is refined to take the value 0.225 after the seventh step.

	constraint enforced							
Object		(C5)	(C6)	(C2)	(C1)	(C3)	(C4)	(C1)	· · ·
e	⊥	0.50	0.50	0.50	0.50	0.50	0.50	0.50	· · ·
f	⊥	⊥	0.45	0.45	0.45	0.45	0.45	0.45	· · ·
b	⊥	⊥	⊥	0.20	0.20	0.20	0.20	0.20	· · ·
a	⊥	⊥	⊥	⊥	0.10	0.10	0.10	0.225	· · ·
c	⊥	⊥	⊥	⊥	⊥	0.45	0.45	0.45	· · ·
d	⊥	⊥	⊥	⊥	⊥	⊥	1.00	1.00	· · ·
min(a,max(c,e))	⊥	⊥	⊥	⊥	0.10	0.10	0.10	0.225	· · ·

There are many issues remaining to be studied in semantics of continuous monotone programs that this example does not show. One interesting area deals with programs such as

$$\begin{aligned} a &\geq 3 \times b - 8 \\ b &\geq 2 \times a + 1 \end{aligned}$$

where D is the real numbers with least element $\bot = -\infty$. This program is not easily solved by relaxation. (Indeed, it can be solved by elimination only if we can apply algebraic properties of the operators + and × over the value domain.) Relaxation does not produce informative minimal models *unless it is provided with a good starting valuation.* The least model of this program is the trivial valuation assigning $\bot$ to a and b. If we adjoin $\top = \infty$ to D, this program has *greatest* model [8] assigning $\top$ to a and b. However the 'most informative' model assigns 1 to a and 3 to b, and in this case is the *optimal fixed point* [63] of the program.

Obtaining good initial values for iterations has been a largely heuristic heuristic process until recently. For example, the *continuation method* [6,7] finds solutions to a system of constraints by beginning with a similar system with a known solution, then following curves that deform the system (and its solution) to the desired system. The approach has promise and may make iterative approaches possible where they have not been in the past.

Note that some programs are not finitely evaluable [34]. This is illustrated by the program

$$z_n \ \geq \ \min(\ 2^{-n},\ z_{n+1}\ )$$

where D is the nonnegative rational numbers ordered by ≥, with the usual arithmetic operators, and B is the set of arithmetic expressions involving nonnegative rationals

and atoms in $\{ z_n \mid n \geq 1 \}$. This program has as its least model ρ the valuation

$$\rho(z_n) = 0 \qquad (n \geq 1),$$

but any naive, finite evaluation of the program will fail to find this.

However, given a set of inequalities we can permit as our semantics *any* decision procedure for solving inequalities such as [9, 19, 49, 83, 84], or procedural semantics can incorporate this procedure. This idea has been exploited nicely in the 'Constraint Logic Programming' scheme [47, 48], notably its implementations in *CLP*(**R**) over the real numbers, which incorporate an incremental simplex constraint solver for logic programs augmented with linear equality and inequality constraints. Using general constraint solvers could permit us to handle both of the problematic programs above.

4. Examples of Partial Order Programming

To gain some understanding of how partial order programming can be used, we give a sequence of different situations in which it arises naturally. Many of the examples here have direct applications in modeling of systems. Also, many of the examples here can be solved iteratively, by relaxation. The relationship between partial order programming and relaxation will be investigated later.

Some of the examples given below are more naturally stated as maximization problems than as minimization problems. For every maximization problem, however, there is an equivalent minimization problem. Given a maximization problem using the partial order $\sqsupseteq$, we can simply "reverse" the partial order to give a new problem whose goal is minimized where it was maximized by the original problem. This is only a consequence of the fact that every partial order is a binary relation with two possible directions for extrema, and the direction called "minimal" is arbitrary.

Formally, if $\sqsupseteq'$ is the reversal of $\sqsupseteq$ defined by

$$u \sqsupseteq' v \quad \text{iff} \quad v \sqsupseteq u$$

and $<B,C,D,\sqsupseteq>$ represents a partial order program using the partial order $\sqsupseteq$, then the problem

maximize	G
subject to	$<B,C,D,\sqsupseteq>$

is equivalent to its reversal

minimize	G
subject to	$<B,C,D,\sqsupseteq'>$

Every constraint $v \sqsupseteq u$ of the original problem is reversed to $u \sqsupseteq' v$, so maximal values under $\sqsupseteq$ are minimal values under $\sqsupseteq'$. Below then, problems are stated in the most natural way.

4.1. Mathematical Programming

Many problems can be modeled as mathematical programs, systems with objective functions and collections of inequality constraints. The *nonlinear programming problem*

minimize	$f(\mathbf{x})$
subject to	$g_i(\mathbf{x}) \geq 0$
	$(1 \leq i \leq m)$,

requires the functions f and $\{g_i\}$ to be real-valued, and uses f to define an ordering among feasible solution vectors x satisfying $\{g_i\}$. When f and $\{g_i\}$ are linear functions, we obtain the *linear programming problem* in canonical form

minimize	$\mathbf{c}' \mathbf{x}$
subject to	$\mathbf{A}\mathbf{x} \geq \mathbf{b}$
	$\mathbf{x} \geq \mathbf{0}$

where $'$ is matrix transposition [74]. Mathematical programming has countless applications, including, as a simple example, most spreadsheet models.

The similarity between these problems and partial order programming problems is obvious. However, there is a significant difference: In mathematical programming the objective is real-valued, and defines a *total order* on the space of feasible solutions. In partial order programming, goal objects may take on any value in a *partial order*.

This difference is significant for some problems. It is evident from the popularity of linear and nonlinear programming that in many cases a real-valued objective function is just what is wanted. However use of an objective function is sometimes artificial. This is certainly the case when there are *multiple* objectives, or hierarchical objectives [51]. Multiple objectives frequently arise in problems involving value, utility, or preference, and in situations where multiple decision makers cooperate. For example, the problem

minimize	*f*(*Cost*, *CompletionTime*, *Shoddiness*)
subject to	*C*(*Cost*, *CompletionTime*, *Shoddiness*)

where f is some real-valued objective function and C is a set of constraints, seems inherently more artificial than the problem

minimize	<*Cost*, *CompletionTime*, *Shoddiness*>
subject to	*C*(*Cost*, *CompletionTime*, *Shoddiness*)

which requests *all* minimal (Pareto optimal) solutions under the 3-dimensional vector partial ordering of domination. For example, we have the domination

< \$1000, 20 days, failure rate 0.1 > ≤ < \$1500, 48 days, failure rate 0.2 >

but the two scenarios

< \$1000, 48 days, failure rate 0.2 >
< \$1500, 20 days, failure rate 0.1 >

are incomparable, i.e., neither dominates the other. Where the first problem maps the three attributes of interest into a number, the second leaves them in a partial order. Often it is not obvious why there should be a single number determining the desirability of feasible solutions. (Why should apples be better or worse than oranges?) The entire book [51] works at formalizing situations in which real-valued objectives are reasonable.

Partially ordered value domains introduce the possibility of using structured values in optimization problems. These values might enter as algebraic expressions specifying bounds on unspecified parameter values as done with *CLP*(**R**) [48, 55], or as

explanations or justifications (histories of derivation, qualifications, or endorsements of values clarifying their accuracy or sensitivity). To some extent structures can be encoded with numerical parameters, and this is certainly done in practice: for example sets are encoded as collections of 0–1 variables, and certainties are encoded as constants. However it is worthwhile to represent structures directly.

4.2. Linear Algebraic Systems

In the linear program

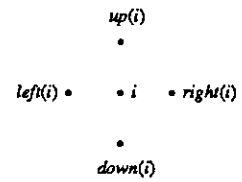
minimize	$\mathbf{c}'\,\mathbf{u}$
subject to	$\mathbf{Au} \geq \mathbf{b}$
	$\mathbf{Au} \leq \mathbf{b}$

where **A** is a square, non-singular matrix, the solution **u** of the constraints is uniquely determined. Elimination methods are commonly used to solve linear systems, the Gaussian and Gauss-Jordan elimination methods being the best well-known.

Relaxation methods are also used for some problems. Probably the most familiar example of a linear system solvable by relaxation is the finite difference version of Laplace's equation

$$\nabla^2 u = \frac{\partial^2}{\partial x^2} u + \frac{\partial^2}{\partial y^2} u = 0.$$

Let u_i be the solution of the discretized approximation to the equation. The discretization is given by a rectilinear grid **G**, with boundary conditions defining u only on the boundary points of **G**. Numbering the points in **G** as $\{u_1, \ldots, u_n\}$ arbitrarily, let *left*(i), *right*(i), *up*(i), and *down*(i) give the neighbors of node i in the interior of the grid:



A node i on the boundary of **G** takes on a boundary value $u_i = b_i$. Each node i in the interior of **G** satisfies the constraint

$$u_i = \frac{u_{left(i)} + u_{right(i)} + u_{up(i)} + u_{down(i)}}{4}.$$

The problem is to find values for $u_1, \ldots, u_n$. If we define the coefficient matrix $\mathbf{A} = (a_{ij})$ by

$$a_{ij} = \begin{cases} 1/4 & \text{if } i \text{ is an interior node and } j \in \{left(i), right(i), up(i), down(i)\} \\ 0 & \text{otherwise} \end{cases}$$

and boundary condition vector $\mathbf{b} = (b_i)$ by

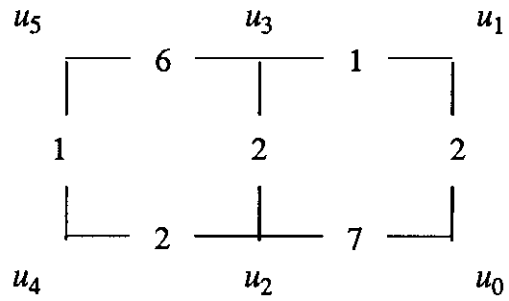
$$b_i = \begin{cases} u_i & \text{if } i \text{ is a boundary node} \\ 0 & \text{otherwise} \end{cases}$$

then the vector $\mathbf{u} = (u_i)$ satisfies the matrix equation $\mathbf{u} = \mathbf{Au} + \mathbf{b}$, and hence both $\mathbf{u} \geq \mathbf{Au} + \mathbf{b}$ and $\mathbf{u} \leq \mathbf{Au} + \mathbf{b}$.

This vector constraint may be solved iteratively [45]. Specifically, given any finite initial values for the interior nodes, the equation above may be used as an assignment, $\mathbf{u} := \mathbf{Au} + \mathbf{b}$. Since $\mathbf{A}$ is positive, the function $\mathbf{f(u)} = \mathbf{Au} + \mathbf{b}$ is monotone, and since $\mathbf{A}$ has spectral radius less than 1, the iteration always converges (though possibly in an infinite number of steps).

4.3. Shortest Paths, Path Problems, and Dynamic Programming

Let $G = <V,E>$ be a graph with vertices V and edges E, such that for each edge $<i,j> \in E$ there is a *cost* $a_{ij} \geq 0$. Each node i is assigned a path distance cost u_i, which is the *sum* of the costs of edges in a path between node i and the source node 0. The single-source shortest path problem is then to find the minimum value for each u_i [74]. For example, with the graph



the shortest path from node 4 to node 0 first visits nodes 2, 3, and 1, having total cost $u_4 = 2+2+1+2 = 7$. For simplicity, the domain D of values and edge costs here is restricted to the *nonnegative* real numbers.

The requirements on the path costs can be expressed as the following partial order program:

minimize	$<u_0, \ldots, u_n>$			
subject to	u_0	$\geq$	0	
	u_i	$\geq$	$\min_{0 \leq j \leq n}$	$a_{ij} + u_j \quad (i \neq 0).$

Assuming that the a_{ij} are nonnegative and that we are given suitable initial values for the u_i (such as $+\infty$), and we fix u_0 at 0, then repeated execution of the following ALGOL-like program may be used to find a solution:

$$u_i := \min_{0 \leq j \leq n} a_{ij} + u_j.$$

For example, with the graph above execution of the assignment statement above for the variables u_2, u_4, u_1, u_3, u_5, u_2, u_4, u_5 at respective times $t_1 < ... < t_8$ would produce the following sequence of value assignments:

	Time										
Object	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	...
u_0	0	0	0	0	0	0	0	0	0	0	...
u_1	$\perp$	$\perp$	$\perp$	**2**	2	2	2	2	2	2	...
u_2	$\perp$	**7**	7	7	7	7	**5**	5	5	5	...
u_3	$\perp$	$\perp$	$\perp$	$\perp$	**3**	3	3	3	3	3	...
u_4	$\perp$	$\perp$	**9**	9	9	9	9	**7**	7	7	...
u_5	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	**9**	9	9	**8**	8	...

Assignments are marked in boldface, and the symbol $\perp$ represents the initial value $+\infty$. The values after t_8 are unchanged by further iteration, and give the desired shortest path costs.

The ALGOL-like program is coincidentally a program in the recently developed UNITY (Unbounded Nondeterministic Iterative Transformations) paradigm [22]. A UNITY program is a collection of declarations and of assignment statements. Inspiration for the paradigm came from studying difficulties arising in spreadsheet computations. Assignments can be single, as with

$$u := v$$

where u is a program variable, and v is an evaluable expression, or multiple, as with

$$u_1, \ldots, u_n := v_1, \ldots, v_n.$$

These assignments in a program are executed atomically but nondeterministically, in any order, subject to the fairness constraint that every assignment is executed infinitely often. UNITY is thus a paradigm for expressing loosely-coupled parallel and distributed algorithms.

Results that are related to the method for solving the shortest path problem above appear in [89] for path problems. The *single-source path expression problem* is to find regular expressions u_i specifying all paths from node i to the source node of a graph. The problem may be expressed as follows:

minimize	$<u_0, \ldots, u_n>$		
subject to	u_0	$\supseteq$	$\{\varepsilon\}$
	u_i	$\supseteq$	$\bigcup_{j \text{ adjacent to } i} a_{ij} \bullet u_j \quad (i \neq 0).$

Here a_{ij} is a symbol for the edge between nodes i and j, $\cup$ is union, ε is the empty string, and $\bullet$ is concatenation. While iteration does not solve this problem, since it does not reach a solution, we can reformulate it as an all-pairs shortest-path problem that can be solved iteratively. If u_{ij} denotes the regular expression of all paths from node i to node j, the problem becomes

$$\begin{array}{lll} \text{minimize} & \langle u_{00}, \ldots, u_{n0} \rangle & \\ \text{subject to} & u_{00} \supseteq \{\varepsilon\} & \\ & u_{ij} \supseteq \bigcup_{\substack{k \text{ adjacent to } i \\ j \text{ adjacent to } k}} u_{ik} \bullet (u_{kk})^* \bullet u_{kj} & (i \neq 0). \end{array}$$

This formulation is clearly related to the Warshall-Floyd-Kleene transitive closure algorithm [57], which is equivalent to Gauss-Jordan elimination without pivoting given the appropriate definition for *. (On numeric domains one should take $a^* = 1/(1-a)$.) Tarjan shows that a variety of problems – finding shortest paths, solving sparse systems of linear equations, and performing global flow analysis of computer programs – have solutions that are homomorphic images of solutions to the path expression problem [89]. Thus he reduces each of these problems to the path expression problem.

Dynamic programming is also related to path problems. In dynamic programming, Bellman's "principle of optimality" dictates that the optimal solution of a problem be obtained by combining optimal solutions of its subproblems. This reductive approach defines path relationships between problems and subproblems. Gnesi, Montanari, and Martelli [39] show that the general dynamic programming problem

$$\begin{array}{lll} \text{minimize} & \langle x_1, \ldots, x_n \rangle & \\ \text{subject to} & x_i = \tau_i(x_1, \ldots, x_n) & (1 \leq i \leq n) \end{array}$$

is solvable with fixed point iteration when all τ_i are monotone (as they are in the shortest-paths problem). Monotonicity of the cost of combined subproblems is sufficient for Bellman's principle of optimality to apply.

For the sake of concreteness, consider the well-known problem of optimally forming the matrix product $M_1 \times \cdots \times M_n$ where M_i is a matrix of dimensions $r_{i-1} \times r_i$, $1 \leq i \leq n$ [1]. Let the cost of multipying $n \times m$ by $m \times p$ matrices be nmp multiplications, and for arbitrary $i < j$ denote the optimal cost of forming the subproblem product $M_i \times \cdots \times M_j$ by m_{ij}. The monotone dynamic programming formulation can be cast as the following partial order programming problem:

$$\begin{array}{lll} \text{minimize} & m_{1n} & \\ \text{subject to} & m_{ii} \geq 0 & \\ & m_{ij} \geq \min_{i \leq k < j} m_{ik} + m_{k+1\,j} + r_{i-1}\, r_k\, r_j & (i < j). \end{array}$$

4.4. Approximate Consistent Labeling and Constraint Networks

Consistent labeling problems, also called constraint satisfaction problems, begin with a set of labels and a collection of variables, and seek to assign labels to the variables in a way that satisfies a given set of constraints. The literature is dispersed [40, 41, 59, 68, 70, 77, 79, 94]. Formally, there is a collection of variables

$\{u_i \mid 1 \le i \le n\}$ that can be labeled with members of the finite set of labels Λ. For specified subsets of indices $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, $m \ge 1$, there are constraint predicates

$$P_{i_1 \ldots i_m}(u_{i_1}, \ldots, u_{i_m}).$$

Since the constraint predicate $P_{i_1 \ldots i_m}$ can be viewed as a subset of Λ^m (a relation), the constraint is sometimes written

$$< u_{i_1}, \ldots, u_{i_m} > \in P_{i_1 \ldots i_m}.$$

We seek a *consistent labeling*, i.e., an assignment of labels to variables that satisfies all of the constraints.

There are many real applications, but the 8 queens problem and four-coloring of planar maps are commonly cited. To illustrate here we pick Lewis Carroll's "Salt and Mustard" problem [13]. Barry, Cole, Dix, Lang, and Mill are friends who meet daily to dine, and take certain condiments with their meal — salt, mustard, both, or neither. They agree to abide by the following constraints whenever beef appears at their table:

(1) Barry takes salt iff either Cole or Lang takes only one of the condiments;
Barry takes mustard iff either Dix takes neither condiment, or Mill takes both.

(2) Cole takes salt iff either Barry takes only one condiment or Mill takes neither;
Cole takes mustard iff either Dix or Lang takes both.

(3) Dix takes salt iff either Barry takes neither condiment or Cole takes both;
Dix takes mustard iff either Lang or Mill takes neither.

(4) Lang takes salt iff either Barry or Dix takes only one condiment;
Lang takes mustard iff either Cole or Mill takes neither.

(5) Mill takes salt iff either Barry or Lang takes both condiments;
Mill takes mustard iff either Cole or Dix takes only one.

The problem is to determine which of the condiments each friend takes. If we represent the possibilities for each friend as a set Λ = {n,s,m,b} (for 'neither', 'salt only', 'mustard only', and 'both'), then constraints (1)-(5) are equivalent to the following consistent labeling problem for the objects B,C,D,L,M [32]:

$$
\begin{array}{lll}
<B, C, L> & \in & \Lambda \times \Lambda \times \Lambda - \{s,b\} \times \{n,b\} \times \{n,b\} \\
<B, C> & \in & \Lambda \times \Lambda - \{n,m\} \times \{s,m\} \\
<B, L> & \in & \Lambda \times \Lambda - \{n,m\} \times \{s,m\} \\
<B, D, M> & \in & \Lambda \times \Lambda \times \Lambda - \{m,b\} \times \{s,m,b\} \times \{n,s,m\} \\
<B, D> & \in & \Lambda \times \Lambda - \{n,s\} \times \{n\} \\
<B, M> & \in & \Lambda \times \Lambda - \{n,s\} \times \{b\} \\
<B, C, M> & \in & \Lambda \times \Lambda \times \Lambda - \{n,b\} \times \{s,b\} \times \{s,m,b\} \\
<C, M> & \in & \Lambda \times \Lambda - \{n,m\} \times \{n\} \\
<B, C> & \in & \Lambda \times \Lambda - \{s,m\} \times \{n,m\} \\
<C, D, L> & \in & \Lambda \times \Lambda \times \Lambda - \{m,b\} \times \{n,s,m\} \times \{n,s,m\} \\
<C, D> & \in & \Lambda \times \Lambda - \{n,s\} \times \{b\} \\
<C, L> & \in & \Lambda \times \Lambda - \{n,s\} \times \{b\} \\
<B, C, D> & \in & \Lambda \times \Lambda \times \Lambda - \{s,m,b\} \times \{n,s,m\} \times \{s,b\} \\
<B, D> & \in & \Lambda \times \Lambda - \{n\} \times \{n,m\} \\
<C, D> & \in & \Lambda \times \Lambda - \{b\} \times \{n,m\} \\
<D, L, M> & \in & \Lambda \times \Lambda \times \Lambda - \{m,b\} \times \{s,m,b\} \times \{s,m,b\} \\
<D, L> & \in & \Lambda \times \Lambda - \{n,s\} \times \{n\} \\
<D, M> & \in & \Lambda \times \Lambda - \{n,s\} \times \{n\} \\
<B, D, L> & \in & \Lambda \times \Lambda \times \Lambda - \{n,b\} \times \{n,b\} \times \{s,b\} \\
<B, L> & \in & \Lambda \times \Lambda - \{s,m\} \times \{n,m\}
\end{array}
$$

$$
\begin{array}{lll}
\langle D, L\rangle & \in & \Lambda \times \Lambda - \{s,m\} \times \{n,m\} \\
\langle C, L, M\rangle & \in & \Lambda \times \Lambda \times \Lambda - \{s,m,b\} \times \{m,b\} \times \{s,m,b\} \\
\langle C, L\rangle & \in & \Lambda \times \Lambda - \{n\} \times \{n,s\} \\
\langle L, M\rangle & \in & \Lambda \times \Lambda - \{n,s\} \times \{n\} \\
\langle B, L, M\rangle & \in & \Lambda \times \Lambda \times \Lambda - \{n,s,m\} \times \{n,s,m\} \times \{s,b\} \\
\langle B, M\rangle & \in & \Lambda \times \Lambda - \{b\} \times \{n,m\} \\
\langle L, M\rangle & \in & \Lambda \times \Lambda - \{b\} \times \{n,m\} \\
\langle C, D, M\rangle & \in & \Lambda \times \Lambda \times \Lambda - \{n,b\} \times \{n,b\} \times \{m,b\} \\
\langle C, M\rangle & \in & \Lambda \times \Lambda - \{s,m\} \times \{n,s\} \\
\langle D, M\rangle & \in & \Lambda \times \Lambda - \{s,m\} \times \{n,s\}
\end{array}
$$

There are two kinds of constraints here, forbidding labelings either for triples of friends or for pairs of friends. For example, the first constraint forbids Barry to take salt when Cole and Lang take neither or both. The second constraint forbids Barry to take either no condiment or mustard only when Cole takes either salt or mustard only.

Many heuristics for labeling could be tried to solve the constraints above, rather than backtracking through all possibilities. However many inconsistent labelings can first be "culled" by a relaxation process if we convert the problem to a partial order programming problem. Let us permit the objects $\{u_i\}$ to be labeled with *sets* of labels instead of unique labels, the constraint

$$\langle u_{i_1}, \ldots, u_{i_m}\rangle \quad \in \quad P_{i_1 \cdots i_m}$$

becomes, for each k between 1 and m, the "separated" constraint

$$u_{i_k} \quad \subseteq \quad P_{i_1 \cdots i_m} \ltimes_k \left(u_{i_1} \times \cdots \times u_{i_{k-1}} \times u_{i_{k+1}} \times \cdots \times u_{i_m} \right)$$

where $\ltimes_k$, the k^{th}-semijoin operation, is defined by

$$P \ltimes_k u \quad = \quad \{ \; x_k \mid \langle x_1, \ldots, x_m\rangle \in P \text{ and } \langle x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m\rangle \in u \; \}.$$

For example, writing $\ltimes_1$ as $\ltimes$, the separated constraint for Barry looks like:

$$
\begin{array}{lll}
B \subseteq & (\Lambda \times \Lambda \times \Lambda - \{s,b\} \times \{n,b\} \times \{n,b\}) & \ltimes \; (C \times L) \\
\cap & (\Lambda \times \Lambda - \{n,m\} \times \{s,m\}) & \ltimes \; (C) \\
\cap & (\Lambda \times \Lambda - \{n,m\} \times \{s,m\}) & \ltimes \; (L) \\
\cap & (\Lambda \times \Lambda \times \Lambda - \{m,b\} \times \{s,m,b\} \times \{n,s,m\}) & \ltimes \; (D \times M) \\
\cap & (\Lambda \times \Lambda - \{n,s\} \times \{n\}) & \ltimes \; (D) \\
\cap & (\Lambda \times \Lambda - \{n,s\} \times \{b\}) & \ltimes \; (M) \\
\cap & (\Lambda \times \Lambda \times \Lambda - \{n,b\} \times \{s,b\} \times \{s,m,b\}) & \ltimes \; (C \times M) \\
\cap & (\Lambda \times \Lambda - \{s,m\} \times \{n,m\}) & \ltimes \; (C) \\
\cap & (\Lambda \times \Lambda \times \Lambda - \{s,m,b\} \times \{n,s,m\} \times \{s,b\}) & \ltimes \; (C \times D) \\
\cap & (\Lambda \times \Lambda - \{n\} \times \{n,m\}) & \ltimes \; (D) \\
\cap & (\Lambda \times \Lambda \times \Lambda - \{n,b\} \times \{n,b\} \times \{s,b\}) & \ltimes \; (D \times L) \\
\cap & (\Lambda \times \Lambda - \{s,m\} \times \{n,m\}) & \ltimes \; (L) \\
\cap & (\Lambda \times \Lambda \times \Lambda - \{n,s,m\} \times \{n,s,m\} \times \{s,b\}) & \ltimes \; (L \times M) \\
\cap & (\Lambda \times \Lambda - \{b\} \times \{n,m\}) & \ltimes \; (M)
\end{array}
$$

This new problem is easily stated as a partial order programming problem:

maximize	$\langle B,C,D,L,M\rangle$		
subject to	B	$\subseteq$	$\cdots$
	C	$\subseteq$	$\cdots$
	D	$\subseteq$	$\cdots$
	L	$\subseteq$	$\cdots$
	M	$\subseteq$	$\cdots$

where the $\cdots$ entries are the appropriate separated constraints.

A feature of this transformation of problems is that any solution of the original problem will be contained in the solution to this "approximation". Furthermore, this approximation of the problem permits iterative determination of maximal labeling sets that satisfy the separated constraints. Now, just as the original consistent labeling problem is NP-complete, this approximation is also [14]. However, the point is that the approximate problem with separated constraints permits iterative solution, and the result can then be used as a starting point for the original problem, or can be coupled with a search technique to do repeated culling.

The approximate consistent labeling problem is particularly nice on graphs, where it is sometimes called the *constraint network* problem. For each node i of a graph we associate an object u_i denoting its labels. These objects are subject to 'node constraints'

$$u_i \subseteq b_i$$

for each i, and 'arc constraints'

$$u_i \subseteq a_{ij} \ltimes u_j,$$

for each arc $<i,j>$ in the graph, where $a \ltimes u = \{ x \mid <x,y>$ is in a, and y is in $u \}$. These constraints can be combined to give a single constraint on each object, and the partial order programming problem

maximize	$<u_1, \ldots, u_n>$		
subject to	u_i	$\subseteq$	$\left(\bigcap_{j=1}^{n} a_{ij} \ltimes u_j \right) \cap b_i.$

Montanari points out furthermore [68] that the m-ary constraint $P_{i_1 \ldots i_m}$ in the general problem can be approximated by the $m(m-1)/2$ binary constraints of this form. It therefore follows that any consistent labeling problem can be approximated by a constraint network.

4.5. Polymorphic Type Inference for Prolog and Abstract Interpretation

Like Lisp, Prolog was designed as a language with only a single type: terms. In developing large Prolog programs, it is usually desirable to impose structure on the types actually handled by specific procedures. This is done by restricting the arguments of the procedure to meet certain type expressions.

Type expressions are terms constructed from type constructors (such as *int*, *list*, *pair*) and type variables (X, Y, Z, ...). For example,

list(X), list(pair(Y,Z)), list(list(int))

are type expressions. The first type expression here is *more general* than either the second or the third. We write this relationship of generality as

$$list(X) \supseteq list(pair(Y,Z)),$$
$$list(X) \supseteq list(list(int)).$$

We also introduce 'extended types' $<\tau_1, \ldots, \tau_n>$ and $<\tau_1, \ldots, \tau_n> \rightarrow \tau_{n+1}$, given types $\{\tau_i\}$ and $n \geq 0$, and extend $\supseteq$ to be the partial order of generality, or subsumption, on all of these types.

The partial ordering of generality among type expressions can be defined in terms of substitutions:

$\sigma \supseteq \tau$ if there exists a substitution θ such that $\sigma\theta = \tau$.

The constraint $\sigma \supseteq \tau$ can therefore be *satisfied* by finding a most general substitution θ such that $\sigma\theta = \tau\theta$, then replacing τ by $\tau\theta$ (and leaving σ intact).

Chou [23] has shown how type inference for a Prolog program reduces to solution of a set of inequalities on these type expressions. Given a program P, a *typing* of P consists of a set of *type attachments* assigning a non-extended type to each term and subterm in P, and *type premises* assigning a type term to each predicate, functor, and variable of P. These premises are written respectively as

$$p: <\tau_1, \ldots, \tau_n>$$
$$f: <\tau_1, \ldots, \tau_n> \rightarrow \tau_{n+1}$$
$$X: \tau.$$

A typing of P is a *well-typing* if it satisfies the following conditions. Let $p(t_1, \ldots, t_n)$ be a predicate and arguments appearing somewhere in a clause of P. If the typing has type premise $p: <\tau_1, \ldots, \tau_n>$ for p and assigns types $\sigma_1, \ldots, \sigma_n$ to the arguments $t_1, \ldots, t_n$, then:

(1) When $p(t_1, \ldots, t_n)$ is the *head* of the clause, the typing must satisfy

$$<\tau_1, \ldots, \tau_n> \supseteq <\sigma_1, \ldots, \sigma_n>$$
$$<\sigma_1, \ldots, \sigma_n> \supseteq <\tau_1, \ldots, \tau_n>.$$

In a well-typing, every clause agrees with the typing of its predicate.

(2) When $p(t_1, \ldots, t_n)$ is in the *body* of the clause, the typing must satisfy

$$<\tau_1, \ldots, \tau_n> \supseteq <\sigma_1, \ldots, \sigma_n>.$$

In a well-typing, every use of a predicate obeys the typing of the predicate.

For example, let us find a well-typing for the following Prolog program:

```
append(nil, L, L).
append(cons(X,L1), L2, cons(X,L3)) :- append(L1, L2, L3).
```

We use the functors nil and cons instead of the more common [] and [_|_] for the sake of clarity.

Assume that we begin with types for nil and cons, and the corresponding type premises:

nil:	$\rightarrow list(T)$
cons:	$<T, list(T)> \rightarrow list(T)$
append:	$<A, B, C>$
L:	D
X:	E
L1:	F
L2:	G
L3:	H

The unknown types $A, B, ..., H$ are then constrained as follows:

(C1.1)	$<A, B, C>$	$\sqsupseteq$	$<list(T), D, D>$
(C1.2)	$<list(T), D, D>$	$\sqsupseteq$	$<A, B, C>$
(C2.1)	$<A, B, C>$	$\sqsupseteq$	$<I, G, J>$
(C2.2)	$<I, G, J>$	$\sqsupseteq$	$<A, B, C>$
(C2.3)	$<T, list(T)> \rightarrow list(T)$	$\sqsupseteq$	$<E, F> \rightarrow I$
(C2.4)	$<T, list(T)> \rightarrow list(T)$	$\sqsupseteq$	$<E, H> \rightarrow J$
(C2.5)	$<A, B, C>$	$\sqsupseteq$	$<F, G, H>$

Here I and J are two more unknowns introduced for subexpressions. The partial order programming problem that determines the most general well-typing for the append predicate is thus

maximize	$<A,B,C>$
subject to	(C1.1)–(C2.5).

Among the inequalities of this problem, (C1.1), (C2.1), (C2.2) and (C2.5) are satisfied but (C1.2), (C2.3) and (C2.4) are not. We therefore force satisfaction of these constraints by repeatedly "relaxing" them, i.e., finding most general variable substitutions that satisfy their inequalities. Relaxing (C1.2), we obtain the substitution

$$A := list(T1), \quad B := D1, \quad C := D1,$$

and the constraints are transformed to:

(C1.1)	$<list(T1), D1, D1>$	$\sqsupseteq$	$<list(T), D, D>$
(C1.2)	$<list(T), D, D>$	$\sqsupseteq$	$<list(T1), D1, D1>$
(C2.1)	$<list(T1), D1, D1>$	$\sqsupseteq$	$<I, G, J>$
(C2.2)	$<I, G, J>$	$\sqsupseteq$	$<list(T1), D1, D1>$
(C2.3)	$<T, list(T)> \rightarrow list(T)$	$\sqsupseteq$	$<E, F> \rightarrow I$
(C2.4)	$<T, list(T)> \rightarrow list(T)$	$\sqsupseteq$	$<E, H> \rightarrow J$
(C2.5)	$<list(T1), D1, D1>$	$\sqsupseteq$	$<F, G, H>$

Now (C2.1), (C2.3), (C2.4), and (C2.5) are not satisfied. Relaxing them in turn and relaxing (C2.2) again, we eventually arrive at the assignment

$$A := list(T5), \quad B := list(T5), \quad C := list(T5)$$

satisfying all of the inequalities. For this partial order programming problem, then, the assignments $A := list(T)$, $B := list(T)$, $C := list(T)$ give the solution. The type of append is correspondingly inferred to be

$$< list(T), list(T), list(T) >.$$

Type inference for a program is a simple kind of *abstract interpretation*, a general process of characterizing execution behavior of programs. In their seminal paper, the Cousots showed how global dataflow analysis, type checking, program testing, symbolic program evaluation, program performance analysis, verification of program correctness, discovery of program invariants, proof of program termination, etc. were special cases of abstract interpretation. Recently the subject has drawn interest in logic programming. For example, in [66] it is shown how a good deal of useful information about Prolog programs (mode declarations, determinacy, sharing of structures) can be obtained with abstract interpretation. Also [18] shows how "compile-time garbage collection" can be performed as abstract interpretation.

4.6. Rewriting and Reduction

Rewrite systems consist of sets of rules that are typically used to reduce an expression to another equivalent, but simplified, expression. A well-written summary of recent work is [56]. Consider the example in [43,44], where the following set of rewrite rules is presented for simplifying expressions over a Boolean algebra:

$$\begin{array}{lcl}
X \vee Y & \rightarrow & (X \wedge Y) \oplus X \oplus Y \\
X \supset Y & \rightarrow & (X \wedge Y) \oplus X \oplus \top \\
X \equiv Y & \rightarrow & X \oplus Y \oplus \top \\
\neg X & \rightarrow & X \oplus \top \\
X \oplus \bot & \rightarrow & X \\
X \oplus X & \rightarrow & \bot \\
X \wedge \top & \rightarrow & X \\
X \wedge X & \rightarrow & X \\
X \wedge \bot & \rightarrow & \bot \\
X \wedge (Y \oplus Z) & \rightarrow & (X \wedge Y) \oplus (X \wedge Z)
\end{array}$$

In the case where the Boolean algebra is ordinary logic, $\top$ is **true**, $\bot$ is **false**, and $\oplus$ is the 'exclusive-or' operator.

The idea is that, with commutativity and associativity of the operators taken into account, these rewrite rules define a schema of inequalities on Boolean expressions. For example, if we use the set of expressions involving $\wedge$, $\vee$, $\supset$, $\equiv$, $\neg$ over a set $\{p,q\}$ of propositional variables, the rules define a reduction (simplification) relationship among the objects, such as the following:

$$p \wedge (\neg(p \wedge q) \oplus q) \quad \rightarrow \quad (p \wedge \neg(p \wedge q)) \oplus (p \wedge q).$$

These reductions form the constraints of a partial order program. Specifically, the

objects are the Boolean expressions, and the values are *irreducible* expressions, expressions on which no rule can be used. The constraints are all the inequalities $u \sqsupseteq v$ such that $u \rightarrow v$, and the partial order $\sqsupseteq$ on values is therefore trivial. For example, if C represents all the reduction relationships $u \sqsupseteq v$ among objects, the problem

minimize	$p \wedge (\neg(p \wedge q) \oplus q)$
subject to	C

is solved by assigning the goal expression $p \wedge (\neg(p \wedge q) \oplus q)$ the (irreducible) value p. This follows from the following sequence of reductions:

$$
\begin{aligned}
& p \wedge (\neg(p \wedge q) \oplus q) \\
\rightarrow\ & (p \wedge \neg(p \wedge q)) \oplus (p \wedge q) \\
\rightarrow\ & (p \wedge ((p \wedge q) \oplus \top) \oplus (p \wedge q) \\
\rightarrow\ & ((p \wedge p \wedge q) \oplus (p \wedge \top)) \oplus (p \wedge q) \\
\rightarrow\ & (p \wedge q) \oplus (p \wedge \top) \oplus (p \wedge q) \\
\rightarrow\ & (p \wedge \top) \\
\rightarrow\ & p
\end{aligned}
$$

The rewrite system here has nice properties. First, it is *confluent,* i.e., any sequence of reductions ultimately will produce the same result (up to associativity and commutativity of $\wedge$ and $\oplus$). Moreover, in [43] and very recently in [11] it is shown that this system can be used as the basis for a complete first-order logic automated theorem prover. (A system based on $\{\wedge,\vee,\neg\}$ cannot, as "minterm" representation of expressions is not unique.)

Traditionally rewrite systems are given a model theory obtained from some equality relation. It appears that frequently this reliance on equality can be replaced directly with inequality (reduction). Where the equality relation incorporates the axioms of symmetry and substitution of equals for equals, the inequality relation uses antisymmetry and "substitution of lessers for greaters in monotone contexts". Since equational systems are often intractable computationally, this approach may provide some useful results.

4.7. Logic and Lattice Programming

We noted earlier that all logical implications are inequalities, and that the logical implication $P \leftarrow Q$ is equivalent to the constraint "*truth*(P)" $\sqsupseteq$ "*truth*(Q)". With this insight we can view logic programming as a kind of partial order programming. Horn rules are inequalities: the rule $H \leftarrow G$ may be written as $H \sqsupseteq G$. We will work this out in detail later, but basically here our objects are predicates taking values over the lattice {**true**,**false**}, and the familiar $\wedge$ and $\vee$ operators are the greatest lower bound and least upper bound operators on this lattice.

We can capitalize on this insight by permitting predicates to take values other than **true** and **false**. This generalizes logic programming to take on a functional flavor. For example, we can write the append predicate functionally as

append([ ], L, L) ≥ **true**.
append([X|L1], L2, [X|L3]) ≥ append(L1, L2, L3).

so that the object append([a,b],[c],[a,b,c]) will take the value **true**. We could also write the predicate as

append([ ], L) ≥ L.
append([X|L1], L2) ≥ [X | append(L1, L2)].

so that the object append([a,b],[c]) will take the value [a,b,c]. The scope of the approach can range from an implementation intent on exploiting Prolog such as [33, 71], to a slower system with more general reduction capability.

A useful application of this approach is given by van Emden [34], who investigates an extension of logic programming in which the semantics of a logic program are generalized from true-false assignments to assignments of real values between 0 and 1, which can be viewed as representing some kind of certainty factor. The quantitative logic program

$$
\begin{array}{lll}
a & \leftarrow 0.50 - & b \,\&\, f \\
a & \leftarrow 0.50 - & c \,\&\, d \\
b & \leftarrow 0.20 - & \\
c & \leftarrow 0.45 - & \\
c & \leftarrow 0.30 - & \\
d & \leftarrow 1.00 - & \\
e & \leftarrow 0.50 - & \\
f & \leftarrow 0.90 - & e
\end{array}
$$

has semantics assigning the object b the value (greatest lower bound) 0.20, c the value max(0.45, 0.30) = 0.45, and a the value

max(0.50 * min(0.20, 0.90*(0.50)), 0.50 * min(0.45, 1.00)) = 0.225,

because the '&' operator is defined to operate like 'min' and different clauses are combined with 'max', with the various attenuation factors multiplied in. Thus the goal $\leftarrow a$ will obtain the value 0.225.

The corresponding partial order programming problem makes the semantics of this program evident:

minimize	a		
subject to	a	$\geq$	0.50 * min(b, f)
	a	$\geq$	0.50 * min(c, d)
	b	$\geq$	0.20
	c	$\geq$	0.45
	c	$\geq$	0.30
	d	$\geq$	1.00
	e	$\geq$	0.50
	f	$\geq$	0.90 * e

This equivalence has been noted independently by O'Keefe [73].

We can generalize logic programming to *lattice programming* by replacing the usual {**true**,**false**} value domain of Prolog with any lattice and its particular $\wedge$, $\vee$ operations. Fitting shows how Ginsberg's bilattices can be used as an interesting generalized space of truth values for logic programs [36]. Although this generalization gives no real increase in *power* over logic programming, since predicates can always generate values with a supplemental argument, this generalization does result in *convenience* in many situations (reduction computations, inexact reasoning, error handling, etc.), and declarative semantics for such a system may be derived straightforwardly.

4.8. Numerical Iteration

There is a deep connection between solution of inequalities, mathematical programming, ordering, iteration, and computation in general. Suppose that, given a number b such that $0 < b < 1$, we wish to find x satisfying the following conditions:

minimize	x		
subject to	x	$\geq$	1
	x	$\geq$	$x\,(2 - bx)$.

This partial order programming problem can be solved iteratively. If initially x is assigned the value 1 to satisfy the first inequality, repeatedly assigning $x := x(2-bx)$ then causes the value of x to increase until it reaches $1/b$ and satisfies the second. (This iteration is often used to perform numerical inversion in computer arithmetic units.)

Root finding frequently amounts to iterative solution of inequalities. Given a function f defined over domain D, we wish to find an x in D that meets the constraint $f(x) = 0$ (equivalently, $f(x) \geq 0$ and $f(x) \leq 0$). Often this may be done using Newton-Raphson iteration

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

with a suitable initial value $x^{(0)}$. Specifically, if the initial value is in a region where f/f' is non-negative, monotone decreasing, continuous, and bounded by $x^{(0)}$, the

iteration gives a monotonically increasing sequence $x^{(0)} \le x^{(1)} \le \cdots$ converging quadratically to the root. This is then equivalent to finding the least solution x to the system

minimize	x		
subject to	x	$\ge$	$x^{(0)}$
	x	$\ge$	$x - f(x) / f'(x)$.

For example, with $f(x) = 1/x - b$ and $x^{(0)} = 1$ we get the system of inequalities for numerical inversion above. This equivalence follows directly from symbolically taking limits of both sides of the Newton-Raphson iteration, using the property of limits of continuous monotone functions, e.g.,

$$\lim f(x^{(i)}) \;=\; f(\lim x^{(i)}) \;=\; f(x).$$

The iteration we use depends directly on the ordering defined by the constraints. This can be demonstrated dramatically. Suppose we are given the partial order programming problem

minimize	$\langle x, y \rangle$		
subject to	$\langle 1, k_0 \rangle$	$\sqsupseteq$	$\langle x, y \rangle$.

where k_0 is a value in [0,1], and $\sqsupseteq$ is the reflexive transitive closure of the relation

$$\langle u, v \rangle \;\sqsupseteq\; \langle \tfrac{1}{2}(u + v), \sqrt{uv} \rangle$$

defined on the domain [0,1]×[0,1]. Taking $x = 1$, $y = k_0$ gives a starting value that can be refined by repeatedly executing $\langle x,y \rangle := \langle \tfrac{1}{2}(x + y), \sqrt{xy} \rangle$. Since initially $x \ge y$, this iteration must reach a fixed point, because the arithmetic mean $\tfrac{1}{2}(x + y)$ and geometric mean $\sqrt{xy}$ then satisfy $x \ge \tfrac{1}{2}(x + y) \ge \sqrt{xy} \ge y$. The resulting value for $\langle x,y \rangle$ is the least possible under $\sqsupseteq$.

This iteration is known as the *arithmetic-geometric mean* (AGM) algorithm, and was developed by Gauss at the age of fourteen. It is a quadratically convergent method for computing the elliptic integral

$$K(m) \;=\; \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - m \sin^2\theta}}$$

for $0 \le m \le 1$. With the initial values $x = 1$, $y = \sqrt{1-m}$, both x and y converge to the same value $\pi/(2K(m))$ under the iteration.

The point here is that the vector inequality for $\sqsupseteq$ cannot be decoupled into inequalities on individual variables, and the vector assignment $\langle x,y \rangle := \langle \tfrac{1}{2}(x+y), \sqrt{xy} \rangle$ cannot be decoupled into the two assignments

$$A_1 : \quad x \;:=\; \tfrac{1}{2}(x + y)$$
$$A_2 : \quad y \;:=\; \sqrt{xy}.$$

To establish this, note that the AGM vector assignment iteration computes, for $i \ge 0$,

$$< x^{(i+1)}, y^{(i+1)} > \;=\; < \tfrac{1}{2}(x^{(i)} + y^{(i)}), \sqrt{x^{(i)}y^{(i)}} >$$

decreasing the value of the vector under $\sqsupseteq$ at each step. However, applying assignments in the order $A_1\, A_2\, A_1\, A_2\, A_1 \cdots$ computes

$$< x^{(i+1)}, y^{(i+1)} > \;=\; < \tfrac{1}{2}(x^{(i)} + y^{(i)}), \sqrt{x^{(i+1)}y^{(i)}} >$$

and in this case for $m > 0$ it can be shown from results in [20] that

$$\frac{1}{\lim x^{(i)}} \;=\; \frac{1}{\lim y^{(i)}} \;=\; \frac{1}{\sqrt{m}} \operatorname{arccosh}\left[\frac{1}{\sqrt{1-m}} \right].$$

Furthermore, if $m < 1$ and we apply assignments in the order $A_2\, A_1\, A_2\, A_1\, A_2 \cdots$, then

$$\frac{1}{\lim x^{(i)}} \;=\; \frac{1}{\lim y^{(i)}} \;=\; \frac{1}{2\sqrt{1-\sqrt{1-m}}} \ln\left[\frac{1 + \sqrt{1-\sqrt{1-m}}}{1 - \sqrt{1-\sqrt{1-m}}} \right].$$

This simple demonstration shows how iterations must follow the inequality constraints of the problem. This is not an artificial example; important similar iterations, such as the CORDIC iteration [30], yield the same conclusion. It is also evident from the example that the multiple assignment statements

$$u_1, \ldots, u_n \;:=\; v_1, \ldots, v_n$$

in UNITY [22] are important.

5. Semilinear Partial Order Programming

Many of the examples of partial order programming problems shown above are somehow linear in nature. This is not too surprising, since almost all of the problems use only continuous monotone operators, and continuous monotone functions can usually be "linearized", i.e., transformed to linear functions with some suitable changes of variables. Also, many of the problems are path problems by nature. In this section we clarify this linearity using the algebraic framework of semimodules, an elegant generalization of vector spaces, fields, rings, etc.

Linearity is interesting not only because it provides a new point of view, but also because it permits us to draw on the vast arsenal of known algorithms for solving linear algebraic systems. Specifically we can use elimination methods to find solutions, such as Gaussian or Gauss-Jordan elimination, as well as relaxation methods.

5.1. Linearizable Problems

Some of the examples considered earlier have identical structure. The discretized Laplace equation imposes the linear constraints

$$u_i = \frac{u_{left(i)} + u_{right(i)} + u_{up(i)} + u_{down(i)}}{4}$$

for all points i in the interior of a grid, the single-source shortest paths problem has least-cost path constraints

$$u_i \leq \min\left(\left[\min_{1 \leq j \leq n} a_{ij} + u_j \right] , a_{i0} \right)$$

for $i \neq 0$, and the (approximate) graph consistent labeling problem has labeling constraints

$$u_i \subseteq \left[\bigcap_{j=1}^{n} a_{ij} \ltimes u_j \right] \cap b_i.$$

Each of these three well-known problems is, in fact, a *linear inequality system.* That is, for each problem there are binary operators $\boxplus$ and $\boxtimes$ with which the problem can be expressed as the vector constraint

$$\mathbf{u} \sqsupseteq \mathbf{A} \boxplus.\boxtimes \mathbf{u} \boxplus \mathbf{b}$$

where $\boxplus.\boxtimes$ is the *generalized inner product* available in the APL programming language, using the infix operators $\boxplus$ for (commutative, associative) addition and $\boxtimes$ for multiplication. If $X = (x_{ij})$ and $Y = (y_{ij})$ are matrices respectively of size $m \times n$ and $n \times p$, the generalized inner product $Z = X \boxplus.\boxtimes Y$ is a matrix of size $m \times p$ defined by

$$z_{ij} = \boxplus_{k=1}^{n} x_{ik} \boxtimes y_{kj}.$$

The following table shows that $\boxplus$ is associative and commutative:

	Laplace's Equation	*Shortest Paths*	*Consistent Labeling*
$\sqsupseteq$	numeric $\geq$ and $\leq$	numeric $\geq$	set inclusion $\subseteq$
$\boxplus$	numeric addition	numeric minimum	set intersection
$\boxtimes$	numeric multiplication	numeric addition	relational semijoin

These three examples are not close to being exhaustive. For example, Zimmermann [95] lists many other problems that easily fit in this framework: path reliability, path connectivity, maximum capacity paths, k-shortest paths, regular expressions, word abbreviations, path and cutset enumeration, and certain scheduling problems. The examples are representative, however. Most published applications of relaxation reduce either to numeric linear systems, numeric path problems, or symbolic path problems.

A simple result showing that many partial order programs are linearizable can be derived for *continuous monotone constraint networks,* algebraic partial order programs in which at most two objects participate in any inequality.

Theorem 10
To every partial order program over the complete lattice $<D,\sqsupseteq>$ with objects $B = \{u_i \mid 1 \leq i \leq n\}$ and constraints

$$u_i \sqsupseteq b_i \qquad 1 \leq i \leq n,$$
$$u_i \sqsupseteq a_{ij} \bullet u_j \qquad 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j,$$

where the a_{ij} are continuous functions, there corresponds a $n \times n$ matrix $\mathbf{A}$, an n-vector $\mathbf{b}$, and binary continuous monotone operators $\boxplus$, $\boxtimes$ on D such that $\mathbf{u} = (u_i)$ satisfies all the constraints if and only if it also is a solution of

$$\mathbf{u} \sqsupseteq \mathbf{A} \boxplus.\boxtimes \mathbf{u} \boxplus \mathbf{b}.$$

Proof Let $\boxplus$ be the least upper bound $\bigsqcup$, and let $\boxtimes$ be $\bullet$ (functional composition). Thus $\boxtimes : ((D \to D) \times D) \to D$ is continuous monotone in its second argument when its first argument is continuous monotone in all occurrences here. Since $\boxplus$ is also continuous monotone, commutative and associative, the combination of the constraints is continuous monotone, and is equivalent to the program $\mathbf{u} \sqsupseteq \mathbf{A} \boxplus.\boxtimes \mathbf{u} \boxplus \mathbf{b}$. □

5.2. Ordered Semimodules and Semilinear Programming

The result of Theorem 10 is not the most general possible when we can say more about the coefficients a_{ij}. Specifically, when the values a_{ij} are taken from a special kind of semiring, and the objects u_j come from a vector-space-like structure matching this semiring, the least solution of the inequalities has an explicit form. The theorem below generalizes similar results for "closed semirings" in [1, 37, 57] to take into account the semimodule structure underlying many partial order programs.

The simple algebraic setting of semimodules generalizes all of the linear relaxation

examples in the previous section. While readers unaccustomed to the generality may find it challenging to appreciate at first, it is worth the effort: many important problems fall out as simple special cases.

Definition
A *monoid* is a system $<R,\theta,e>$ such that:

(M1) θ is closed on R $(\theta: R\times R\rightarrow R)$.
(M2) θ is associative.
(M3) e is an identity for θ.

Definition
A *semiring* is a system $<R,\oplus,\otimes,0,1>$ satisfying the following conditions:

(SR1) $<R,\oplus,0>$ is a monoid.
(SR2) $<R,\otimes,1>$ is a monoid.
(SR3) $\oplus$ is commutative.
(SR4) $\otimes$ distributes over $\oplus$: for all $x,y,z \in R$,

$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z),$$
$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z).$$

(SR5) 0 is a multiplicative annihilator on R: for all $x \in R$,

$$0 \otimes x = 0$$
$$x \otimes 0 = 0.$$

In particular, every lattice is a semiring $<D,\wedge,\vee,\bot,\top>$.

Definition
A *semimodule* (a left R-semimodule) is a system $<R,\oplus,\otimes,0,1, D,\boxplus,\boxtimes>$ satisfying the following conditions:

(SM1) $<R,\oplus,\otimes,0,1>$ is a semiring.
(SM2) $\boxtimes: R \times D \rightarrow D$.
(SM3) 1 is a left identity for $\boxtimes$: for all $x \in D$,

$$1 \boxtimes x = x.$$

(SM4) $\boxtimes$ distributes over $\boxplus$: for all $a \in R$, $x,y \in D$,

$$a \boxtimes (x \boxplus y) = (a \boxtimes x) \boxplus (a \boxtimes y).$$

(SM5) $\boxtimes$ distributes over $\oplus$ as $\boxtimes$: for all $a,b \in R$, $x \in D$,

$$(a \oplus b) \boxtimes x = (a \boxtimes x) \boxplus (b \boxtimes x).$$

(SM6) $\otimes$ associates with $\boxtimes$ as $\boxtimes$: for all $a,b \in R$, $x \in D$,

$$(a \otimes b) \boxtimes x = a \boxtimes (b \boxtimes x).$$

Definition

A *complete ordered semiring (cosr)* is a system $<R,\geq,\oplus,\otimes,0,1>$ satisfying the following conditions:

(COSR1) $<R,\geq>$ is a pointed cpo, with least element 0.
(COSR2) $<R,\oplus,\otimes,0,1>$ is a semiring.
(COSR3) $\oplus$ is monotone: for all $a,b,c \in R$,

$$a \geq b \quad \text{implies} \quad (a \oplus c) \geq (b \oplus c),$$
$$(c \oplus a) \geq (c \oplus b).$$

(COSR4) $\otimes$ is monotone: for all $a,b,c \in R$,

$$a \geq b \quad \text{implies} \quad (a \otimes c) \geq (b \otimes c),$$
$$(c \otimes a) \geq (c \otimes b).$$

Definition
A *complete ordered semimodule (cosm)* is a system $<R,\geq,\oplus,\otimes,0,1, D,\sqsupseteq,\boxplus,\boxtimes>$ satisfying the following conditions:

(COSM1) $<D,\sqsupseteq>$ is a cpo.
(COSM2) $<R,\geq,\oplus,\otimes,0,1>$ is a complete ordered semiring.
(COSM3) $<R,\oplus,\otimes,0,1, D,\boxplus,\boxtimes>$ is a semimodule.
(COSM4) $\boxplus$ is monotone: for all $x,y,z \in D$,

$$x \sqsupseteq y \quad \text{implies} \quad (x \boxplus z) \sqsupseteq (y \boxplus z),$$
$$(z \boxplus x) \sqsupseteq (z \boxplus y).$$

(COSM5) $\boxtimes$ is monotone: for all $a,b \in R$, $x \in D$,

$$a \geq b \quad \text{implies} \quad (a \boxtimes x) \sqsupseteq (b \boxtimes x),$$

and for all $a \in R$, $x,y \in D$,

$$x \sqsupseteq y \quad \text{implies} \quad (a \boxtimes x) \sqsupseteq (a \boxtimes y).$$

When $\boxplus$, $\boxtimes$, $\oplus$, and $\otimes$ are *continuous* as well as monotone, we have a *continuous complete ordered semimodule (ccosm).*

Finally, if a ccosm is pointed, satisfying the following restrictions, we obtain a *pointed continuous complete ordered semimodule (pccosm):*

(PCCOSM1) $<R,\geq,\oplus,\otimes,0,1, D,\sqsupseteq,\boxplus,\boxtimes>$ is a ccosm.
(PCCOSM2) $<D,\sqsupseteq>$ is a pointed cpo, with least element $\bot$.
(PCCOSM3) $<D,\boxplus,\bot>$ is a monoid.
(PCCOSM4) 0 is a left multiplicative annihilator for $\boxtimes$: for all $x \in D$,

$$0 \boxtimes x = \bot.$$

(PCCOSM5) $\bot$ is a right multiplicative annihilator for $\boxtimes$: for all $a \in R$,

$$a \boxtimes \bot = \bot.$$

Theorem 11
Let $<R,\geq,\oplus,\otimes,0,1,\ D,\sqsupseteq,\boxplus,\boxtimes>$ be a pccosm. Suppose $\mathbf{u} = (u_i)$ satisfies the inequality

$$\mathbf{u} \ \sqsupseteq \ \mathbf{A} \boxplus.\boxtimes \mathbf{u} \ \boxplus \ \mathbf{b},$$

where b_i is in D and a_{ij} is in R, $1 \leq i,j \leq n$. Then there is an $\mathbf{A}^*$ in $R^{n\times n}$ such that

$$\mathbf{u} \ = \ \mathbf{A}^* \boxplus.\boxtimes \mathbf{b}$$

is a least solution of the inequality.

Proof
Define the closure $\mathbf{A}^*$ by

$$\mathbf{A}^* \ = \ \bigoplus_{k\geq 0} \mathbf{A}^k \ = \ \mathbf{A}^0 \oplus \mathbf{A}^1 \oplus \mathbf{A}^2 \oplus \cdots$$

where $\mathbf{A}^0 = \mathbf{I}$ is the identity matrix (δ_{ij}) on R, with entries

$$\delta_{ij} \ = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise,} \end{cases}$$

and $\mathbf{A}^k \ = \ \mathbf{A} \oplus.\otimes \mathbf{A}^{k-1}$, for $k>0$. Using the properties of a pccosm, we can show by induction on K that

$$\left[\bigoplus_{k=0}^{K} \mathbf{A}^k \right] \boxplus.\boxtimes \mathbf{b} \ = \ \left[\bigboxplus_{k=0}^{K} \mathbf{A}^k \boxplus.\boxtimes \mathbf{b} \right],$$

and that

$$\left[\bigboxplus_{k=0}^{K} \mathbf{A}^k \boxplus.\boxtimes \mathbf{b} \right] \ = \ \mathbf{f}^{K+1}(\bot\!\!\!\bot)$$

where $\mathbf{f}(\mathbf{u}) \ = \ (\mathbf{A} \boxplus.\boxtimes \mathbf{u} \ \boxplus \ \mathbf{b})$. SM3, PCCOSM3, and PCCOSM4 are used to prove $\mathbf{A}^0 \boxplus.\boxtimes \mathbf{b} \ = \ \mathbf{b}$, SM5 to convert $\oplus$ to $\boxplus$, SM6 to convert $\otimes$ to $\boxtimes$, and PCCOSM3–5 to show $\mathbf{f}(\bot\!\!\!\bot) = \mathbf{b}$. Now $\mathbf{f}$ is continuous monotone (by COSR3–4, COSM4–5) on D^n, which is a cpo since D is. Therefore $\{\mathbf{f}^{K+1}(\bot\!\!\!\bot) \mid K \in \omega\}$ is an ascending chain with a least upper bound in D^n, and

$$\mathbf{A}^* \boxplus.\boxtimes \mathbf{b} \ = \ \left[\bigoplus_{k\geq 0} \mathbf{A}^k \right] \boxplus.\boxtimes \mathbf{b} \ = \ \lim_{K} \left[\bigboxplus_{k=0}^{K} \mathbf{A}^k \boxplus.\boxtimes \mathbf{b} \right] \ = \ \bigsqcup_{K \in \omega} \mathbf{f}^{K+1}(\bot\!\!\!\bot).$$

Thus

$$\mathbf{A}^* \boxplus.\boxtimes \mathbf{b} \ = \ \left[\bigboxplus_{k\geq 0} \mathbf{A}^k \boxplus.\boxtimes \mathbf{b} \right].$$

So the value $\mathbf{u} \ = \ \mathbf{A}^* \boxplus.\boxtimes \mathbf{b}$ satisfies

$$\begin{aligned} \mathbf{A} \boxplus.\boxtimes \mathbf{u} \ \boxplus \ \mathbf{b} \ &= \ \mathbf{A} \boxplus.\boxtimes \left[\mathbf{A}^* \boxplus.\boxtimes \mathbf{b} \right] \ \boxplus \ \mathbf{b} \\ &= \ \mathbf{A} \boxplus.\boxtimes \left[\bigboxplus_{k\geq 0} \mathbf{A}^k \boxplus.\boxtimes \mathbf{b} \right] \ \boxplus \ \mathbf{b} \\ &= \ \left[\bigboxplus_{k\geq 0} \mathbf{A}^k \boxplus.\boxtimes \mathbf{b} \right] \\ &= \ \mathbf{u}. \ \square \end{aligned}$$

The result above on continuous complete ordered semimodules is very general! To help make it more concrete, we show how Theorem 11 can be applied for the problems mentioned earlier in this section. Since the main issue here is how to phrase the various problems as inequalities over a pccosm, we present a table giving a translation for each problem, extending the brief table given earlier:

	Laplace's Equation	*Single-Source Shortest Paths*	*Consistent Labeling*
R	closed interval [0,1]	nonnegative reals $\mathbf{R}_+ \cup \{\infty\}$	valid subsets of $\Lambda\times\Lambda$
$a \geq b$	$a \geq b$	$a \leq b$	$a \supseteq b$
$a \oplus b$	$a + b$	$\min(a,b)$	$a \cap b$
$a \otimes b$	$a \times b$	$a + b$	$a \bowtie b$
0	0	∞	$\Lambda \times \Lambda$
1	1	0	$\{ \langle x,x\rangle \mid x \in \Lambda \}$
D	nonnegative reals $\mathbf{R}_+ \cup \{\infty\}$	nonnegative reals $\mathbf{R}_+ \cup \{\infty\}$	nonempty subsets of Λ
$x \sqsupseteq y$	$x \geq y$	$x \leq y$	$x \subseteq y$
$x \boxplus y$	$x + y$	$\min(x,y)$	$x \cap y$
$a \boxtimes x$	$a \times x$	$a + x$	$a \ltimes x$
$\perp$	0	∞	Λ
$\mathbf{A} = (a_{ij})$	$a_{ij} = \begin{cases} 1/4 & i \text{ interior, } j \text{ adjacent} \\ 0 & \text{otherwise} \end{cases}$	matrix of arc costs	matrix of permissible arc label sets
$\mathbf{b} = (b_i)$	$b_i = \begin{cases} u_i & i \text{ boundary} \\ 0 & \text{otherwise} \end{cases}$	vector with entries ∞, except 0 for source node	vector of permissible node label sets

The terminology for the approximate graph consistent labeling problem needs some explanation. As before, $\ltimes$ is a relational semijoin operator [14] defined by

$$a \ltimes x = \{ p \mid \langle p,q\rangle \in a,\ q \in x \},$$

while $\bowtie$ is a relational join operator:

$$a \bowtie b = \{ \langle p,r\rangle \mid \langle p,q\rangle \in a,\ \langle q,r\rangle \in b \}.$$

(Order of the arguments is significant in both of these operators.) Also, a subset a of $\Lambda \times \Lambda$ is *valid* if for each element p in Λ, there are elements $\langle p,q\rangle \in a$ and $\langle r,p\rangle \in a$. Without restricting R to valid subsets, $\langle R,\oplus,\otimes,0,1\rangle$ is not a semiring, since 0 fails to be a multiplicative annihilator.

In any event we see that the "semi-linear programming" problem

minimize	$\mathbf{u}$			
subject to	$\mathbf{u}$	$\sqsupseteq$	$\mathbf{A} \boxplus.\boxtimes \mathbf{u}$	$\boxplus$ $\mathbf{b}$

is solvable either by elimination or by relaxation, when $\mathbf{A}$ is a square matrix. When $\mathbf{A}$ is not a square matrix, relaxation techniques still apply, but a great deal of work remains in understanding how such problems can be solved in general.

The excellent survey by Zimmermann [95] reviews known results in exactly this setting for numerous problems in combinatorial optimization: path, eigenvalue, linear programming, network flow, and independent set problems. Interesting forms of duality can also be expressed on semimodules. Note also that in two of the three problems above, $\otimes$ is commutative, so $<R,\oplus,\otimes,0,1>$ is a commutative semiring, and R is a subset of the real numbers. Zimmermann shows how each of these common situations has been exploited with specific constraint satisfaction algorithms.

6. Partial Order Programming and Computer Programming

A significant feature of partial order programming is the perspective it gives on different programming paradigms. First, it offers a way to abstract the essentials of both least fixed point and procedural semantics of different paradigms. Second, it offers a framework for investigating new ways to blend paradigms.

In this section we show how the logic and functional programming paradigms can be expressed as continuous monotone partial order programming, where C is a reduction relation on expressions or terms. Demonstrations for other 'pure' paradigms, such as the message-passing and inheritance kernel of object-oriented programming, will be similar, but we omit them in this presentation. We concentrate instead on how functional and logic programming can be integrated.

6.1. Least Fixed Point Semantics and Partial Order Programming

Any recursive programming paradigm with least fixed point semantics can be presented as partial order programming. This is immediate. Least fixed point semantics assign to each program π a continuous monotone functional $T_\pi : D \rightarrow D$ on some space D of semantic "interpretations" of the program. These interpretations are typically subsets of the input-output relation defined by the program. The smallest interpretation I satisfying $I = T_\pi(I)$ gives the least fixed point semantics for π. See [62, 82].

Given π then, consider the partial order programming problem

minimize	I
subject to	$I \sqsupseteq T_\pi(I)$

where the program is $P = \langle B, C, D, \sqsupseteq \rangle$ with

B: $Term(\{T_\pi\}, \{I\} \cup D)$
C: T_π
D: interpretations of π
$\sqsupseteq$: inclusion ($\subseteq$) among interpretations.

Thus B is $\{ T_\pi^k(x) \mid k \in \omega, x = I \text{ or } x \in D \}$, and $\langle D, \sqsupseteq \rangle$ is a complete lattice whose least element is the least fixed point of T_π. The least model is that valuation assigning I the limiting fixed point value

$$\bigsqcup_k T_\pi^k(\bot).$$

Any programming paradigm with least fixed point semantics is in this abstract sense a special case of partial order programming.

6.2. Functional Programming

We adopt the lambda calculus as a prototypical functional programming language. The lambda calculus has many different procedural semantics, just as it has many different models [12]. One semantics is captured here with partial order programming. We introduce terminology rapidly; for a very readable presentation, see [92].

Letting V be a countable set of *variables* $\{x,y,z, \cdots\}$, *terms* are defined inductively:

(1) Every variable in V is a term.
(2) If M and N are terms, the *combination* $(M\ N)$ is also.
(3) If x is a variable and M is a term, the *abstraction* $(\lambda x\ .\ M)$ is also.

The λ-*terms* are defined to be the equivalence classes under α-congruence of these terms. Two terms are α-*congruent* if one can be converted to the other by renaming of bound variables (of abstractions). A variable x occurs *free* in a λ-term if it does not appear within a subterm $(\lambda x\ .\ \cdots)$; otherwise x is *bound*. A λ-term with no free variables is called a *combinator*.

The λ-term $((\lambda x\ .\ M)\ N)$ is β-*reducible* to the term $M[x{:=}N]$, i.e., M with free occurrences of x replaced by N. Free variables in N are renamed if necessary to avoid being bound. The λ-term $((\lambda x\ .\ M)\ x)$ is η-*reducible* to the term M, provided that x does not occur free in M.

Terms are "evaluated" by eliminating λ-abstractions using reduction. One λ-term is *reducible* to another if there is a sequence of β- or η-reductions from one to the other. The *normal-order reduction* of one term to another is a sequence of leftmost reductions. In the λ-term

$$\lambda x_1\ \cdots\ \lambda x_m\ .\ (\ \cdots\ (((\lambda x\ .\ M)\ N)\ M_1\ )\ \cdots\ M_n\ ),$$

the reduction of $((\lambda x\ .\ M)\ N)$ is called *head reduction*. Every sequence of head reductions is thus a normal-order reduction. A λ-term is a *normal form* if it cannot be reduced. If z is a variable, a λ-term

$$\lambda x_1\ \cdots\ \lambda x_m\ .\ (\ \cdots\ (\ z\ M_1\ )\ \cdots\ M_n\ ) \qquad (m \geq 0,\ n \geq 0)$$

is called a *head normal form*. Every normal form is a head normal form, where each of the λ-terms M_i is in turn a normal form. The significance of head normal form is that some important λ-terms such as the combinator

$$\Theta\ \equiv\ (\ (\lambda x\ .\ \lambda y\ .\ y\ ((x\ x)\ y))\ \ (\lambda x\ .\ \lambda y\ .\ y\ ((x\ x)\ y))\ )$$

are reducible to a head normal form but no normal form, and head normal form captures exactly the notion of 'solvability' of λ-terms [92, 93].

To avoid writing out large terms, we use the symbol $\equiv$ to introduce names for combinators. For example, writing

$$\begin{aligned} \Theta\ &\equiv\ (\mathbf{A}\ \mathbf{A}) \\ \mathbf{A}\ &\equiv\ (\lambda x\ .\ \lambda y\ .\ y\ ((x\ x)\ y)) \end{aligned}$$

is equivalent to the definition for Θ above.

Θ has the nice property that if F is any λ-term, then $(\Theta\ F)$ *reduces* to $F(\Theta\ F)$. (This makes Θ a *fixed point combinator*. Although $\mathbf{Y} \equiv (\lambda f\ .\ f((\lambda x\ .\ f(x\ x))(\lambda x\ .\ f(x\ x))))$ is a better-known fixed point combinator, it turns out not to have this reduction property.

See [12]; (**Y** *F*) is λ-*convertible* to *F*, not λ-reducible to *F*.) Specifically, if ⊒ represents reduction, (Θ *F*) has the following reduction:

$$\begin{aligned}(\Theta\ F) &\equiv (\mathbf{A}\ \mathbf{A})\ F \equiv ((\lambda x\ .\ \lambda y\ .\ y\ ((x\ x)\ y))\ \mathbf{A})\ F \\ &\sqsupseteq (\lambda y\ .\ y\ ((\mathbf{A}\ \mathbf{A})\ y))\ F \\ &\sqsupseteq F\ ((\mathbf{A}\ \mathbf{A})\ F).\end{aligned}$$

The final term is *F* (Θ *F*).

Combinators provide a computational structure for the lambda calculus. For example, for factorials we can define:

$\Theta \equiv (\mathbf{A}\ \mathbf{A})$
$\mathbf{A} \equiv \lambda x\ .\ \lambda y\ .\ (y\ ((x\ x)\ y))$
factorial ≡ (Θ **fact**)
fact ≡ λ*f* . λ*n* . (**if** (**zero** *n*) 1 (**times** *n* (*f* (**pred** *n*)))).

Expressions using these combinators can then be evaluated reductively. For example, the term (**factorial** 1) can be reduced to the value 1 as follows:

(**factorial** 1)
((Θ **fact**) 1)
((**fact** (Θ **fact**)) 1)
(((λ*f* . λ*n* . (**if** (**zero** *n*) 1 (**times** *n* (*f* (**pred** *n*)))))) (Θ **fact**)) 1)
((λ*n* . (**if** (**zero** *n*) 1 (**times** *n* ((Θ **fact**) (**pred** *n*)))))) 1)
(**if** (**zero** 1) 1 (**times** 1 ((Θ **fact**) (**pred** 1)))))
(**times** 1 ((Θ **fact**) (**pred** 1)))
(**times** 1 ((**fact** (Θ **fact**)) 0))
(**times** 1 (((λ*f* . λ*n* . (**if** (**zero** *n*) 1 (**times** *n* (*f* (**pred** *n*)))))) (Θ **fact**)) 0))
(**times** 1 ((λ*n* . (**if** (**zero** *n*) 1 (**times** *n* ((Θ **fact**) (**pred** *n*)))))) 0))
(**times** 1 ((**if** (**zero** 0) 1 (**times** 0 ((Θ **fact**) (**pred** 0)))))))
(**times** 1 ((**if true** 1 (**times** 0 ((Θ **fact**) (**pred** 0)))))))
(**times** 1 1)
1

For simplicity, we have omitted definitions and reduction sequences for **if**, **times**, **pred** (predecessor), and **zero**. Definitions for these combinators can be written given suitable λ-term representations for numerals including 0 and 1, such as:

0	≡	λ*x* . *x*
1	≡	(**succ** 0)
succ	≡	λ*x* . λ*z* . ((*z* **false**) *x*)
pred	≡	λ*x* . (*x* **false**)
zero	≡	λ*x* . (*x* **true**)
true	≡	λ*x* . λ*y* . *x*
false	≡	λ*x* . λ*y* . *y*
if	≡	λ*x* . *x*
plus	≡	(Θ (λ*plus* . λ*x* . λ*y* . (**if** (**zero** *x*) *y* (**succ** (*plus* (**pred** *x*) *y*)))))
times	≡	(Θ (λ*times* . λ*x* . λ*y* . (**if** (**zero** *x*) 0 (**plus** (*times* (**pred** *x*) *y*) *y*))))

With these definitions **succ** operates as a successor combinator, **pred** as predecessor,

and the application of **zero** to 0 (respectively any successor of 0) reduces to **true** (respectively **false**). See chapter 6 of [12]. Also, following the convention that combination associates to the left, (**if true** *a b*) is (((**if true**) *a*) *b*), and reduces to *a*.

At this point, we can express the reduction semantics of the lambda calculus with a suitable continuous monotone partial order program. The program is constructed from the following components:

- Recalling that V is the set of variables, let $\perp$ be a distinguished new symbol and define $A = V \cup \{ \perp \}$. Let Op be the set of function symbols

$$\{ (\lambda x \,.\, \_) \mid x \in V \} \;\cup\; \{ (\_\ \_) \}.$$

- The value domain D is the set including the distinguished term $\perp$ and the head normal forms from the λ-terms $Term(Op,V)$.
- The set of objects is $B = Term(Op, A \cup D)$. We call these objects *partial λ-terms*.
- The partial order $\sqsupseteq$ on D is that of *lifting:* every term M in D satisfies $M \sqsupseteq \perp$. Operators in Op then correspond to continuous monotone operators on D:

  (1) For every x in A, the abstraction operator $(\lambda x \,.\, \_)$ corresponds to the mapping $\alpha_x : D \to D$ defined by

$$\alpha_x(M) \;=\; (\lambda x \,.\, M)$$

  if $M \neq \perp$, and otherwise $\alpha_x(\perp) \;=\; \perp$.

  (2) The combination operator $(\_\ \_)$ corresponds to the composition function $\bullet : D \times D \to D$ defined by reduction on λ-terms and

$$\begin{aligned} \perp \bullet M &= \perp \\ M \bullet \perp &= \perp. \end{aligned}$$

  Specifically, on head normal forms in D

$$\begin{aligned} &(\lambda x_1 \ldots x_m \,.\, ((z\ M_1) \ldots M_n)) \;\bullet\; (\lambda y_1 \ldots y_p \,.\, ((w\ N_1) \ldots N_q)) \\ =\;& (\lambda x_2 \ldots x_m \,.\, ((z\ M_1) \ldots M_n))\ [\ x_1 := (\lambda y_1 \ldots y_p \,.\, ((w\ N_1) \ldots N_q))\ ] \end{aligned}$$

  when $m \geq 1$. (η-reduction can result when $p = q = 0$ and $w = x_1$.) The resulting λ-term is a head normal form, even when z is x_1. When $m = 0$,

$$\begin{aligned} &((z\ M_1) \ldots M_n) \;\bullet\; (\lambda y_1 \ldots y_p \,.\, ((w\ N_1) \ldots N_q)) \\ =\;& (\ ((z\ M_1) \ldots M_n)\ (\lambda y_1 \ldots y_p \,.\, ((w\ N_1) \ldots N_q))\ ) \end{aligned}$$

  which again is a head normal form.

- C is head reduction on ordinary λ-terms, using elimination of λ-abstractions.* For example, we have

*The reduction shown above for factorials is not exactly what C would produce, since it used non-head reductions. However, this is only because we declined to show reduction of **times** and the other arithmetic combinators.

$$C(\ (((\lambda x\ .\ \lambda y\ .\ x)\ a)\ b)\ )\ \equiv\ ((\lambda y\ .\ a)\ b)$$
$$C(C(\ (((\lambda x\ .\ \lambda y\ .\ x)\ a)\ b)\ ))\ \equiv\ a$$
$$C(C(C(\ (((\lambda x\ .\ \lambda y\ .\ x)\ a)\ b)\ )))\ \equiv\ a$$

If M is any λ-term including $\perp$ as a subterm, we include the reduction

$$C(M)\ =\ \perp.$$

Given the facts above about operators corresponding to *Op*, since normal-order reduction selects the leftmost expression for reduction, C meets the restrictions on computation rules for continuous monotone partial order programs: C is the identity on D, C maps $Term(Op,D)$ to D, and C has a recursive definition on $B = Term(Op,A \cup D)$.

Summarizing, the lambda calculus can be expressed as a partial order program $P = <B,C,D,\sqsupseteq>$ where

B: partial λ-terms (α-equivalence classes)
C: head reduction
D: head normal forms (α-equivalence classes), and $\perp$
$\sqsupseteq$: lifting ($v \sqsupseteq \perp$, for all v in D).

6.3. Logic Programming

We pointed out earlier that logical implications are just inequalities over the truth lattice {**true**,**false**}, so the implication $H \leftarrow G$ expresses the constraint "*truth*(H)" $\geq$ "*truth*(G)". Intuitively, then, it is clear how logic programs might be expressed as partial order programs: all Horn rules are inequality constraints, and the rule $H \leftarrow G$ may be written meaningfully as $H \sqsupseteq G$.

We see first how this intuition is sufficient for *propositional* logic programs. In the propositional case the problem of evaluating a goal G amounts to finding a truth value for G. All rules in these programs can be cast in the form

$$u\ \leftarrow\ v$$

where u is a propositional variable in a finite set A, and v is either the value **true** or a disjunctive normal form expression using propositional variables in A.

Any propositional logic program π can be expressed easily as a partial order program. Define A as above, $Op = \{\wedge,\vee\}$, and:

B: and/or expressions over propositional variables in A
C: Horn rule reduction for π with a suitable 'evaluation rule'
D: {**true**,**false**}
$\sqsupseteq$: logical implication ($\leftarrow$) (**true** $\sqsupseteq$ **false**).

Here $B = Term(Op,A \cup D)$, for example, we can define a breadth-first reduction rule C as follows. Let C be the identity on D. For every u in A, define:

$$C(u) = \begin{cases} v & u \leftarrow v \text{ is the rule for } u \text{ in } \pi \\ \textbf{false} & \text{there is no rule for } u \text{ in } \pi \end{cases}$$

and for each and/or expression u in B, define:

$$C(u_1 \wedge u_2) = \begin{cases} \textbf{false} & u_1 = \textbf{false} \text{ or } u_2 = \textbf{false} \\ u_2 & u_1 = \textbf{true} \text{ and } u_2 \neq \textbf{false} \\ u_1 & u_2 = \textbf{true} \text{ and } u_1 \neq \textbf{false} \\ C(u_1) \wedge C(u_2) & \text{otherwise,} \end{cases}$$

$$C(u_1 \vee u_2) = \begin{cases} \textbf{true} & u_1 = \textbf{true} \text{ or } u_2 = \textbf{true} \\ u_2 & u_1 = \textbf{false} \text{ and } u_2 \neq \textbf{true} \\ u_1 & u_2 = \textbf{false} \text{ and } u_1 \neq \textbf{true} \\ C(u_1) \vee C(u_2) & \text{otherwise.} \end{cases}$$

This definition can be easily modified to implement other evaluation rules. Note that C operates here like the continuation semantic function **C** of denotational semantics, and like the fixed point computation rules of Manna [62]. Since C also meets the requirements for continuous monotone partial order programs, $P(\pi) = <Op,A,C,D,\sqsupseteq>$ is a continuous monotone partial order program with the same procedural semantics as π under the chosen evaluation rule.

This construction for propositional logic programs could be extended for predicate logic programs with logical variables. Basically the predicate logic case can be reduced to the propositional case by viewing a logic program as the collection of all 'ground instances' of its clauses [58]. Then if θ is a 'correct answer substitution' derivable for a logic program goal G under some computation rule, then the corresponding partial order program here assigns the goal $G\theta$ the value **true**. However, this construction is unlike logic programming in that the answer substitution must be provided in advance.

Instead, therefore, we use a direct translation of predicate logic programs to partial order programs. We extend the value domain {**true**,**false**} to a domain of *bindings*. Roughly speaking, we permit a logic program goal to take as a value an and-or expression of equations between logical variables and first-order terms. For example, with the logic program

$$p(a,b) \leftarrow$$
$$p(c,d) \leftarrow$$

the goal $p(X,Y)$ takes as its value the binding

$$(X = a \wedge Y = b) \vee (X = c \wedge Y = d)$$

while the goal $p(b,Z)$ takes the binding **false**.

Let us quickly introduce more terminology. A *first-order term* is a member of $Term(Op,V)$ where V is an infinite set of logical variables and Op is a set of function symbols. Each symbol f in Op has an associated *arity* giving its number of arguments.

The set A of all *first-order atomic formulas* consists of all terms of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity $n \geq 0$, and each argument t_i is a first-order term. A *binding formula* is an atomic formula $X = t$, where X is a logical variable and t is a first-order term. Binding formulas are atomic formulas with predicate symbol "=" of arity 2.

A *goal* is an and-or expression of atomic formulas. In other words, a goal is a member of $Term(\{\wedge,\vee\},A)$. A *binding* is a goal all of whose atomic formulas are binding formulas.

A *logic program* is a collection of *rules* of the form

$$p(X_1, \ldots, X_n) \leftarrow G$$

where $p(X_1, \ldots, X_n)$ is an atomic formula with (universally quantified) variables $X_1, \ldots, X_n$ and G is a goal. We assume without loss of generality that there is only one rule for each predicate symbol of the program, and that the arguments in the head of the rule are distinct variables. The collection of ($m \geq 1$) rules

$$p(t_{11}, \ldots, t_{1n}) \leftarrow G_1$$
$$\ldots$$
$$p(t_{m1}, \ldots, t_{mn}) \leftarrow G_m$$

is logically equivalent to the rule

$$p(X_1, \ldots, X_n) \leftarrow$$
$$(X_1 = t_{11} \wedge \ldots \wedge X_n = t_{1n} \wedge G_1) \vee \ldots \vee (X_1 = t_{m1} \wedge \ldots \wedge X_n = t_{mn} \wedge G_m)$$

provided we include the rule

$$X = X \leftarrow \textbf{true}.$$

Colmerauer's "equation systems" are conjunctions of binding formulas. In [26] an algorithm is given that rewrites equation systems to equivalent *reduced equation systems,* conjunctions of binding formulas in which

(1) the left hand sides of the formulas are distinct variables, and

(2) there is no *endless subsystem* of equations, a collection of formulas in which every left hand side also appears as a right hand side.

Reduced equation systems are always solvable in the algebra of *rational trees* (possibly infinite first-order terms whose set of subterms is finite). The algorithm can be extended for first-order unification with the occur check, corresponding to solvability over first-order terms, if this is desired.

Bindings here generalize Colmerauer's equation systems with disjunction. The generalization is only slight, since every binding is equivalent (via distribution of $\wedge$ over $\vee$) to a disjunction of equation systems.

The procedural semantics for logic programming can now be viewed accurately as a process of rewriting a goal to another goal that contains a reduced binding (reduced equation system) as a disjunct. The rewriting process combines the Prolog II reduction process for equations, presented in section 3 of [26], and goal elimination, the replacement of atomic subgoals by goals corresponding to the bodies of rules. See [58].

Rather than reproduce the reduction algorithm formally as a function C, we give an example that should illustrate the process clearly. With the convention above the standard 'append' predicate looks like

append(A,B,AB) $\leftarrow$
$(A = [\,] \wedge B = AB) \;\vee\; (A = [X \mid L] \wedge AB = [X \mid LB] \wedge \text{append}(L,B,LB))$.

The goal append(Y,Z,[a]) is reducible to a binding with the following sequence:

append(Y,Z,[a])
$(Y = [\,] \wedge Z = [a]) \vee$ $(Y = [X_1 \mid L_1] \wedge [a] = [X_1 \mid LB_1] \wedge \text{append}(L_1,Z,LB_1))$
$(Y = [\,] \wedge Z = [a]) \vee$ $(Y = [X_1 \mid L_1] \wedge X_1 = a \wedge LB_1 = [\,] \wedge \text{append}(L_1,Z,LB_1))$
$(Y = [\,] \wedge Z = [a]) \vee$ $(Y = [X_1 \mid L_1] \wedge X_1 = a \wedge LB_1 = [\,] \wedge$ $((L_1 = [\,] \wedge Z = [\,]) \vee$ $(L_1 = [X_2 \mid L_2] \wedge [\,] = [X_2 \mid LB_2] \wedge \text{append}(L_2,Z,LB_2))))$
$(Y = [\,] \wedge Z = [a]) \vee$ $(Y = [X_1 \mid L_1] \wedge X_1 = a \wedge LB_1 = [\,] \wedge$ $((L_1 = [\,] \wedge Z = [\,]) \vee$ $(L_1 = [X_2 \mid L_2] \wedge \textbf{false} \wedge \text{append}(L_2,Z,LB_2))))$
$(Y = [\,] \wedge Z = [a]) \vee$ $(Y = [X_1 \mid L_1] \wedge X_1 = a \wedge LB_1 = [\,] \wedge$ $((L_1 = [\,] \wedge Z = [\,]) \vee$ **false**))
$(Y = [\,] \wedge Z = [a]) \vee$ $(Y = [X_1 \mid L_1] \wedge X_1 = a \wedge LB_1 = [\,] \wedge L_1 = [\,] \wedge Z = [\,])$

To translate logic programming to partial order programming, we need the components $<B,C,D,\sqsupseteq>$ of a continuous monotone partial order program:

- B is the set of goals $Term(\{\wedge,\vee\},A)$. We impose on B the equivalence relation of logical equivalence on and-or expressions, given that all variables are existentially quantified.
- The computation rule C is the reduction process just illustrated. It is similar to the "surface deduction" process developed by Cox and Pietrzykowski formally in [27], extending Colmerauer's work. C meets the restrictions on computation rules of continuous monotone partial order programs since it always replaces subgoals by other goals that are either obtained from a constraint (rule) of the program, or

are equivalent over the algebra of rational trees (or first-order terms, as appropriate). The example here shows a sequence of leftmost reductions, but parallel reductions could have been used instead. In other words, the C reflected here is depth-first reduction, but breadth-first reduction is possible as well.

At first it might appear from this example that C is not a function, since it introduces different variables for different invocations of append. It *is* a function, however, and treats logical variables as having common names but different scopes determined by universal or existential quantifiers for the rule in which they appear, as in the lambda calculus. The reduction above uses subscripts to distinguish among occurrences of variables with different scope.

- The value domain D can be either definite or irreducible binding sets, depending on whether we need single solutions or all solutions. A *definite binding* is a goal with at least one disjunction that is a reduced equation system. An *irreducible binding* is a binding in which every disjunction is a reduced equation system. The drawback of insisting on irreducible bindings is that goals whose proof trees fail to terminate on one or more or-branches will be indefinite, i.e., fail to take on a value.
- Finally, we use limited form of *generality* as a partial ordering $\sqsupseteq$ on bindings. Specifically, every binding b satisfies $b \sqsupseteq$ **false**. Thus, $\wedge$ and $\vee$ are monotone operators on D. By *minimizing* a logic programming goal we are therefore reducing it as much as possible; coincidentally the binding that results is a disjunction of *most general* substitutions [58].

Summarizing, a logic program π is a partial order program with:

B: and/or goals (modulo equivalence of variables and and/or expressions)
C: goal elimination determined from π, and binding reduction
D: definite bindings (modulo equivalence)
$\sqsupseteq$: limited generality among bindings.

Here $\perp$ is **false**, the value assigned logic programs that fail.

It appears that a number of extensions of logic programming now being proposed can be represented with partial order programming by adapting the construction above. Specifically we can generalize it for Colmerauer's Prolog II with inequation constraints [25], Jaffar et al's Constraint Logic Programming with linear equality and inequality constraints [47,48], and Ait-Kaci and Nasr's system that generalizes first-order terms and unification to a distributive lattice of 'ψ-terms' and a meet operator on the lattice [2,3]. In each case we must generalize the bindings and function C given above to handle more expressive constraints.

6.4. Integrating Paradigms

Historically, programming systems have offered only one computational paradigm. Recently, however, a great deal of interest has been directed at developing systems integrating multiple paradigms, particularly functional and logic programming. The

book [29] contains many proposals for integrating logic, equations, and functions, and references the rapidly growing literature in this area. An excellent recent survey of existing alternative approaches appears in [4].

Partial order programming provides some perspective in surveying the large number of alternatives. Above it was shown that logic programming can be modeled by the scheme

B_{LP}: and/or expressions over atomic formulas
C_{LP}: Horn rule and unification reduction
D_{LP}: definite bindings
$\sqsupseteq_{LP}$: limited generality among bindings

and functional programming by the scheme

B_{FP}: partial λ-terms
C_{FP}: head reduction
D_{FP}: head normal forms
$\sqsupseteq_{FP}$: lifting.

One way for viewing integrated functional-logic programming systems is to study how they combine these two schemes. There are many combinations for integrating B_{LP} and D_{LP} with B_{FP} and D_{FP}. For example, using the nomenclature of [4], the *syntactic approach* integrates D_{LP} with B_{FP}, by permitting predicates to return arbitrary values. This amounts to the ''lattice programming'' mentioned earlier, and is really a syntactic sugaring of logic programming, since such programs can be translated directly into logic programs. By contrast the *algebraic approach* uses the equality predicate (and hence unification) as a point of integration of B_{LP} with B_{FP}. It augments Horn logic with a special equality predicate, and generalizes ordinary syntactic unification with ''E-unification'', which is typically implemented operationally via rewriting. This perspective is sharpened by formalizing these approaches as continuous monotone partial order programming schemes $<Term(Op,A \cup D),C,D,\sqsupseteq>$. Approaches differ widely in their treatment of the collected set of operators

$$Op \subseteq \{ \wedge \} \cup \{ \vee \} \cup \{ (\_\ \_) \} \cup \{ (\lambda x\ .\ \_) \mid x \in V \}$$

and the set of goal terms A.

Partial order programming suggests another new direction for integrating these paradigms. Both functional and logic programming rest on *elimination* mechanisms: λ-reduction and SLD-resolution are computation methods that repeatedly substitute one expression for another. They can be generalized as methods for solving inequalities by substituting one object by another that it dominates in some ordering. Previous attempts to integrate these paradigms centering around E-unification methods operate by ''substitution of equals for equals''. This slogan can be generalized powerfully to *''monotone substitution of lessers for greaters''*, or more precisely ''substitution of objects by their bounds, in monotone contexts''. If a function f is monotone in its first argument and $a \sqsupseteq b$, $f(a,c)$ is reducible to $f(b,c)$. This approach works whether f is a

user-defined function, or a monotone operator such as $\wedge$ in logic programming.

This generalization extends rewriting with the concepts of inequality and monotonicity. It permits integration such as combining the rewrite system of Hsiang [43] for Boolean algebras, and Traugott's nested resolution rule [91] or the replacement rules of Manna and Waldinger [64] mentioned earlier. It also permits very interesting treatment of negation, since negation operators can be treated as introducing "anti-monotone" (i.e., monotone decreasing) contexts, in which one can "substitute greaters for lessers" in a reduction process.

The subject of formalizing integrated paradigms with partial order programming is a large one and needs a much more thorough study than the brief discussion here. However, it would appear from this discussion that we are still only at the beginning of exploring ways to integrate functional and logic programming.

7. Conclusions

A great deal of work has been done recently on generalizing existing programming paradigms, particularly in connecting logic programming, functional programming, and object-oriented programming. Also, programming styles based on fixed point concepts have been proposed [22, 31, 75]. At the same time, researchers have made many proposals on computational frameworks for knowledge representation. Invariably, these proposals lack a foundation that is both general and formal.

This monograph has introduced the concept of partial order programming, and has illustrated the power of the paradigm with a sequence of examples. Chandy and Misra [22] point out, however, "the utility of a new approach is suspect, especially when it is a radical departure from the conventional." To conclude, then, we first summarize what has been presented, and then offer some perspectives on the potential of partial order programming.

7.1. Summary

Partial order programming is a new paradigm. It is general and formally defined. This work has studied the following progressively restricted classes of programs:

Partial order programs
Reductive partial order programs
Algebraic partial order programs
Continuous monotone partial order programs
Semilinear partial order programs.

The examples listed earlier show important problems in diverse fields that can be expressed naturally with the paradigm.

Partial order programming encompasses a number of novel concepts:

(1) A paradigm based on ordering can be useful in modeling complex systems. Ordering is basic to human knowledge representation. Structures can be modeled with partial orders with an effectiveness that is not possible with total orders.

(2) Many results in logic can be naturally and profitably viewed as properties of ordering and monotonicity.

(3) Generalizing mathematical programming by introducing partially-ordered value domains has real applications.

(4) Relaxation, an iterative constraint satisfaction technique, has a natural formalization in terms of monotone operators on partial orders.

(5) "Assignment refinement" semantics can be used as a replacement for single-assignment semantics. The assignment $G := d$ made in a reductive partial order program can be superseded by $G := d'$ provided that $d' \sqsupseteq d$. Assignment refinement is an alternative foundation for formalizing destructive assignment.

(6) Programming paradigms with a least fixed point semantics map straightforwardly into partial order programming.

(7) Logic programming and functional programming can be naturally expressed in partial order programming. The constraints of the partial order program reflect both the constraints of the original paradigms as well as their procedural semantics.

(8) Lattice programming is a way to generalize on logic programming, and on conventional inexact reasoning mechanisms.

(9) The common distinction between numerical and symbolic computing blurs if we view domains of values as partial orders having specified properties. When the domain is a semiring with an addition and multiplication operator, for example, certain partial order programs correspond to linear inequality systems and can be solved as such.

(10) Partial order programming permits exploitation of monotonicity. Monotonicity is important for a variety of reasons:

- Monotonicity is order preservation.
- Monotone operators have useful properties, notably that the composition of monotone operators is monotone.
- Monotonicity is the foundation for dynamic programming and Bellman's principle of optimality: when f is monotone, to maximize $f(x_1, \ldots, x_n)$ one must maximize $x_1, \ldots, x_n$.
- Continuous monotone constraints can be solved iteratively, with a unique minimal solution.
- Iterative solution of monotone constraints can be terminated early, providing approximate solutions.
- There is a strong connection between linearity and monotonicity. On certain ordered domains, monotone operators are linearizable.

7.2. Implementation

Earlier we presented the conjecture behind this work, that partial order programming will prove useful in modeling complex systems. It is beyond the scope of this monograph to investigate implementation of partial order programming, but for concreteness let us outline how the ideas covered in this monograph might be incorporated in an implementation.

Let us call a hypothetical partial order programming system based on the foregoing *MONOTONE*. Among other things, *MONOTONE* should provide different methods for solving systems of inequalities. In appropriate contexts it could use all of the methods discussed earlier: reduction with "monotone substitution of lessers for greaters", relaxation (assignment refinement), and elimination.

Up to this point all examples of partial order programs have used only a single partial order $\sqsupseteq$ and domain D. Recall however that the partial order can be a *sum* or *product* of n partial orders:

$$<D,\sqsupseteq> \quad = \quad <D_1,\sqsupseteq_1> + \ldots + <D_n,\sqsupseteq_n>.$$
$$<D,\sqsupseteq> \quad = \quad <D_1,\sqsupseteq_1> \times \ldots \times <D_n,\sqsupseteq_n>.$$

MONOTONE should provide an extensible *library* of partial orders, with for example:

$:\!-$	generality among logic program bindings
$\sqsupseteq$	generality among first-order terms
$\geq$	numeric ordering
$\supseteq$	set containment (Boolean algebra ordering)
isa	type inclusion
partof	part inclusion

More broadly, *MONOTONE* should incorporate a library of systems $<B,C,D,\sqsupseteq>$ that are useful. Domains D with operators Op having certain structure (e.g., monotonicity, monoid properties, etc.) should be declared and exploited in constraint solution.

Problems in *MONOTONE* can take the form

minimize(*Goal*,*Constraints*)

Minimization is performed relative to the partial order determined by the *Goal* object. This problem is in turn an object, whose value is the solution of the problem. In ways **minimize** is similar to the Prolog **call** metapredicate, but specifies the constraints to be used. Note that this may avoid some of the problematic issues surrounding meta-level inference in the logic programming framework [15, 16, 17]. Also, this construct introduces the possibility for *recursive minimization,* a feature that may be of interest for mathematical programming problems.

MONOTONE should include two basic components, then:

(1) A definitional component, for announcing that certain $<B,C,D,\sqsupseteq>$ systems exist, that operators are *monotone* (or anti-monotone) in certain arguments with respect to certain partial orders, that systems are *pointed continuous complete ordered semi-modules,* and so forth.

(2) An inequality handling component, defining how the minimization process works for each $<B,C,D,\sqsupseteq>$ systems, or combined systems. It is not necessary for these solution mechanisms to be programmed in *MONOTONE*, although this would be desirable.

The syntax of *MONOTONE* programs is an open matter. It is attractive to use something like Prolog syntax, because Prolog programs can elegantly *generate* constraint programs. For example, ordinary Prolog programs naturally generate equality constraint programs (bindings). These constraint programs can be solved either concurrently with their generation, or somewhat later. This concurrent (''on-line'', ''incremental'') approach is used both in Prolog II [25] and in *CLP*(**R**) [47, 48].

To see how inequality solution can be useful, consider an example: use of first-order term inequality ($\sqsupseteq$). Every Prolog clause

$$p(\, t_1 \,, \ldots, \, t_n \,) \;\leftarrow\; G$$

can be written

$$p(\, X_1 \,, \ldots, \, X_n \,) \;\leftarrow\; (X_1 \sqsupseteq t_1 \wedge t_1 \sqsupseteq X_1) \wedge \ldots \wedge (X_n \sqsupseteq t_n \wedge t_n \sqsupseteq X_n) \wedge G,$$

so a system that handles $\sqsupseteq$ constraints generated by expansions of clauses like this one is enough to interpret Prolog programs, and the term inequality primitive $\sqsupseteq$ can be quite useful. Among other things, it can be used for type inference and abstract interpretation as in the example given earlier. Also $\sqsupseteq$ can provide a simple type subsumption mechanism for knowledge representation systems as in LOGIN [3], and provide a kind of "one-way unification" for logic programs. One-way unification is useful in stream processing and functional computations in general.

MONOTONE would not be a trivial extension of Prolog. For example, it does not appear possible to implement $\sqsupseteq$ directly in Prolog, even on Prolog systems supporting Colmerauer's **dif** and **freeze** primitives [21, 25]. Although the evident implementation

```
X ⊒ Y  :-  dominates_currently(X,Y), dominates_henceforth(X,Y).
dominates_currently(X,Y)  :-  copy_term(X,T), unify(T,Y).
dominates_henceforth(X,Y)  :-  ...
```

forces X to be more general than Y at the point of invocation, it is not clear how the future generality constraint **dominates_henceforth** can be implemented. To see the difficulties here, notice that the Prolog goal

```
?- pair(X,Y) ⊒ pair(U,V),   X = Y.
```

should result in the unification U = V, but this will not be achieved without internal modifications to the unifier (goals frozen in the execution of $\sqsupseteq$ will not be executed when X = Y is executed), or extensions to the definitions of **dif** and **freeze**. Interestingly, however, an implementation is possible using the general **delay** primitive [21], which activates its delayed goal whenever the variable upon which the delay is done is bound to any value, variable or nonvariable.

MONOTONE is also different from Constraint Logic Programming in several respects. Primarily, *MONOTONE* emphasizes monotonicity and multiple partial orders, rather than constraint processing in general. *CLP*(**R**) currently covers an important class of constraints over a total order (linear equalities and equalities over the real numbers). Naturally, it is advantageous to handle many kinds of partial ordering constraints and thereby provide multiple "paradigms". Also, it would seem to that a modularization mechanism that distinguishes subproblems (like the minimization operator above) is useful in decomposing large constraint systems into smaller ones. Decomposition can be very important; practical use of existing elimination methods often requires constraints to be solved in modest batches.

This discussion is brief and abstract, but hopefully conveys what is possible here. The field of constraint programming languages is an exciting field, and we can expect rapid improvements in the next years. Because of the generality of partial order

programming, its implementation is an open area for research, and will continue to offer interesting challenges for some time.

7.3. Partial Order Programming Prospects

In closing here, it seems appropriate to state why we are optimistic about the future of partial order programming.

The past few years have seen an increasing emphasis on concurrent computation, declarative programming, and constraint-based formulations of problems. Some of these problems are not easily solvable, or solvable at all, using conventional procedures. Examples of difficult-to-treat problems are learning (inductive or otherwise), and adaptive behavior or control in changing environments. These problems involve, by their essence, the modeling of complex systems: adaptive behavior involves the application of a model of the environment, while learning involves the construction of models.

Current approaches to solving these problems tend to be informal. Suggestive terms such as spreading activation, connectionism, hill-climbing, and relaxation are used routinely. Although intuitively reasonable or correct, however, these terms are often left imprecise. As these problems grow in importance, and as parallel machines become more commonly available to support these solution approaches, a formal framework will become imperative.

It would seem that *any* formal setting for these problems will necessitate formal treatment of concepts such as iteration, convergence, monotonicity, and fixed points. These concepts are fundamental in constraint satisfaction and adaptive search for equilibria. Therefore, if partial order programming is not the right framework for these problems, it at least has the right basic foundations.

Of course, we believe partial order programming *is* the right framework for these problems. First, although a great deal of research is routinely performed on systems of equalities, ordering relationships and inequalities are more basic and, in many models, more natural. Humans are evidently better at reasoning about inequality than about equality, and partial order programming brings ordering to the fore.

Not by coincidence, partial orders make natural models of concurrency and "flow". Any sort of precedence or sequencing constraints induces a partial order, and recently interest has grown in partial order models of concurrency. As Pratt points out in [78], some concepts of concurrency are definable only for partial orders, the meaning of concurrency of two events in partial order models does not depend on the granularity of atomicity of the events, and in certain cases partial order models are easier to reason about than linear models.

Second, partial order programming is a natural setting for exploiting monotonicity, or order-preservation. George Dantzig has observed that while the world is not linear, linear programming is still useful. In the same way, the world is not monotone, yet

many worldly problems can be abstracted to continuous monotone partial order programming problems. Korf's macro operators [54] illustrate how abstractions of non-monotone phenomena can be monotone, and Pearl [76] has pointed out how many heuristics for solving problems use methods for solving approximate versions of the problems. Because of the relationship between monotonicity and iterative fixed point solution of constraints, monotone models are able to ignore detail and tolerate inaccuracy, remaining simple yet robust.

Finally, partial order programming offers perspective. It is difficult to get perspective these days, since current field boundaries artificially separate notions such as logic and ordering, symbolic and numeric computing, computer programming languages and mathematical programming, solution by elimination and solution by relaxation, to name a few. A century ago, as early as 1880, the mathematician-semioticist C.S. Peirce generalized upon Boole's algebraic system of logic for the laws of thought with a theory of ordered sets and, ultimately, what we now call lattices. Sadly, these ideas of Peirce have not attained the eminence they should have, and today we find ourselves with formal logics that are unable to deal effectively with various important problems (such as learning and adaptive behavior or control). Partial order programming is a platform for cutting across field boundaries, using Peirce's basic tool of ordering.

Acknowledgement

The author is grateful to Paul Eggert for many great ideas and suggestions that have helped shape this work. Alan Stoughton patiently and carefully explained concepts in denotational semantics, and provided a number of improvements in the definition of partial order programming. Hassan Ait-Kaci, Howard Blair, Arman Bostani, Kam Chow, Rina Dechter, Mel Fitting, Forouzan Golshani, Mike Gorlick, Shen-Tzay Huang, Richard Huntsinger, Dean Jacobs, Carl Kesselman, Rich Korf, Koen Lecot, Bob Meyer, Yiannis Moschovakis, Dick Muntz, Roger Nasr, Sanjai Narain, Steve Russell, Matt Stillerman, Loring Tu, Judi Uttal, Victor Vianu, Scott Wilson, the UCLA Logic-based KR group, and many others provided greatly appreciated comments and suggestions that are reflected here.

Appendix I: Proofs of Fixed Point Theorems

Theorem 1

Let $f : D \to D$ be a continuous monotone map on the pointed cpo $<D,\sqsupseteq>$. Then f has a least fixed point equal to

$$\mathbf{fix}\, f \;=\; \bigsqcup \{\, f^k(\bot) \mid k \in \omega \,\}$$

where f^k is f iterated k times.*

Proof

We claim that, for every $k \geq 0$,

$$f^{k+1}(\bot) \;\sqsupseteq\; f^k(\bot).$$

This can be shown by induction: $f(\bot) \sqsupseteq \bot$ establishes k=0, and monotonicity of f gives the induction step. As a result,

$$S \;=\; \{\, \bot,\; f(\bot),\; f^2(\bot),\; \cdots \,\}$$

forms a chain, and

$$f(\,\mathbf{fix}\, f\,) \;=\; f(\,\bigsqcup S\,) \;=\; \bigsqcup f(S) \;=\; \bigsqcup \{\, f(f^k(\bot)) \mid k \in \omega \,\} \;=\; \mathbf{fix}\, f$$

so $\mathbf{fix}\, f$ is a fixed point of f.

$\mathbf{fix}\, f$ is also a least fixed point. For any other fixed point z we must have $z \sqsupseteq f^k(\bot)$ for all $k \geq 1$. To see this, note that $z \sqsupseteq \bot$, and applying monotonicity of f gives $f(z) \sqsupseteq f(\bot)$. Since $z = f(z)$, it follows inductively that $z \sqsupseteq \mathbf{fix}\, f$. □

Theorem 2

Let $f : D \to D$ be a monotone function on the complete lattice $<D,\sqsupseteq>$. Then f has a fixed point in D.

Proof

Let S be the set of all x in D such that $f(x) \sqsupseteq x$. Now let $z = \bigsqcup_{x \in S} x$. Since f is monotone, and $z \sqsupseteq x$ for every x in S, it follows that $f(z) \sqsupseteq f(x) \sqsupseteq x$ for every x in S. So $f(z)$ is an upper bound for S, so z must be a member of S, since then $f(z) \sqsupseteq z$. Applying monotonicity to this inequality, we get $f(f(z)) \sqsupseteq f(z)$, so $f(z)$ is in S as well. But $z \sqsupseteq x$ for every x in S, so $z \sqsupseteq f(z)$. Thus $f(z) = z$, and f has a fixed point. □

*Recall ω is the set of natural numbers.

References

1. Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.
2. Ait-Kaci, H., "A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures," Ph.D. dissertation, University of Pennsylvania, Dept. of Computer and Information Science, Philadelphia, PA, 1984.
3. Ait-Kaci, H. and R. Nasr, "LOGIN: A Logic Programming Language with Built-in Inheritance," *Journal of Logic Programming*, vol. 3, no. 3, pp. 185-215, October 1986.
4. Ait-Kaci, H. and R. Nasr, "Residuation: A Paradigm for Integrating Logic and Functional Programming," Technical Report AI-359-86, MCC, October 1986.
5. Ait-Kaci, H., P. Lincoln, and R. Nasr, "Le Fun: Logic, equations, and Functions," *Proc. Symp. on Logic Programming*, pp. 17-23, IEEE Computer Society #799, September 1987.
6. Allgower, E. and K. Georg, "Simplicial and continuation methods for approximating fixed points and solutions to systems of equations," *SIAM Review*, vol. 22, pp. 28-85, 1980.
7. Allgower, E. and K. Georg, "Predictor-Corrector and Simplicial Methods for Approximating Fixed Points and Zero Points of Nonlinear Mappings," in *Mathematical Programming: The State of the Art*, ed. A. Bachem, M. Groetschel, B. Korte, pp. 15-56, Springer-Verlag, New York, 1983.
8. Apt, K.R. and M.H. van Emden, "Contributions to the Theory of Logic Programming," *Journal of the ACM*, vol. 29, no. 3, pp. 841-862, July 1982.
9. Aspvall, B., "Efficient Algorithms for Certain Satisfiability and Linear Programming Problems," Report No. STAN-CS-80-822 (Ph.D. thesis), Stanford University, Stanford, CA, September 1980.
10. Attardi, G. and M. Simi, "A Description-Oriented Logic for Building Knowledge Bases," *Proceedings IEEE*, vol. 74, no. 10, pp. 1335-1344, October 1986.
11. Bachmair, L. and N. Dershowitz, "Inference Rules for Rewrite-Based First-Order Theorem Proving," *Proc. 2nd Symp. on Logic in Computer Science*, pp. 331-337, IEEE Computer Society #793, June 1987.
12. Barendregt, H.P., *The Lambda Calculus,* North-Holland, New York, 1984.
13. Bartley, W.W., *Lewis Carroll's Symbolic Logic,* Clarkson N. Potter, Inc., New York, 1986.
14. Bernstein, P.A. and D-M.W. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, vol. 28, no. 1, pp. 25-40, January 1981.
15. Bowen, K.A. and R.A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming," in *Logic Programming*, ed. K. Clark, S.-A. Tarnlund, pp. 153-172, Academic Press, New York, 1982.
16. Bowen, K.A. and T. Weinberg, "A Meta-Level Extension of Prolog," *Proc. Symposium on Logic Programming*, pp. 48-53, IEEE Computer Society #636, Boston, 1985.

17. Bowen, K.A., "Meta-Level Programming and Knowledge Representation," *New Generation Computing*, vol. 4, 1986.

18. Bruynooghe, M., G. Janssens, A. Callebaut, and B. Demoen, "Abstract Interpretation: Towards the Global Optimization of Prolog Programs," *Proc. Symp. on Logic Programming*, pp. 192-204, IEEE Computer Society #799, San Francisco, September 1987.

19. Bundy, A., *The Computer Modeling of Mathematical Reasoning,* Academic Press, New York, 1983.

20. Carlson, B.C., "Algorithms Involving Arithmetic and Geometric Means," *American Math Monthly*, vol. 78, no. 5, pp. 496-505, May 1971.

21. Carlsson, M., "Freeze, Indexing, and Other Implementation Issues in the WAM," *Proc. 4th Intnl. Conf. on Logic Programming*, pp. 40-58, MIT Press, Melbourne, Australia, May 1987.

22. Chandy, K.M. and J. Misra, "Parallel Program Design: A Foundation," book preprint, University of Texas, February 1987. To be published by Addison-Wesley, 1987.

23. Chou, C.-T., "Relaxation Processes: Theory, Case Studies and Applications," Report CSD-860057 (M.S. Thesis), UCLA Computer Science Dept., Los Angeles, CA, February 1986.

24. Clancey, W.J., "Classification Problem Solving," Technical Report STAN-CS-84-1018, Department of Computer Science, Stanford University, 1984.

25. Colmerauer, A., H. Kanoui, and M. van Caneghem, "Prolog, theoretical principles and current trends," *Technology and Science of Informatics*, vol. 2, no. 4, pp. 255-292, 1983.

26. Colmerauer, A., "Equations and Inequations on Finite and Infinite Trees," *Proc. Intnl. Conf. on Fifth Generation Computer Systems (FGCS'84)*, pp. 85-99, North-Holland, Tokyo, November 1984.

27. Cox, P.T. and T. Pietrzykowski, "Surface Deduction: a uniform mechanism for logic programming," *Proc. Symposium on Logic Programming*, pp. 220-227, IEEE Computer Society #636, Boston, 1985.

28. Davis, M., *Computability and Unsolvability,* Dover Publications, Inc., New York, 1982. Appendix 2.

29. DeGroot, D. and G. Lindstrom, *Logic Programming: Functions, Relations, and Equations,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

30. Despain, A.M., "Fourier Transform Computers Using CORDIC Iterations," *IEE Trans. Comput.*, vol. C-23, no. 10, pp. 993-1001, October 1974.

31. Dijkstra, E.W. and C.S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1-4, 29 August 1980.

32. Dodgson, C.L., *Solution,* B:=b, C:=n, D:=n, L:=m, M:=s (Barry–both, Cole–neither, Dix–neither, Lang–mustard only, Mill–salt only), November 2, 1896.

33. Eggert, P.R. and D.V. Schorre, "Logic Enhancement: A Method for Extending Logic Programming Languages," *Proc. 1982 Symp. on LISP and Functional*

Programming, pp. 74-80, Pittsburgh, PA, August 1982.

34. Emden, M.H. van, "Quantitative Deduction and Its Fixpoint Theory," *Journal of Logic Programming*, vol. 3, no. 1, pp. 37-53, April 1986.
35. Fitting, M., "A Deterministic Prolog Fixpoint Semantics," *J. Logic Programming*, vol. 2, no. 2, pp. 111-118, July 1986.
36. Fitting, M., "Logic Programming on a Topological Bilattice," Technical Report, Dept. of Computer Science, CUNY, New York 10036, 1987.
37. Fletcher, J.G., "A More General Algorithm for Computing Closed Semiring Costs Between Vertices of a Directed Graph," *Comm. ACM*, vol. 23, no. 6, pp. 350-351, June 1980.
38. Fraenkel, A.A., *Abstract Set Theory, 1st Ed.*, p. 172, North-Holland, Amsterdam, 1953.
39. Gnesi, S., U. Montanari, and A. Martelli, "Dynamic Programming as Graph Searching: An Algebraic Approach," *Journal of the ACM*, vol. 28, no. 4, pp. 737-751, October 1981.
40. Haralick, R.M. and L.G. Shapiro, "The Consistent Labeling Problem: Part I," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-1, no. 2, pp. 173-184, April 1979.
41. Haralick, R.M. and L.G. Shapiro, "The Consistent Labeling Problem: Part II," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-2, no. 3, pp. 193-203, May 1980.
42. Hitchcock, P. and D. Park, "Induction Rules and Termination Proofs," in *Automata, Languages, and Programming*, ed. M. Nivat, pp. 225-251, North-Holland, 1973.
43. Hsiang, J. and N. Dershowitz, "Rewrite Methods for Clausal and Non-Clausal Theorem Proving," in *Proc. 10th ICALP*, ed. J. Diaz, pp. 331-346, Springer-Verlag, LNCS #154, Barcelona, 1983.
44. Huet, G., "Deduction and Computation," in *Fundamentals of Artificial Intelligence*, ed. W. Bibel, Ph. Jorrand, pp. 39-74, Springer-Verlag, LNCS #232, New York, 1986.
45. Isaacson, E. and H.B. Keller, *Analysis of Numerical Methods*, J. Wiley & Sons, New York, 1966. (Chapter 9, Section 2: Solution of Laplace Difference Equations.)
46. Jacobson, N., *Basic Algebra I*, W.H. Freeman, 1974.
47. Jaffar, J. and J-L. Lassez, "Constraint Logic Programming," *Proc 12th ACM Symposium on Principles of Programming Languages*, January 1987.
48. Jaffar, J. and S. Michaylov, "Methodology and Implementation of a CLP System," *Proc. 4th Intnl. Conf. on Logic Programming*, pp. 196-218, MIT Press, Melbourne, Australia, May 1987.
49. Kaeufl, T., "Program Verifier 'Tatzelwurm': Reasoning abut Systems of Linear Inequalities," in *Proc. 8th CADE*, ed. J.H. Siekmann, pp. 300-305, Springer-Verlag, LNCS #230, Oxford, 1986.

50. Keenan, E.L. and L.M. Faltz, *Boolean Semantics for Natural Language,* D. Reidel Publishing Co., Boston, 1985.

51. Keeney, R.L. and H. Raiffa, *Decision with Multiple Objectives: Preferences and Value Tradeoffs,* John Wiley & Sons, New York, 1976.

52. Kleene, S.C., *Introduction to Metamathematics,* Van Nostrand, New York, 1952. Section 66, Theorem XXVI.

53. Kneale, W. and M. Kneale, *The Development of Logic,* Clarendon Press, Oxford, 1984.

54. Korf, R.E., *Learning to Solve Problems by Searching for Macro-Operators,* Morgan Kaufmann Publishers, Palo Alto, CA, 1985.

55. Lassez, C., K. McAloon, and R. Yap, "Constraint Logic Programming and Option Trading," *IEEE Expert*, vol. 2, no. 3, pp. 42-50, Fall 1987.

56. LeChenadec, P., *Canonical Forms in Finitely Presented Algebras,* J. Wiley & Sons, Inc., New York, 1986.

57. Lehmann, D.J., "Algebraic Structures for Transitive Closure," *Theoretical Computer Science*, vol. 4, pp. 59-76, 1977.

58. Lloyd, J., *Foundations of Logic Programming,* Springer-Verlag, New York, 1984.

59. Mackworth, A.K. and E.C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence*, vol. 25, pp. 65-74, 1985.

60. Manna, Z., S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs," *Comm. ACM*, vol. 16, no. 8, pp. 491-502, August 1973.

61. Manna, Z., *Mathematical Theory of Computation,* McGraw-Hill, New York, 1974. Chapter 5: The Fixed-Point Theory of Programs.

62. Manna, Z., *Mathematical Theory of Computation,* McGraw-Hill, New York, 1974.

63. Manna, Z. and A. Shamir, "The Theoretical Aspects of the Optimal Fixedpoint," *SIAM J. Comput.*, vol. 5, no. 3, pp. 414-426, 1976.

64. Manna, Z. and R. Waldinger, "Special Relations in Automated Deduction," *Journal of the ACM*, vol. 33, no. 1, pp. 1-59, January 1986.

65. Mellish, C.S., *Computer Interpretation of Natural Language Descriptions,* John Wiley & Sons (Ellis Horwood Ltd.), New York, 1985.

66. Mellish, C.S., "Abstract Interpretation of Prolog Programs," *Proc. 3rd Intnl. Conf. on Logic Programming*, pp. 463-474, Springer-Verlag LNCS 225, London, July 1986.

67. Minsky, M., *The Society of Mind,* Simon and Schuster, New York, 1986.

68. Montanari, U., "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences*, vol. 7, pp. 95-132, 1974.

69. Mosses, P.D. and G.D. Plotkin, "On Proving Limiting Completeness," *SIAM J. Comput.*, vol. 16, no. 1, pp. 179-194, February 1987.

70. Nadel, B.A., "The General Consistent Labeling (or Constraint Satisfaction) Problem," Technical Report DCS-TR-170, Dept. of Computer Science, Rutgers University, New Brunswick, NJ 08903, January 1986.

71. Narain, S., "A Technique for Doing Lazy Evaluation in Logic," *J. Logic Programming*, vol. 3, no. 3, pp. 259-276, October 1986.

72. Nelson, G., "The generalized limit theorem," Note CGN46, in errata to Report SRC-16, DEC Systems Research Center, 130 Lytton Ave, Palo Alto CA 94301, June 1987.

73. O'Keefe, R.A., "Finite Fixed-Point Problems," *Proc. 4th Intnl. Conf. on Logic Programming*, pp. 729-743, MIT Press, Melbourne, Australia, May 1987.

74. Papadimitriou, C.H. and K.S. Stieglitz, *Combinatorial Optimization,* Prentice-Hall, Englewood Cliffs, NJ, 1982.

75. Parnas, D.L., "A Generalized Control Structure and Its Formal Definition," *Comm. ACM*, vol. 26, no. 8, pp. 572-581, August 1983.

76. Pearl, J., *Heuristics,* Addison-Wesley, Reading, MA, 1984.

77. Peleg, S., "A New Probabilistic Relaxation Scheme," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-2, no. 4, pp. 362-369, July 1980.

78. Pratt, V., "Modelling Concurrency with Partial Orders," *International J. Parallel Programming*, vol. 15, no. 1, 1986. Also Stanford Tech. Report STAN-CS-86-1113, June 1986.

79. Rosenfeld, A., R.A. Hummel, and S.W. Zucker, "Scene Labelling by Relaxation Operators," *IEEE Trans. Systems, Man, and Cybernetics*, vol. SMC-6, no. 6, pp. 420-433, June 1976.

80. Saint-Dizier, P., "On Syntax and Semantics of Adjective Phrases in Logic Programming," Rapport de Recherche 381, INRIA, B.P. 105, 78153 Le Chesnay Cedex, FRANCE, March 1985.

81. Scarf, H.E., "Fixed-Point Theorems and Economic Analysis," *American Scientist*, vol. 71, no. 3, pp. 289-296, May-June 1983.

82. Schmidt, D.A., *Denotational Semantics,* Allyn and Bacon, Inc., Boston, 1986.

83. Shostak, R.E., "On the SUP-INF Method for Proving Presburger formulae," *Journal of the ACM*, vol. 24, no. 4, pp. 529-543, October 1977.

84. Shostak, R.E., "Deciding Linear Inequalities by Computing Loop Residues," *Journal of the ACM*, vol. 28, no. 4, pp. 769-779, October 1981.

85. Smart, D.R., *Fixed Point Theorems,* Cambridge University Press, New York, 1980.

86. Southwell, R.V., *Relaxation Methods in Engineering Science,* Oxford U. Press, 1940.

87. Stoy, J., *Denotational Semantics: The Scott-Strachey Approach To Programming Language Theory,* MIT Press, Cambridge, MA, 1977.

88. Stoy, J., "Some Mathematical Aspects of Functional Programming," in *Functional Programming and its Applications*, ed. J. Darlington, P. Henderson, D.A. Turner, Cambridge University Press, New York, 1982.

89. Tarjan, R.E., "A Unified Approach to Path Problems," *Journal of the ACM*, vol. 28, pp. 577-593, 1981.

90. Tarski, A., "A Lattice-Theoretical Fixpoint Theorem and its Applications," *Pacific J. Math*, vol. 5, pp. 285-309, 1955.

91. Traugott, J., "Nested Resolution," in *Proc. 8th CADE*, ed. J.H. Siekmann, pp. 394-402, Springer-Verlag, LNCS #230, Oxford, 1986.

92. Wadsworth, C.P., "The Relation Between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-Calculus," *SIAM J. Comput.*, vol. 5, no. 3, pp. 488-521, September 1976.

93. Wadsworth, C.P., "Approximate Reduction and Lambda Calculus Models," *SIAM J. Comput.*, vol. 7, no. 3, pp. 337-356, August 1978.

94. Waltz, D., "Generating Semantic Descriptions from Drawings of Scenes with Shadows," in *The Psychology of Computer Vision*, ed. P. Winston, pp. 19-92, McGraw-Hill, New York, 1975.

95. Zimmermann, U., *Linear and Combinatorial Optimization in Ordered Algebraic Structures,* North-Holland, New York, 1981. Annals of Discrete Mathematics, vol. 10.