Order Number 9009770

A semantics for parallel logic programs

Piccarello, James, Ph.D.

City University of New York, 1989

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

A SEMANTICS FOR PARALLEL LOGIC PROGRAMS

by

James Piccarello

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1989

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the disseration requirement for the degree of Doctor of Philosophy.

9/28/84
Date

Chair of Examining Committee

9/28/84
Date

Executive Officer

Prof. Michael Anshel

Prof. Frank S. Beckman

Prof. Melvin Fitting

Prof. Stathis Zachos

Supervisory Committee

The City University of New York

Abstract

A Semantics for Parallel Logic Programs

by

James Piccarello

Adviser: Professor Eralp Akkoyunlu

Denotational semantics for logic programs have been developed by Apt, Van Emden, Kowalski, Fitting, Lassez, and Maher. These efforts provide a precise definition of what relations are computed by logic programs. They do not address in detail how these results are computed. This thesis provides such a semantics using the CSP algebra of processes, developed by Hoare.

A class of CSP search tree processes is investigated. Such processes may be used to represent all possible solution sequences. The effect of a control component can be seen as constraining such a process. This allows a seperate specification of the control component of a logic program. This control component is itself specified by a Horn clause logic program. This is done by extending CSP to include Horn clause guards. a formal semantics using techniques developed by Fitting. Thus, a parallel logic program may be specified with two sets of Horn clauses, one describing the manipulations of user data and the other describing the search process to be used.

Acknowledgements

If there is one thing that is universally true of doctoral research, then it is that there are many people who cannot be adequately thanked. The proper care and feeding of a doctoral candidate ranges from technical assistance to bureaucratic favors to downright coddling.

Eralp Akkoyunlu served as chief technical adviser and (most importantly) chief prodder, ensuring there was always enough pressure to overcome inertia. He also did the unthinkable - he left a sunny Carribean vacation to preside over a doctoral defense. If that's not dedication nothing is!

The members of my examining committee were also generous with their time and advice. I could always rely on Stathis Zachos for both his mathematical ability and Mediterranean good cheer. He provided many helpful suggestions. Melvin Fitting must be thanked not only for always being ready to resolve many of my mental knots about logic, but also for developing many ideas about the semantics of logic programs that are essential to the paths I've taken here. Frank Beckmann must be thanked for the administrative wizardry he displayed in helping me overcome a number of unspeakable bureaucratic hurdles. Finally, every student should be fortunate enough to have Mike Anshel as an adviser. I must thank him for much good advice. Life would have been easier if I followed *all* of it. In addition, he introduced me to the elegance of theoretical computer science and formal languages.

To these I should also add Athanasios Glavas, Steve Lucci, and Robert Nirenberg -

fellow students whose conversations were always welcome. I thank Valentin Turchin for introducing me to the pleasures of research and Russian food. I thank Karel Hrabcek for introducing me to denotational semantics. It was in his class that I first saw the power and elegance of Scott's theory of domains.

But Man, and yes, even doctoral students, do not live by research alone. George Ross gave me the oppurtunity to earn my daily bread at CCNY while I did the research for this thesis. AT&T Bell Labs did the same while I did the writing. If nothing else, you'll see within these pages the wonders of *troff* and *eqn*.

But I haven't lived by bread and research alone. My family has provided the most important ingredients. I thank my parents for all those things everyone needs to thank their parents for. In addition, I can't imagine a more generous or efficient funding agency! Caitlin and Matthew never really did obey the orders to not disturb daddy when he worked. But those interruptions were usually the best part of the day. Thanks for *not* leaving me alone!

And lastly, I wasn't kidding about that coddling stuff! This task fell on the shoulders of one who didn't know *for better or for worse* included doctoral studies. Mary, I couldn't have done it without you! You made sure I left a doctoral student but came back a human being. Now it's time for *better*.

CONTENTS

1. Introduction 1

   1.1 Motivation 1

   1.2 Goals 5

   1.3 Outline 7

2. Domains 9

   2.1 Introduction 9

   2.2 Mathematical Definitions 9

   2.3 Why Domains? 15

   2.4 Domains: Formal Definition 18

   2.5 Domain Operations 20

   2.6 Least Fixed Points and Induction 24

   2.7 Examples of Domains 25

3. Processes 33

   3.1 Informal Theory of Processes 33

   3.2 A Formal Theory of Processes 40

   3.3 A Basis for the Domain of Processes 42

   3.4 Operations on Processes 46

   3.5 Reasoning about CSP 57

4. Logic Programs 59

   4.1 Horn Clause Logic Programs 59

   4.2 The Control of Logic Programs 66

4.3 Metalevel Inference 74

4.4 Domains for Logic Programs 79

5. The Semantics of Control 92

5.1 Introduction 92

5.2 Reductions 95

5.3 Trees 108

5.4 Operations on Trees 111

5.5 Mapping Confluent Reductions to Processes 123

6. Rules and Logic Programs 134

6.1 Reductions for Logic Programs 134

6.2 Continuations for Logic Processes 138

6.3 The SLD Relation 145

6.4 Specifying Control 149

7. Appendix 1 - Proofs 155

7.1 Proofs for Chapter 3 155

7.2 Proofs for Chapter 4 161

8. References 166

A Semantics for Parallel Logic Programs

James Piccarello

1. Introduction

1.1 Motivation

There is currently a great deal of interest in the possible sorts of computers and languages that will prove most useful in the near future. Prolog has gotten its share of the attention. Publications as diverse as *AI Expert* and *BYTE* have devoted special issues to Prolog.

In [GG85], Genesereth and Ginsberg made the bold statement:

> *Although we do not expect that logic programming will completely supplant traditional software engineering, its advantages and range of applicability suggest that it may become the dominant programming methodology in the next century.*

Why all the interest? A major task of programmers is to represent objects in a

user domain by the data types of an implementation domain. At first, machine locations were the only data objects available to the programmer. This left a large gap between the two domains. The development of assemblers, which introduced numbers and variables, shortened the gap by providing more sophisticated objects. Data types also involve *operations* on objects and the emergence of languages like Fortran, which allow arithmetic expressions, add more complex operations. The addition of more complex control structures and data types, such as files and records, are part of an evolution which brings the implementation domain closer to the user domain.

In addition to this bottom-up evolution there is also a top-down one. Here the problem is to describe the widest range of user domains, in some more or less abstract form, and then to work towards a machine implementation. It differs from the bottom-up approach in that rather than attack problems singly, a general theory of user domains is sought that will allow a clear operational solution.

There have been a number of approaches to this problem. One promising and elegant one has been to represent objects in the user domain by some universal data structure, such as terms, property lists, or frames, which record essential assertions about each object. The various relations that hold among such objects, as well as concepts that involve such objects, can be expressed by predicates. Finally, the predicates themselves may be characterized by axioms and since operations are special cases of relations, these predicates can also be used to characterize operations on the user's domain.

I shall refer to the assertion that symbolic logic and its associated metatheory can

form a useful, comprehensive, and efficient basis for representing user domains as the *Logic Programming Hypothesis*, or *LPH*.

The usefulness of symbolic logic has been demonstrated by its emergence, in the twentieth century, as a fundamental tool of mathematicians and philosophers. Determining whether computer users will find it an appropriate formalism is unclear at present because of the relative novelty of using logic for this purpose and because the uses to which computers are being put is changing so rapidly. Moreover, the inherent nondeterminism of logical derivations raises doubts regarding the possibility of efficient implementations.

However, there are reasons to predict that the LPH will eventually be justified.

- The theoretical equivalence of provability and computability is well known.
- The emergence of relational data bases, expert systems, and artificial intelligence as significant technologies support the usefulness of logic as a computational formalism. [CODD70, MCC83, SHA83b, STE85]
- Scientists such as Hoare, Gries and Dijkstra have shown that logic is extremely useful, perhaps even essential, in understanding traditional procedural languages. [DIJ75, DIJ76, GR81, HO69]
- The emergence of Prolog as a general purpose language suggests the broad applicability of logic programming. [CM81, CUCU85]
- The efficiency of Prolog is comparable to that of LISP.
- The predicted hardware innovations resulting from VLSI makes possible

specialized and efficient logic machines.

How does logic programming languages compare with traditional procedural languages? If functional programming is programming solely with functions, then what should procedural programming be, forgetting for a moment that it already has a well-established meaning? It should be programming only with procedures. In such a language the only kind of statement would then be a procedure call. All control would involve the sequencing of procedure invocations and all manipulation of data would be through parameter passing. In this sense one may say that logic programming languages are quintessentially procedural - far more so than even ALGOL.

It is just this complete reliance on procedure invocation that gives programming in PROLOG its distinct flavor. In a language like PASCAL, parameter passing is accomplished by a relatively simple mechanism. No work is done getting into a procedure, while a great deal of work is done while in the procedure. It is just the opposite with PROLOG. Unification can be very complex, especially in some of its more exotic variations involving built-in operators, while the only work done inside of a PROLOG procedure is to call other procedures. In a sense most of the work is done between PROLOG procedures rather than in them. This makes it a good choice for consideration as a language to program multi-processor systems in which many processors would perform relatively simple tasks which taken together yield a complex result. Unification provides a more complex and flexible communication mechanism than traditional parameter passing mechanisms.

1.2 Goals

A good part of this interest involves logic programs executed on parallel processors. See [ES85, GG85, HOG85, KO85, LL88, MCC83, SHA83a, SHA83b, STE85]. The research reported here concerns the theory of such systems. In particular, a formal semantics for such systems is presented.

When dealing with the semantics of a class of languages it is a typical strategy to first construct a semantics for a single metalanguage capable of expressing all the necessary semantic ideas. Then members of that class of language can be given a semantics by translating them into this "intermediate metacode". The advantage is that the mathematical work is done only once. This has been done for sequential languages, by Scott and others, and for concurrent languages by Hoare, Milner and others.

The goal of denotational semantics is to allow the mapping of a program *P* in some language language **L** to follow the "natural" syntax of **L**. The syntax of **L** defines *P* in terms of its immediate constituents. To be "natural", one wants the mapping of *P* to be equivalent to some function of the mapping of its immediate constituents. Moreover, this should also apply recursively to the immediate constituents themselves. Those syntactic categories that have no constituents would have a mapping defined directly.

With respect to logic programs, we intend to interpret Kowalski's equation:

$$Algorithm = Logic + Control$$

as a syntactic equation for logic programs. That is, if we let *LP* stand for logic program, then the syntax of a logic program would have the following BNF production:

$$LP ::= Logic + Control$$

The meaning of a logic program can then be given a by the following equation:

$$f(LP) = g(\nu(Logic), \kappa(Control))$$

where each of f, g, ν, and κ maps from a syntactic category to a semantic domain. ν give the declarative meaning, specified by the *Logic* component, while κ gives the procedural component. The function g combines these to give f, the meaning of the logic program itself.

It should also be noted that this idea can be extended to logic programming languages which do not globally separate logic from control. This covers virtually all current implementations of PROLOG, including PARLOG, Concurrent Prolog, as well as proposed extensions. The idea is to change the definition of ν and κ to each take *LP* as a parameter. ν will ignore procedural aspects of programs while κ ignores declarative aspects. This can be considered as a formal explication of the notions of a procedural versus a declarative reading. We thus have:

$$f(LP) = g(\nu(LP), \kappa(LP))$$

To characterize his own work, J.S. Conery presents the following table:

Levels of Abstraction	
Theory	A model of computation and semantics of programs in the model
Operation	Abstract interpreter; rules for performing steps of a computation and ordering steps
Implementation	Concrete interpreter specifying representation of programs and data
Machine	Low level implementation; hardware components, interconnection, distribution

The place of the research described here is the top row.

1.3 Outline

In Chapter 2 the elements of domain theory using *complete partial orders* is reviewed and in Chapter 3 the elements of process theory, as characterized by Hoare in [BHR84], is reviewed. This theory of processes is itself based on the theory of domains.

In Chapter 4 the main ideas of logic programs over the Herbrand Universe are reviewed. The control mechanism of a particular logic programming system is a key feature which distinguishes it from other logic programming systems. In this chapter, we briefly review some of the mechanisms that have been proposed to control logic programs executing in parallel. We then consider how such systems can be described by logic programs themselves. Finally, we consider a semantics for logic programs based on domain theory. This theory is due essentially to Prof. Melvin Fitting.

The semantics for logic programs described in Chapter 4 focuses on the declarative meaning of a logic program. In Chapter 5 we consider domains that are suitable for

focusing on the procedural semantics of logic programs. For this, we view a logic program as a computation on search trees. An algebra for such trees is described.

In order to account for nondeterminism and concurrency we use the theory of processes outlined in Chapter 2 to define a class of processes which has a tree structure. The evolution of such trees corresponds to the execution of the logic program. This is developed in a setting involving binary relations on objects, which is more general than that of logic programs.

In Chapter 6 we consider how to map from logic programs to the domains described in Chapter 5. *Rules* are to be considered as finite syntactic representations of binary relations. We first consider how rules can be mapped to the domains of Chapter 5, and then consider how logic programs can be considered as rules.

2. Domains

2.1 Introduction

The central problem in providing a mathematical semantics for a programming language is how to map from texts in that language to a suitably chosen domain of mathematical objects. Such a mapping should capture our intuitions about the meaning of such texts as well as be precisely defined. The domain itself should somehow capture our intuitions about the possible processes that can be generated by programs in the language. Unfortunately not every set of mathematical objects can provide an adequate domain; we must be somewhat selective. Fortunately there already is a theory which characterizes the criteria that a set must satisfy in order for it to be a reasonable domain. See, for example, [STOY].

In this chapter we review the general theory of domains.

2.2 Mathematical Definitions

In this section we review some basic mathematical definitions and fix our notation. When talking about functions we avoid the terms *domain* and *range*. This is because *domain* will have a different meaning. We use instead the terms *source* and *target*. Let S be a set, x,y,z be members of S, and $\leq$ a partial order defined on S. **A** and **E** are the universal and existential quantifier, respectively. **and**, **or**, =>, and $\neg$ are the operators for conjunction, disjunction, implication, and negation, respectively.

2.2.1 Sets, Posets and Functions

DEF *Functional Composition* is denoted by $*$, where:

$$(f*g)(x)=f(g(x))$$

DEF $x \sim y$ *iff* $x \leq y$ **or** $y \leq x$.

In this case x and y are said to be *comparable*.

DEF $x \mid y$ *iff* $\neg(x \sim y)$.

In this case x and y are said to be *incomparable*.

DEF $x \uparrow y$ *iff* $(\mathbf{E}z)(x \leq z$ **and** $y \leq z)$.

In this case x and y are said to be *upward compatible*.

DEF $x \downarrow y$ *iff* $(\mathbf{E}z)(z \leq x$ **and** $z \leq y)$

In this case x and y are said to be *downward compatible*.

DEF $\uparrow x = \{y \in S : x \leq y\}$.

If $Z \subseteq S$, then $\uparrow Z = \{\uparrow x : x \in Z\}$.

DEF $\downarrow x = \{y \in S : x \geq y\}$.

If $Z \subseteq S$, then $\downarrow Z = \{\downarrow x : x \in Z\}$.

DEF $maximal(x) \equiv \uparrow x = \{x\}$

DEF $minimal(x) \equiv \downarrow x = \{x\}$

DEF The set of maximal elements of S that are greater than some x is denoted $[x]$, where:

$$[x] = \{y \in \uparrow x : maximal(y)\}$$

We will be more interested in maximal elements than minimal elements. Thus, no notation is introduced for a set of minimal elements.

DEF $[Z] = \bigcup_{x \in Z} [x]$.

DEF The set of maximal elements with respect to a given set $Z \subseteq S$ is denoted $\lceil Z \rceil$ where:

$$\lceil Z \rceil = \{x : x \in Z \textbf{ and } \neg(\textbf{E}y)(y \in Z \textbf{ and } x < y)\}$$

DEF Similarly, the set of minimal elements with respect to a given set $Z \subseteq S$ is denoted $\lfloor Z \rfloor$ where:

$$\lfloor Z \rfloor = \{x : x \in Z \textbf{ and } \neg(\textbf{E}y)(y \in Z \textbf{ and } x > y)\}$$

DEF $ideal(Z) \equiv (\textbf{A}x)(x \in Z \textbf{ and } y \leq x \Rightarrow y \in Z)$

DEF $filter(Z) \equiv (\textbf{A}x)(x \in Z \textbf{ and } x \leq y \Rightarrow y \in Z)$

DEF *Power* (S) denotes the powerset of S.

DEF *Finite* (S) denotes the set of finite subsets of S.

DEF *Ideals* $(S) = \{Z \subseteq S : ideal(Z)\}$. $\downarrow S$ are the *principal ideals of S*

DEF *Filters* $(S) = \{Z \subseteq S : filter(Z)\}$. $\uparrow S$ are the *principal filters of S*

DEF *Direct* (S) denotes the directed subsets of S. A subset D of S is *directed iff* every pair of elements in D have an upper bound in D.

DEF *Chains* (S) denotes the chains of S. A subset C of S is a *chain iff* each of its elements are comparable. Every chain is a directed set.

DEF S is a *complete partial order (cpo) iff:*

- S has a least element, denoted $\mathbf{bottom}_S$. The subscript will be omitted when it causes no confusion.
- $Z \in Direct(B) \Rightarrow lub(Z) \in S$

Let **Cpo** denote the set of all partial orders.

The following propositions follow from these definitions:

$$\uparrow(X \cup Y) = \uparrow X \cup \uparrow Y$$

$$\uparrow \lfloor Z \rfloor = Z$$

$\downarrow\lceil Z\rceil=Z$

2.2.2 Sequences

Another useful structure associated with a set S is the set of sequences which contain members from S. In addition there is a concatenation operation on such sequences. This operations will be indicated by either juxtaposition or the operator "^". Sequences are actually tuples and in expressions referring to sequences angular brackets will be used to surround constants. Symbols not enclosed in angular brackets are, therefore, variables. We will omit the use of angular brackets when the meaning is clear from the context.

DEF $S^*=\{S^n : 0<n<\omega\}\bigcup\{<>\}$, where $<>$ is the empty sequence.

DEF The length of a sequence is defined as follows:

- $|<>|=0$
- $|<a>w|=|w|+1$

  Clearly, if $w\neq<>$, then $|w|=k$ *iff* $w\in S^k$.

DEF The *prefix ordering* is defined as follows:

- $<>\leq x$, for all $x\in S^*$.
- If $x,y\in S^*$ **and** $x=<a>w$ **and** $y=<a>v$ **and** $w\leq v$ then $x\leq y$

It is well-known that the prefix ordering is a partial ordering with sequences it is not a

complete partial ordering.

2.2.3 Countability

N will denote the set of natural numbers. Occasionally we shall need to provide countability arguments. To aid in these the following bijections will be useful:

$$pair : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

$$seq : \mathbf{N}^* \rightarrow \mathbf{N}$$

$$set : Finite(\mathbf{N}) \rightarrow \mathbf{N}$$

Definitions of these functions and proofs that they are in fact bijections are readily available in many books on recursion. See, for example, [CUTL] or [STOY].

2.2.4 Algebras

A convenient way of characterizing families of functions is through the notion of an *algebra*. An algebra consists of a family of sets, called the *carriers* of the algebra, together with a set of operations defined on those sets. In addition, each set has a name, which is usually called a *sort*, and each function has a name, usually called an *operator*.

The carriers and operations of an algebra are mathematical objects. The sorts and operators are syntactic objects. One can thus use sorts and operators to define a language which has as its meaning an algebra. This syntax is usually given by a *signature*, which is a set containing elements of the form:

$$f : w \rightarrow s$$

where f is an operator, w is a sequence of sorts and s is a sort.

When dealing with signatures we will always have some underlying algebra in mind. The intended meaning of an element in a signature is that the operation associated with the operator f has as its source tuples from the sequence of carriers corresponding to w and has as its target elements of the carrier corresponding to s. That is, a signature element describes the type of parameters to the operation corresponding to f and the type of the result. Signatures can be used to characterize syntax because they tell us how we can construct terms without creating type violations.

Since we always have an underlying algebra in mind we will usually blur the distinction between operators and operations and sort and carriers if this causes no confusion. However, we must provide some definition of the algebra's operations. This will be done either directly, using the set-theoretic definitions of the carriers, or indirectly, using equations involving previously defined operations.

2.3 Why Domains?

Before providing a list of formal definitions it would be profitable to consider informally what properties domains should have. First, it must have objects which correspond to states of a computational process which can be generated after only a finite amount of time. Such elements are called, naturally enough, *finite elements*. We will consider what other kinds of elements a domain can contain shortly. The set of finite elements will never be empty. There will always be an element that corresponds to the state of a process before it has done any work at all.

There are two ways of interpreting these finite elements corresponding to two kinds of programs that are analyzed. One can regard a program as computing a function and the finite elements as approximations to the final value. Alternatively, one can view our program as controlling a system and the finite elements as states of that system. However, for the purposes of a general theory of domains this distinction can largely be ignored; either view will do.

In addition to the elements themselves a domain must have some structure which characterizes how states of a computational process are related. A crucial aspect of computational processes is that they proceed in discrete steps. One should therefore postulate that finite elements be ordered in such a way that earlier states of a process come before later ones. Moreover, the element that represents the beginning of a process should be the least element in that ordering. This ordering can be thought of as an approximation ordering if functions are being computed or a precedence ordering if a process is executing.

It is well known that not every function can be computed in a finite amount of time and that processes such as operating systems are designed so that they may execute for an unbounded amount time. So in addition to the finite elements one must also allow for infinite elements. However, such elements should be infinite in a special way. If a process can only produce finite elements after a finite amount of time, then infinite elements should be obtainable from an infinite number of finite elements which incrementally approximates them. Thus, infinite elements should be limits of infinite sets of finite elements.

Conversely, the domain should contain a limit point whenever there is a set of finite elements which tend towards that limit. In this way all of our elements are either finite or representable by sets of finite elements. Moreover, any functions on or between domains should be determined by their behavior on finite elements. This will referred to this as the *Basis Mapping Property*.

In practice it can be difficult to actually provide a domain not defined from others using domain constructions. One of the reasons is that in seeking to model computational systems one is naturally led to focus on those elements which are finite. These correspond, after all, to the actual objects which are being manipulated at any given time by a system.

An ideal solution would be to define the properties that a structure consisting solely of finite elements should satisfy and then show how a domain can be constructed from such a *predomain*. The necessary mathematical techniques are well-known. See [GUES] for examples of related constructions.

A fortunate fact is that if P is a predomain, then $\langle \mathit{Ideals}(P), \supseteq \rangle$ is a domain with the set of principal ideals as a basis. Moreover, the ordering of P is carried over since $x \leq y \equiv \downarrow x \subseteq \downarrow y$. Thus, any predomain is isomorphic, by an order preserving mapping, to the basis of a domain. It is in this sense that one may say that any predomain can be embedded in a domain. This allows one to model systems using predomains, secure in the knowledge that there is a well-defined construction that allows one to find a domain. It will be our practice in this study to work with predomains while still speaking as if we are using domains.

2.4 Domains: Formal Definition

We now turn to the problem of formalizing our intuitions about domains. The first thing is to formalize what is meant by *tending towards a limit*. This is accomplished by simply identifying it with the notion of *directedness* with respect to the approximation ordering. The *limit* of a directed set is its least upper bound. This leads to the postulate that the least upper bound of all directed sets must exist. Moreover, the least upper bound of finite directed sets of finite elements must themselves be finite elements.

DEF If $S \in \mathbf{Cpo}$, then $fin(S)$ denotes the *finite elements* of S.

$$X \in fin(S) \text{ iff } (\mathbf{A}Z)(Z \in Direct(S) \textbf{ and } x \leq lub(Z) \Rightarrow (\mathbf{E}y)(y \in Z \textbf{ and } x \leq y)$$

DEF A set $B \subseteq S$ is a *basis for S iff*

- $B = fin(S)$
- $x \in S \equiv (\mathbf{E}Z)(Z \in Direct(B) \textbf{ and } x = lub(Z))$
- $x \in B \equiv (\mathbf{E}Z)(Z \in Direct(B) \cap Finite(B) \textbf{ and } x = lub(Z))$

DEF A *predomain* is a poset P such that

- P is countable.
- P has a least element **bottom**.

- Every finite directed subset of P has a least upper bound in P.

Pre will denote the set of predomains.

DEF S is a *domain* (or *countably algebraic cpo*) *iff*:

- S is a *cpo*.
- S has a countable basis.

Dom will denote the set of domains.

DEF A function $f : D_1 \rightarrow D_2$ is *monotonic iff* for all $x, y \in D_1$,

$$x \leq y \Rightarrow f(x) \leq f(y)$$

DEF A function $f : D_1 \rightarrow D_2$ is *continuous iff* for every $X \in Direct(D_1)$,

$$f(lub_1(X)) = lub_2(f(X))$$

$[D_1 \rightarrow D_2]$ denotes the set of all continuous functions from D_1 to D_2. The notion of continuity is a very general one, and is not restricted to domains. To be more precise we will refer to a function as being *allowable* if it is a continuous function between domains.

The following two theorems hold:

Domain Completion Theorem

If $\langle S, \leq \rangle \in$ **Pre** then $\langle Ideals(S), \supseteq \rangle \in$ **Dom**.

Thus the finite elements of a domain determine the content and structure of the whole domain. We now must characterize what kinds of functions we will allow on domains. The result should be that functions on domains are determined by their behavior on finite elements.

Basis Mapping Theorem

Continuous functions between domains have the *Basis Mapping Property, i.e.* if $\mathbf{D_1}, \mathbf{D_2} \in$ **Dom** and $f : D_1 \rightarrow D_2$ is continuous then for all $x \in D_1$,

$$f(x) = lub(\{y : (\mathbf{E}z)(z \leq x \textbf{ and } z \in fin(D_1) \textbf{ and } y \in fin(D_2) \textbf{ and } y \leq f(z))\})$$

2.5 Domain Operations

An important property of domains is that there are a number of operations that allow one to construct complex domains from simpler ones. The definition of these operations is given and the theorem justifying their use is stated without its proof. Let $\mathbf{D_i} = \langle D_i, \leq_i \rangle$ be domains with least elements $\mathbf{bottom}_i$.

DEF $\mathbf{D_1 \times D_2} = \langle D_1 \times D_2, \leq \rangle$ where,

1. $\langle x_1, y_1 \rangle \leq \langle x_2, y_2 \rangle \equiv x_1 \leq_1 y_1$ **and** $x_2 \leq_2 y_2$
2. $\mathbf{bottom}_{\mathbf{D_1 \times D_2}} = \langle \mathbf{bottom}_{\mathbf{D_1}}, \mathbf{bottom}_{\mathbf{D_2}} \rangle$
3. The definition of $\times$ is extended in the obvious way to arbitrary products.

DEF $\mathbf{D_1}+\mathbf{D_2}=<(D_1+D_2)\cup\{\mathbf{bottom}\},\leq>$ where,

1. **bottom** is neither in D_1 nor D_2.
2. $x\in D_1$ **and** $y\in D_2 \Rightarrow x \mid y$
3. for any $x\in D_1+D_2$, **bottom**$\leq x$.

This is extended in the obvious way for arbitrary sums. Note that a new least element is introduced in this construction that is strictly less than $\mathbf{bottom}_1$ and $\mathbf{bottom}_2$. The following definition makes it easy dealing with elements of the summands and their corresponding members in the sum.

DEF Let $\mathbf{D}=\mathbf{D_1}+\cdots\mathbf{D_k}$ and $x\in\mathbf{D_i}$. Then x **in** D denotes the corresponding element (i.e. isomorphic copy) of x in **D**.

DEF $\mathbf{D^*}=<\bigcup\{D^n : n<\omega\},\leq>$, where $\leq$ is the prefix ordering.

DEF If f and g are two functions of the same type, then the *pointwise ordering* between them is defined by:

$$f\leq g \equiv (\mathbf{A}x)(f(x)\leq g(x))$$

DEF $[\mathbf{D_1}\rightarrow\mathbf{D_2}]=<[D_1\rightarrow D_2],\leq>$ where $\leq$ is the pointwise ordering.

In addition to the pointwise ordering, there is a second type of ordering on

functions that we will be concerned with. This is the *transformation ordering*. A transformation will be taken to be a *total* function on some set. Given a transformation f, there may be some subset such that $f(x)=x$. These x are not effected by the transformation. Two transformations are compatible if they do not change some element into different elements. That is, f and g are incompatible if and only if, for some x:

$$f(x)\neq g(x) \textbf{ and } f(x)\neq x \textbf{ and } g(x)\neq x$$

The least upper bound of two compatible transformations is given by taking all the values from each as the changed values of the least upper bound. That is, if $h=lub\{f,g\}$:

$$h(x)=x \text{ if } f(x)=g(x)=x$$

$$h(x)=g(x) \text{ if } f(x)=x$$

$$h(x)=f(x) \text{ if } g(x)=x$$

The definition of compatibility prevents the case where

$$f(x)\neq g(x) \textbf{ and } f(x)\neq x \textbf{ and } g(x)\neq x$$

The fixed points of transformations act like **bottom** for the pointwise ordering. Thus the identity functions corresponds to the least element. When dealing with transformations we will denote the least element by ι, suitably subscripted, if necessary.

DEF The *transformation ordering* is defined by:

$$f \leq g \equiv (\mathbf{A}x)(f(x)=g(x) \textbf{ or } f(x)=x)$$

See *Appendix I* for a proof that this is a partial order and that a predomain is easily constructed.

Domain Construction Theorem

If $\mathbf{D_1}$ and $\mathbf{D_2}$ are domains, then so are $\mathbf{D_1 \times D_2}$, $\mathbf{D_1 + D_2}$, $[\mathbf{D_1 \rightarrow D_2}]$, and $\mathbf{D_1^*}$.

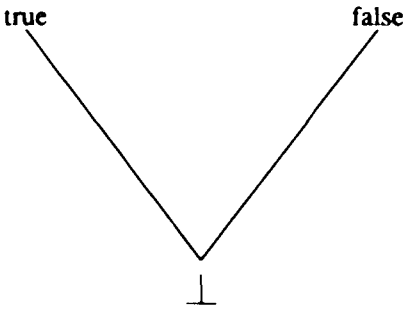
There is also a very easy way to construct a domain out of any countable set. Simply add a new least element and define the ordering such that elements in the original set are incompatible but greater than the new least element. Such posets are called *flat domains* . More formally:

DEF If S is a set, then $S^{\cdot} = \langle S^{\cdot}, \leq \rangle$ where,

- $S^{\cdot} = S \cup \{\textbf{bottom}\}$, for *some* **bottom** $\neg$ *in* S
- $x \in S \Rightarrow \textbf{bottom} \leq x$
- $x, y \in S$ **and** $x \neq y \Rightarrow x \mid y$

Clearly such sets are domains. Although they do not serve all our needs, they do provide an interesting class of domains. They represent those processes which have no intermediate steps. An interesting application of this kind of domain is the theory of computable recursive functions of natural numbers.

Another example, which will play a large part in the domain theory of logic programs is **Bool** = $\{true, false\}^{\perp}$.



For the moment we need it to characterize properties of domains. We would like such properties to be computable in the same way that any other function is computable. That is, the kind of properties we want to consider are those that are definable as continuous functions from a domain to **Bool**.

2.6 Least Fixed Points and Induction

Another important fact about domains, due to Park, is a generalization of the Tarski-Knaster Theorem. It is that monotone functions on domains have least fixed-points and continuous functions have least fixed-points with a closure ordinal of ω.

DEF $\mu f = lub(\{f^n(\textbf{bottom}) : n < \omega\})$

Fixed Point Theorem If f is a continuous function on some domain D, μf is the least

fixed point of f.

DEF A continuous function f is ϕ-*inductive iff* for all $x \in dom(f), \phi(x) => \phi(f(x))$.

Fixed Point Induction Theorem
Suppose that:

- f:**[D→D]**
- ϕ:**[D→Bool]**
- ϕ(**bottom**$_D$) holds
- f is ϕ-*inductive*

Then $\phi(\mu f)$ holds.

There is a distinction between a domain, which is a a pair consisting of a set and a partial order on that set, and the set without the partial order. This is stressed in our notation by writing the domain in boldface. When it it is clear from the context we will relax this conventions and use the same name to refer to both the domain and its underlying set.

2.7 Examples of Domains

It is well known that for any set $\langle Power(S), \subseteq \rangle$ is a complete lattice, and thus a *cpo*. Suppose that S is countable. Since the countability of S implies the countability of $Finite(S)$, $\langle Finite(S), \subseteq \rangle$ is always a predomain. Therefore, the countability of S also implies that $\langle Power(S), \subseteq \rangle \in$ **Dom**.

2.7.1 Streams

As remarked earlier the set of sequences S^* of some set S forms a partial order which is not complete. In particular, infinite chains do not have upper bounds in S^*.

However, if S is countable, then S^* is a predomain under the prefix ordering. This is easy to see. First, note that the empty sequence is the least element in the prefix ordering. Next, S^* is countable. To see this note that one can number elements of S. Let $n(x)$ be the number of an element $x \in S$. Then for each $\langle x_1, \ldots, x_k \rangle \in S^*$ we have a unique number $seq(\langle n(x_1), \cdots, n(x_k) \rangle)$.

We must now establish that finite directed sets have least upper bounds in S^*. This is easy because, under the prefix ordering, compatibility is equivalent to comparability. This implies that all directed sets are chains. The least upper bound of finite chains is simply the largest element in the chain. $\langle Ideals(S^*), \sqsupset \rangle$ is then a domain.

DEF If S is a countable set, then $S^\omega = Ideals(S^*)$.

Note that since any predomain P is countable, that $\langle P^*, \leq \rangle$ is also a predomain. Thus, $\langle P^\omega, \sqsupset \rangle$ is always a domain.

2.7.2 Processes

In Chapter 2 we will encounter domains that are intended to represent the interactions of concurrently executing nondeterministic processes. We defer until then formal definitions.

2.7.3 State Trees

Given any domain **D**, it is possible to construct a new domain which consists of trees whose nodes contain members from **D**. We will encounter such trees in chapter 5 and defer until then formal definitions.

2.7.4 Terms and Substitutions

Two important and related predomains that we consider consist of *terms* and *substitutions*. These are very well studied and we merely review what we need. [LL88] is a good reference to start with.

DEF A *graded alphabet* is a finite set of symbols, each of which is assigned a natural number, called its *arity*.

DEF *Var* denotes the set of variables.

DEF $Term(\Sigma)$ denotes the set of ground terms constructible from the graded alphabet Σ. It is defined by:

$$arity(f)=0 \Rightarrow f \in Term(\Sigma)$$

$$arity(f)=k \text{ and } t_1, \ldots, t_k \in Term(\Sigma) \Rightarrow f(t_1, \ldots, t_k) \in Term(\Sigma)$$

DEF $Term(Var,\Sigma)$ denotes the set of terms constructible from Σ and *Var*. It is defined by:

$$Var \subseteq Term(Var,\Sigma)$$

$$arity(f)=0 \Rightarrow f \in Term(Var,\Sigma)$$

$$arity(f)=k \text{ and } t_1, \ldots, t_k \in Term(Var,\Sigma) \Rightarrow f(t_1, \ldots, t_k) \in Term(Var,\Sigma)$$

DEF A *substitution element* is a member of the set $Var \times Term(Var,\Sigma)$. A substitution element (x,t) will usually be written as $x \leftarrow t$.

DEF A *substitution* is a function from *Var* to $Term(Var,\Sigma)$. $\Theta(Var,\Sigma)$ denotes the set of all such substitutions. Substitutions are also extended as functions on $Term(Var,\Sigma)$ itself by the equation:

$$\theta(f(t_1, \ldots, t_k)) = f(\theta(t_1), \ldots, \theta(t_k))$$

DEF A partial order can be defined on terms using substitutions by:

$$t \leq \theta(t), \text{ for all } \theta \in \Theta(Var,\Sigma)$$

When *Var* and Σ are known from the context, then we will adopt the convention that *Term* stands for $Term(Var,\Sigma)$ and Θ stands for $\Theta(Var,\Sigma)$.

We now consider the predomain of terms. Each term represents a data structure. If a term contains a variable then we may substitute any term for this variable and get a consistent data structure so long as we uniformly do this to ever occurrence in the term. Thus, terms with variables represent partially determined or generalized data structures. If one apples a substitution, then one gets a more determined or more specific data structure. This idea is the basis for the partial order on terms. A variable by itself is completely undetermined and is thus a minimal element. A ground term is completely determined and is thus a maximal element.

This suggests the following partial order on terms:

$$(A\theta \in \Theta)(t \leq \theta(t))$$

We want to be able to construct a domain on Θ as well as on *Term*. One immediate thought is to take unions of substitutions as upper bounds. However, such a union is not necessarily a substitution. It may be the case that there are two distinct values assigned by different atoms to the same variable. What is required is either a way of turning this union into a substitution that is consistent with the individual substitutions that have gone into the union, or a a recognition that such a substitution does not exist. This operation *unifies* the two sets of substitutions.

Martelli and Montanari [MAMO], among others, have studied the relationship between unification and equation solving. The technique that they use to describe unification in terms of equation solving is exactly the technique we need to turn a set of substitutions in a consistent single substitution. In order to apply their techniques we must first view a substitution $\{x_1 \leftarrow t_1, \cdots, x_k \leftarrow t_k\}$ as a set of equations $\{x_1 = t_1, \cdots, x_k = t_k\}$. We then consider the union, not of the set of substitutions, but of the sets of corresponding equations. Let *equations* be the function which turns a set of substitutions into the corresponding set of equations. If S is such a set of substitutions, then we seek a function *solve* which will turn *equations* (S) into a set of equations that form a consistent set of substitutions. That is, *equations*$^{-1}$(*solve* (*equations* (S)) is the desired substitution.

Martelli and Montanari give a nondeterministic algorithm that implements *solve*. Their intentions are to use this algorithm as the theoretical basis for examining a more efficient algorithm for unification. All that we will need is to know what that

algorithm does and that it exists. It is only the definition of *solve* that is important here. We will however consider the operations on sets of equations that are used in their algorithm.

In order to be able to see what *solve* does one must understand what properties a set of equations must satisfy in order to be transformable into a substitution. First, variables may occur exactly once on the left side of equations. Second, if a variable occurs on the left side of an equation then it must not occur on the right. Finally, only variables may occur on the left of equations. Given these conditions it is obvious how to derive the corresponding substitution. A set of equations that satisfies these properties will be said to be *solved.*

Martelli and Montanari define two operations, but in fact use two others. The following set, due to Colmerauer [COL82], allows the same results.

1. Compactification (C): Eliminate any equation of the form "$x=x$".

2. Variable Anteposition (VA): If x is a variable and t is not, then replace $t=x$ by $x=t$.

3. Variable Elimination (VE): If x and y are distinct variables, $x=y$ is in E, and x occurs elsewhere in x, then remove $x=y$ from E and replace each occurrence of x by y.

4. Confrontation (CO): If x is a variable and t_1, t_2 are not variables and $size(t_1) \leq size(t_2)$, then replace the two equations $x=t_1$, $x=t_2$ by the two equations $x=t_1$, $t_1=t_2$.

5. Splitting (S): Replace $f(t_1, \ldots, t_k)=f(s_1, \ldots, s_k)$ by the equations $t_1=s_1, \ldots, t_k=s_k$. If f has arity 0 then eliminate the equation.

It is a matter of inspection to determine if a set of equations can be transformed by at least one of the above rules. Moreover, it can be shown that that there is always only a finite number of applications of these operations. If the resulting system is in solved form then it gives us our desired substitution, otherwise there is no such substitution. Let *unify* denote the function that turns a set of substitutions into as set of equations, applies the above operations until they can be applied no further, and then returns the corresponding final substitution or an error result **bottom**.

We can now define unification between substitutions and terms, respectively, by:

$$unify(\theta_1,\theta_2)=equations^{-1}(solve(equations(\theta_1)\cup equations(\theta_2)))$$

$$unify(t_1,t_2)=equations^{-1}(solve(\{x_1=t_1,x_2=t_2\}))$$

These upper bounds allow us to construct a predomain for both Θ and *Term*. For both terms and substitutions we will be concerned with the predomain rather than the domain. In view of the *Domain Completion Theorem* this causes no difficulty.

We had said that signatures could be used to define syntax. It is exactly the syntax of terms that they define. Consider the signature element:

$$f:w\rightarrow s$$

This expresses that f has an arity equal to the length of w. But it also expresses

something else. Since the result of applying an operation always results in an object that has a type, the signature expresses that there is a restriction as to how one may nest terms. Such a nesting must not violate this type requirement. This can be formalized by a recursive definition of well-typing, which we omit.

3. Processes

3.1 Informal Theory of Processes

We need to associate with each logic program and control component a formal object which characterizes how such a system solves an arbitrary query. Such an object can also represent the possible processes which an interpreter will engage in during the search for this solution. Different control components will determine different processes, all of which arrive at equivalent, but not necessarily identical, solutions. The solutions are all equivalent in that they are all correct answer substitutions. The differences are that the order in which the solutions are generated and the completeness of the generated sets may differ.

To fully accomplish this it is critical to have a theory of processes, formulated in terms of domains, at our disposal. Such a theory has been developed by, among others, Hoare and his associates [HO78, BHR84, HO85]. We sketch here the theory of processes as domains and defer until chapter Chapter 5 the application of this theory to the execution of logic programs.

3.1.1 Events, Traces, and Observations

The basis for their theory is that processes are be understood in terms of what we can observe them do. An *observation* is defined as *a finitely describable experiment to which a process can be subjected.* An *environment* is the context in which we conduct such an experiment. A record of the behavior of a process for a finite amount of time is referred to as a *trace*. *Events* are the smallest unit of observable behavior. Hoare,

et al, informally introduce the main formal concepts of their theory as follows:

> *The ultimate unit in the behavior of a process is an* **event.** *Events are regarded as instantaneous; if we wish to represent an activity with duration, we must introduce two events to represent its start and finish so that other events can occur between them. We shall not be interested in the length of the time interval that separates events, but only in the relative order in which they occur. We let* **A** *stand for the set of all events with which we shall be concerned. The behavior of a process up to some moment in time can be recorded as the sequence of all events in which it has participated; that is known as a* **trace.** *We postulate that a process can only perform a finite number of events in any finite time, and thus all traces have finite length. The set of all possible traces is denoted* $\mathbf{A}^*$.

One of the things we want to understand is how processes interact. One may think of interacting processes as collectively forming, at any point in time, an environment which allows certain observable events. What is possible is determined by the nature of the processes contributing to the environment. The behavior of a single process can then be understood through its interactions with an environment consisting of all other processes currently in the system. If we can understand how this process behaves in all possible environments then we understand completely its observable behavior. Since, for us, this is the only behavior there is, then we would completely understand the process. A convenient way of thinking about a processes environment is that it is a set of events possible for that process at that time.

Two processes that are indistinguishable by every observation are equivalent. This criteria of *observational equivalence* does not preclude the possibility that two processes may have different internal, i.e, unobservable, structures. The existence of such internal structure raises the possibility that processes may spontaneously and invisibly change from the point of view of an external observer. This *nondeterminism* should be distinguished from what might be called *undeterminism*. In the latter case the behavior of the processes would have no cause or explanation, while in the former case the cause of behavior exists but is unknown or ignored. We also allow the possibility of unobserved or ignored interactions.

Conversely, two processes are distinct if, and only if, there is some observation that allows one to distinguish them. They may appear to be equivalent for any finite amount of time, but if they are not equivalent then this *must* be discoverable by a difference in behavior observable after some *finite* amount of time. Therefore, there must exist a trace of each process and an environment such that this environment distinguishes these two process after they have engaged in this trace.

The possibility of nondeterminism makes the task of distinguishing processes quite difficult. Suppose we conduct two experiments and get two different observations. There are two possible explanations. There are different observations either because we have distinct processes or because there is only a single process which has performed some unseen actions, possibility in response to unseen interactions with other processes. The focus therefore shifts from what a process actually does to what is possible or impossible for a process to do. The simplest such property is whether or not a process can proceed in some environment or whether it must deadlock. If one

process may proceed but the other may not then this forms a distinguishing characteristic that is easily specified.

Using these ideas it is now possible to develop concepts that can be formalized using domain theory. First, subsets of A can be used to represent environments. For it to be possible for a process P to make progress in a given environment E, then P must have at least one trace which begins with an event contained within E. Note that we must say *possible* because even if such an event exists it is also possible that P may spontaneously, but silently, transform into a non-identical process Q which does not have a trace which starts with an event in E. We say, in this case, that *P may refuse E*. This is a crucial concept.

If P and Q are distinct, then there must be an environment which, after observing both processes for some time, one may refuse but the other can't. A *failure* of a process is a trace and environment such that after engaging in that trace the process may refuse that environment. Thus, if there is a failure which P has but Q does not then they must be distinct. Conversely, it two processes are distinct then there must be at least one such failure. This allows us to identify processes with their failure sets. Moreover it will turn out that the formal definition of failures leads to a domain. The next step is to formalize these notions by finding suitable mathematical objects that represent them.

3.1.2 Transitions and the Poset of Processes

In addition to the domain-theoretic approach to computation, there is also the operational approach. In the operational approach we characterize computations as

state transformers, with complex computations described in terms of composing simpler computations. For a theory of processes, this would amount to characterizing the evolutions oa a process by the evolution of its state. This should compliment, rather than compete with, the domain-theoretic approach. This requires some notion of equivalence between the two approaches. Hoare also has provided the basis for such an approach. The equivalence is provided by showing that the key concepts of each approach can be defined in terms of the other.

The method is not presented directly in terms of state transformations. Note that after a process P has e ·gaged in an event that it can be considered to have become a different process Q. This new process Q will behave just like P does after P has engaged in its first event. One may thus substitute the notion of a process transformation for that of a state transformation. Furthermore, the transformation that occurs will depend on the which event, of possibly many events, has occurred. This leads to a definition of process transformation as a ternary relation involving two processes and a trace. Such a transformation is described using *transitions*. As defined in [BHR84]:

> *Let s be a trace and let P and Q be processes. A transition is a proposition*
>
> $$P \xrightarrow{s} Q$$
>
> *which means that s is a possible trace of the behavior of P up to some moment in time, and the subsequent behavior of P may be the same as that of Q. Thus if t is a trace of Q, after which it may behave like R, then clearly st (s followed by t) is*

also a possible trace of P, after which it may behave like R.

We will write transitions as *trans*(P,s,Q). If $s=<>$, then we write $P \rightarrow Q$. Since we can always observe a process before is has engaged in any observable events, $P \rightarrow P$ is true for all processes P. Thus, $\rightarrow$ is a reflexive relation on processes. Note also that because of nondeterminism, it is possible for both $P \rightarrow Q$ and $P \neq Q$ to be true! If, however, $P \rightarrow Q$ **and** $Q \rightarrow P$, then there is not any observable way to distinguish P from Q; either may spontaneously and unnoticeably turn into the other. Thus, $\rightarrow$ is anti-symmetric by the criteria of observational equivalence. Moreover, if two processes are equivalent, then no observation can distinguish them. So spontaneous transitions of one to the other will result in no observable difference. Therefore,

$$(P \rightarrow Q \text{ and } Q \rightarrow P) \equiv P = Q$$

Clearly, $\rightarrow$ is also transitive, and thus a partial order on processes.

This partial order, it will be seen, corresponds exactly to the partial order which is used in the domain-theoretic definition. It has a very clear intuitive meaning. Suppose $P \rightarrow Q$ but $P \neq Q$. Then for any s and R such that *trans*(Q,s,R), we also must have *trans*(P,s,R), since P may spontaneously become Q. Any process which Q may become, P may also become. The converse of this does not hold. There are processes which P may turn in to which Q may not. This is required, by observational equivalence, since $P \neq Q$. The interpretation of $P \rightarrow Q$ is then that P is at least as deterministic as Q and strictly less deterministic if $P \neq Q$. The least element in this poset is one which may then spontaneously turn into any other process. Moreover, the

maximal elements are just the deterministic processes. Since this least element is completely chaotic, it is dubbed by Hoare *CHAOS*. It satisfies the law:

$$CHAOS \rightarrow P, \text{ for all processes } P.$$

If D is a deterministic process, then it satisfies the law:

$$D \rightarrow P \equiv D = P$$

3.1.3 CSP as an Algebra of Processes

We are not interested only in defining what processes are; we are also concerned with using them. To facilitate this, we want to avoid having to define each new process from scratch. This requires that we have some way of constructing complex processes form simpler ones. The result is that we need an algebra of processes. CSP, although originally defined as a programming language, can also be defined as an algebra on the domain of processes. Because of its roots as a programming language, it is close to ideas regarding the implementation of concurrent systems. This also allows us to regard it as a system for abstractly describing implementations. It will be our policy to regard CSP in this dual role as an abstract implementation vehicle as well as a formal algebra of processes.

In defining this algebra we have two notions of processes at our disposal, *viz.* the operational notion, using transitions, and the domain-theoretic, using failures. Using transitions, we describe the transition for the complex process using the transitions of the simpler ones. This is done using standard ideas from logic and amounts to an axiomatic description of the operation. The definition of the operations in terms of

failures is done by using set-theoretic operations on the failure set of the operands. Thus, from the domain-theoretic point of view, CSP is a special case of the algebra of sets.

3.2 A Formal Theory of Processes

In this section we provide the formal definitions necessary to define processes mathematically. We begin by providing definitions in terms of traces. Then, we discuss the relation between failures and transitions.

Transitions have already been defined, above. From the discussions in the previous sections, it can be seen that they satisfy the following laws:

L1 $trans(P,s,Q)$ **and** $trans(Q,t,R) \Rightarrow trans(P,s\hat{}t,R)$

L2 $trans(P,s\hat{}t,R) \Rightarrow (\mathbf{E}Q)(trans(P,s,Q)$ **and** $trans(Q,t,R))$

L3 $(P \rightarrow Q$ **and** $Q \rightarrow P) \equiv P=Q$

The following are a number of useful definitions. Note that they can also be defined in terms of failures.

DEF $traces(P) = \{s \in A^{*} : (\mathbf{E}Q)(trans(P,s,Q))\}$

This is the set of all possible behaviors that a process can be observed to engage in. Since $P \rightarrow P$, $\langle\rangle$ is always a member of $traces(P)$.

DEF $initials(P) = \{s \in A : (\mathbf{E}Q)(trans(P, \langle a \rangle, Q))\}$.

initials (P) is the set of those events that a process may engage in on its first step.

DEF $impossible\,(P,X) \equiv X \cap initials\,(P) = \varnothing$.

DEF $refusals\,(P) = \{X \subseteq A : X \text{ is } finite \textbf{ and } (\mathbf{E}Q)(P \rightarrow Q) \textbf{ and } impossible\,(Q,X)\}$.

A *refusal* for P is a finite subset which may prevent P from making any observable progress. Because of nondeterminism, it is possible for both $X \in refusals\,(P)$ and $X \cap initials\,(P) \neq \varnothing$ to hold. This means that an event that is possible for P may seem impossible because P has in fact spontaneously changed into a non-equivalent process. If $refusals\,(P) \neq refusals\,(Q)$, then there is some event, the occurrence of which will distinguish P from Q. This leads us right into the definition of failures in terms of transitions:

DEF $failures\,(P) = \{(s,X) : (\mathbf{E}Q)(trans\,(P,s,Q) \textbf{ and } X \in refusals\,(Q))\}$

The following laws can be proven from this definition. Let P be an arbitrary process and let $F = failures\,(P)$.

P1 $(s,X) \in F \Rightarrow s \in A^{*}$ **and** $X \subseteq A$ **and** X *is finite*

P2 $(\langle\rangle, \varnothing) \in F$

P3 $(s^{\frown}t, X) \in F \Rightarrow (s,X) \in F$

P4 $X \subseteq Y$ **and** $(s,Y) \in F \Rightarrow (s,X) \in F$

P5 $(s,X)\in F$ **and** $(s\hat{}\langle a\rangle,\varnothing)\neg\in F \Rightarrow (s,X\cup\{a\})\in F$

It is also possible to take the laws P1 through P5 as a *definition* of a failure set F.

DEF $PROC=\{X : X\subseteq(A^{*}\times Power(A))$ **and** X *satisfies* $P1-P5\}$

Moreover, one may also then take this as a domain-theoretic definition of processes provided that one can show that the set of all failure sets form a domain and that for all processes P and Q, $P=Q \equiv failures(P)=failures(Q)$, where the first '=' is understood as observational equivalence and the second as identity. We will not prove that *PROC* is a domain; a proof can be found in [BHR84]. We must, however, understand the nature of this domain. It satisfies the following property:

$$P\rightarrow Q \equiv failures(P)\supseteq failures(Q)$$

which implies:

$$P\rightarrow Q \text{ and } Q\rightarrow P \equiv failures(P)=failures(Q)$$

In view of what we've already said regarding the poset of processes, this implies that $P=Q \equiv failures(P)=failures(Q)$. Thus $\rightarrow$ corresponds to the partial order of the superset relation on the set of failures.

3.3 A Basis for the Domain of Processes

There is one aspect of domain theory that is omitted in [BHR84], *viz.* the presentation of a countable basis. This is easily remedied. We seek some countable

set B such that

- $B \subseteq PROC$
- B is countable
- For all $P \in PROC$, there is a set $S \in Direct(B)$ such that $P = \bigcap S$
- If $E \in B$, then there is a finite directed subset of B such that $E = \bigcap S$

We know that any process is a subset of *CHAOS*. Therefore, if we take *CHAOS* as our universe, every process has a complement, denoted $\bar{P}$, such that:

$$P = CHAOS - \bar{P}$$

Note that if P is a process then $\bar{P}$ cannot be a process. This is because $(\langle\rangle, \emptyset)$ must be in every process, and thus cannot be in the complement of any process. We will get our finite elements by defining a partial order on *CHAOS* itself, with $(\langle\rangle, \emptyset)$ as the bottom element.

One of the features of domain theory is that domains can be generated from **bottom** by iterative improvements. Anything generated after a finite number of improvements is a finite element. The rationale for focusing on a processes complement is that the i provement of a process will always result from a removal of some of its elements. The complement of a process provides a convenient way of referring to the elements that must be removed from *CHAOS* to arrive at a given process.

Recall the following facts from section 2.2:

- $\lfloor Z \rfloor = \{x : \neg(\mathbf{E}y)(y \in Z \textbf{ and } y < z)\}$
- $\uparrow Z = \{y : (\mathbf{E})(x \in Z \textbf{ and } x \leq y)\}$
- $\uparrow(X \cup Y) = \uparrow X \cup \uparrow Y$
- $\uparrow \lfloor Z \rfloor = Z$

Now, let $M = \lfloor \bar{P} \rfloor$. This gives us:

$$P = CHAOS - \bar{P}$$

$$P = CHAOS - \uparrow M, \text{ since } M = \lfloor \bar{P} \rfloor$$

$$P = CHAOS - \bigcup_{x \in M} \uparrow x$$

$$P = \bigcap_{x \in M} (CHAOS - \uparrow x)$$

This gives us our basis:

$$B = \{CHAOS - \uparrow x : x \in CHAOS \textbf{ and } (\mathbf{E}P)(P \in PROC \textbf{ and } x \in \lfloor \bar{P} \rfloor)\}$$

In fact, we can even get an alternative definition of *PROC* if we can find a property, defined without reference to *PROC* itself, that characterizes for which set Z, $CHAOS - \uparrow Z$ is a process. This property we will call *independence*. It allows the following definition of *PROC*.

Let:

1. A be some countable set.
2. $CHAOS = A^{*} \times Finite(A)$
3. $PROC = \{CHAOS - \uparrow Z : Z \in Power(CHAOS) \textbf{ and } independent(Z)\}$
4. $B = \{CHAOS - \uparrow Z : Z \in Finite(CHAOS) \textbf{ and } independent(Z)\}$

The intuitive idea behind independence is that if $independent(Z)$ and $f_1, f_2 \in P \textbf{ and } Z$ then the removal of either form P does not require the removal of the other to satisfy P_1-P_5.

DEF $(s,X) \leq (t,Y) \equiv [(s=t \textbf{ and } X \subseteq Y) \textbf{ or } (s \leq t \textbf{ and } X = \emptyset)]$

DEF $independent(S) \equiv S \subseteq CHAOS - \{(\langle\rangle, \emptyset)\}$ and no two distinct $f_1, f_2 \in S$ satisfy either of the following:

$$\mathbf{I_1}: f_1 < f_2 \textbf{ or } f_2 < f_1$$

$$\mathbf{I_2}: f_1 = (s, X \cup \{c\}) \textbf{ and } f_2 = (s^{\frown}\langle c \rangle, \emptyset)$$

We have the following:

Poset Lemma

$\langle CHAOS, \leq \rangle$ is a poset.

Basis Theorem for PROC:

$P \in PROC$ *iff* there is some $Z \subseteq A$ such that Z is independent and $P = CHAOS - \uparrow Z$

Proofs for both can be found in *Appendix I*.

3.4 Operations on Processes

In this section we review the definitions of a number of operations given in [BHR84] and [HO85]. Both failures and transitions will be used. A notation is also introduced which differs from the one in [BHR84, HO85]. This is to accommodate the printer. Binary operators will usually be written in prefix. Where it seems to increase readablily binary operators will be written in infix. Also, we will abbreviate expressions of the form $X_1\ op\ \cdots\ op\ X_k$ as $op(X_1, \ldots, X_k)$. If S is some finite set of events and for each $s_i \in S$, $E(s_i)$ is an expression, we will abbreviate $op(E(s_1), \ldots, E(s_k))$ as $op(x \in S : E(x))$. We will also use the symbols P, Q, and R to designate processes, the symbols A, B, and C, to denote sets of events. A will denote the largest such set, of which the others are subsets. The symbols a, b, and c denote events. When giving definitions of operations, we will first give the one in terms of failures and then the one in terms of transitions.

Since *PROC* is a domain, one may consider functions from arbitrary domains to *PROC*. We thus allow expressions of the form $P(x)$ to denote processes, where x is understood to be a parameter to be replaced by an element in some domain. The most frequent use we will make of this is where x is a finite subset of some countable set. This will not usually be remarked upon. In view of the fact that such sets always allow a domain under $\subseteq$ this is allowable.

3.4.1 Special Constants

There are some processes which will play a special role in the CSP algebra. Here

we introduce constants to denote some of these.

We have already seen *CHAOS*. It has the following definition:

DEF $CHAOS = \{(s,X) : s \in A^* \textbf{ and } X \subseteq A \textbf{ and } X \textit{ is finite}\}$

$(\forall s,P)(trans\,(CHAOS,s,P))$

Recall *CHAOS* is a process capable of unpredictable behavior. Given any event it may either refuse it or engage in it. We will now introduce two new processes. *STOP* is a process that will always refuse, and never engages in, any event. *RUN*, on the other hand, will always engage in, and never refuse any event.

DEF $STOP = \{(\langle\rangle,X) : X \subseteq A \textbf{ and } X \textit{ is finite}\}$

$trans\,(STOP,s,Q) \Rightarrow s = \langle\rangle \textbf{ and } Q = STOP$

DEF $RUN = \{(s,\varnothing) : s \in A^*\}$

$(\forall s)(trans\,(RUN,s,RUN))$

A process is also introduced which can only engage in a single instance of a given event. Such a process will have only a single non-empty trace consisting of just one occurrence of that event. If a is that event, then $proc\,(a)$ denotes that process. *proc* is a function from events to processes.

DEF $proc(a) = \{(\langle\rangle, X) : X \subseteq (A - \{a\})$ **and** X *is finite* $\}$

$trans(proc(a), s, R) \Rightarrow s = \langle a \rangle$ **and** $R = STOP$

Now, *STOP* is a process which simply and unceremoniously halts. It will be useful, especially in defining sequential composition of processes, to have a mechanism to allow a process to halt gracefully. We assume that there is a special event **eop**, for *end of process*, that is in A. We further assume that **eop** will occur only at the end of traces. It is now possible to define processes that halt gracefully. As an example, *SKIP* is a process that does nothing but halt gracefully. It corresponds to a no-op.

DEF $SKIP = proc$ (**eop**)

3.4.2 Nondeterministic Composition

Given two processes, one may form a new process which simply chooses from among these two processes in a nondeterministic matter. The symbol **or** will designate this operation.

DEF **or** $: PROC \times PROC \rightarrow PROC$

or $= P \cup Q$

$trans(P, s, R)$ **or** $trans(Q, s, R) \Rightarrow trans(or(P, Q), s, R)$

3.4.3 Guarded Processes

guard (a,P) is a process which can only engage in a on its first step. After that it behaves like P.

DEF $guard : A \times PROC \rightarrow PROC$

$$guard(a,P) = \{(\langle\rangle, X) : X \subseteq (A - \{a\}) \textbf{ and } X \textit{ is finite}\} \cup \{(\langle a \rangle \hat{} s, X) : (s,X) \in P\}$$

$$trans(P,s,Q) \Rightarrow trans(guard(a,P), \langle a \rangle \hat{} s, Q)$$

guard (a,P) is usually written as $a \rightarrow P$.

3.4.4 Parallel Composition by Interleaving

One may take two processes and run them concurrently and independently. Let *par* designate this operation. If we consider the resulting trace, then clearly if any two events occur in a given order in the trace of P then in the new process they must also occur in that order. However, events that occur in P have no relation to those that occur in Q. Therefore, nothing can be said about the relation between events contributed by P and those contributed by Q to *traces* $(par(P,Q))$. Let an interleaving relation ***shuffle*** be given for A^*, where *shuffle* (s,t,w) iff w results from shuffling, or interleaving, s and t.

DEF $par : PROC \times PROC \rightarrow PROC$

$$par(P,Q) = \{(w,X) : (\mathbf{E}s,t)((s,X) \in P \textbf{ and } (t,X) \in Q \textbf{ and } shuffle(s,t,w))\}$$

$$trans(P,s,P') \textbf{ and } trans(Q,t,Q') \textbf{ and } shuffle(s,t,w) \Rightarrow trans(par(P,Q), w, par(P',Q'))$$

3.4.5 Parallel Composition by Intersection

One may also combine two processes in parallel in such a way that they are required to both always engage in the same event. Thus, the two processes must proceed in a lockstep, synchronized fashion. This forces each to forego the possibility of engaging in events that the other may not.

DEF $sync : PROC \times PROC \rightarrow PROC$

$$sync(P,Q) = \{(s, X \cup Y) : (s,X) \in P \textbf{ and } (s,Y) \in Q\}$$

$$trans(P,s,P') \textbf{ and } trans(Q,sQ') \Rightarrow trans(sync(P,Q), s, sync(P',Q'))$$

$sync(P,Q)$ will sometimes be written as $P \mid\mid Q$.

This definition is too strong for some purposes. It requires that two processes always engage in exactly the same actions. We sometimes want these processes to be able to engage in private events and require synchronization only on common events. To this end there is weak synchronization operator, *w_sync*. It can be defined using *sync* as follows:

DEF $w\_sync : PROC \times PROC \rightarrow PROC$

$$w\_sync(P,Q) = sync(par(P, RUN(X)), par(Q, RUN(Q)))$$

where:

$$X = events(P) - (events(P) \cap events(Q))$$

$$Y = events(Q) - (events(P) \cap events(Q))$$

In this way, $RUN(X)$ ensures that $par(P, RUN(X))$ engages in anything that is possible for $par(Q, RUN(Y))$ and $RUN(Y)$ does the some for $par(R, RUN(X))$.

3.4.6 Conditional Choice

The **or** operator allows one to specify processes which depend on a choice determined by the system in an unknown way. One would also like to specify choices that are made while the process is running. To this end a conditional choice operator, designated *cond* is introduced. This allows us to let the choice depend on the first event that is possible for the environment. Consider the situation where $a \in refusals(P)$ but $a \notin refusals(Q)$. It is possible that $\mathbf{or}(P,Q)$ may refuse a because P was chosen. However, the desired behavior of $cond(P,Q)$ in this case will be to choose Q **after** it has been determined that Q cannot refuse a. If both P and Q treat a the same, then $\mathbf{or}(P,Q)$ will be the same as $cond(P,Q)$.

DEF $cond : PROC \times PROC \rightarrow PROC$

$$cond(P,Q) = \{(\langle\rangle, X) : (\langle\rangle, X) \in P \cap Q\} \cup \{(s,X) : s \neq \langle\rangle \textbf{ and } (s,X) \in P \cup Q\})$$

$$trans(P, \langle a \rangle \hat{} s, R) \textbf{ or } trans(Q, \langle a \rangle \hat{} s, R) \Rightarrow trans(cond(P,Q), \langle a \rangle \hat{} s, R)$$

$\{(\langle\rangle, X) : (\langle\rangle, X) \in P \cap Q\}$ is a set of failures for a process which refuses an event on its first step *only* if it is refused on the first step of *both* P and Q. Thus, $cond(P,Q)$ will refuse an event on its first step only if both P and Q does.

$\{(<a>\hat{}s,X) : (<a>\hat{}s,X) \in P \bigcup Q\})$ is a set of failures of a process which will behave like P or Q so long as the observed behavior begins with the event a. If P cannot engage in a then this set is the same as $\{(<a>\hat{}s,X) : (<a>\hat{}s,X) \in Q\})$. Therefore, in this case, $cond(P,Q)=Q$.

A major use of this operation is in conjunction with guarded processes. The expression $cond(a \rightarrow P, b \rightarrow Q)$ denotes a process which behaves like P or Q, the choice depending on whether a or b may occur. Again, if they both may occur then the choice is nondeterministic. It does, however, introduce at least the possibility of influencing the behavior of processes.

Let P denote a function from a set of events B to the set of processes, *i.e.* $P(x)$ is a parameterized process. Then $(x:B \rightarrow P(x))$ denotes a generalized choice whose behavior is suggested by the expression $cond(a_1 \rightarrow P(a_1), \cdots, a_k \rightarrow P(a_k))$, for all $a_i \in B$. Although it is tempting to use this as a definition, we must stick with saying it "suggests" the behavior of $(x:B \rightarrow P(x))$. B need not be finite and this would require us to define infinite expressions, which we will not do. It is easy enough to modify our definition of *cond* to come up with an appropriate definition. Let **cond** denote this new operations which will take a set of process as its input.

$$\textbf{DEF} \quad \textbf{cond}(S) = \{(<>,X) : (<>,X) \in \bigcap_{P \in S} P\} \bigcup \{(s,X) : s \neq <> \textbf{ and } (s,X) \in \bigcup_{P \in S} P\}$$

$$((\mathbf{E}P)(P \in S \textbf{ and } trans(P, <a>\hat{}s, Q))) => trans(\textbf{cond}(S), <a>\hat{}s, Q)$$

We may now introduce the following:

DEF $(x:B \rightarrow P(x)) = \mathbf{cond}(\bigcup_{x \in B} x \rightarrow P(x))$

We could also have defined $cond(P,Q)$ by $\mathbf{cond}(\{P,Q\})$.

3.4.7 Sequential Composition

In addition to the parallel composition operations, there is also a sequential composition operation. When used as an infix operator it will be denoted ';' and when used as a prefix operator it will be denoted as *sequence*. In order to be able to define $P;Q$ there must be some observable way to know that P is finished. This means that if **eop** occurs in a trace for P and it causes us to start Q, then **eop** can occur in a trace of $P;Q$ only if it occurs at the end of Q. Thus, ';' hides the occurrence of **eop** contributed by P. These considerations lead to the following definitions, where the one using transitions has two clauses:

DEF $sequence:PROC \times PROC \rightarrow PROC$

$$P;Q = \{(s,X) : \neg occur(\mathbf{eop},s) \textbf{ and } (s, X \cup \{\mathbf{eop}\}) \in P\}$$
$$\cup \{(s^\frown t, X) : \neg occur(\mathbf{eop},s) \textbf{ and } s^\frown \mathbf{eop} \in traces(P) \textbf{ and } (t,X) \in Q\}$$

$$trans(P,s,P') \textbf{ and } Q \rightarrow Q' \textbf{ and } \neg occur(\mathbf{eop},s) \Rightarrow trans((P;Q),s,(P';Q'))$$
$$trans(P,s^\frown\langle\mathbf{eop}\rangle,P') \textbf{ and } trans(Q,t,Q') \textbf{ and } \neg occur(\mathbf{eop},s) \Rightarrow trans((P;Q),s^\frown t,Q')$$

3.4.8 Direct Image

One of the things that one wants to be able to do is to use multiple instances of a single process. This cannot be done by simply repeating the expression which defines

a process. The problem is that each such expression explicitly refers to a particular set of events. It is necessary to introduce new sets of events for each new instance of a process. This is possible with *direct image* operators. Suppose that f is an injective function defined on A. Then we can think of of f as renaming events in A. This function can be extended in a natural way to *PROC*. Then one may think of f as producing a copy of a process which retains the original structure but has a new set of underlying events. That is $f(P)$ is just like P except that the names of the events are different.

Formally:

DEF Let $f: A \rightarrow A$ be injective and $P \in PROC$. Then

$$f(P) = \{(f(s), f(X)) : (s, X) \in P\}$$

3.4.9 Synchronization and Communication

The *C* in CSP stands for *Communicating*. So where's the communication? It turns out that in the theoretical version, communication can be defined in terms of the other CSP operators. It is so important in practice that it does get its own notation. The idea is that communication involves four things: a message, a medium, a sender, and a receiver. When a communication event occurs, the sender causes the receiver to get the message through the medium. Now what does *getting the message* mean in this context? It will mean that the receiver will behave differently then if no message was received. Moreover, if more than one message is possible, then it will behave differently depending on which message it actually receives. Thus, the sender has the

opportunity to influence the behavior of the receiver through the occurrence of a communication event.

Consider the expression $(proc(a);P) \mid\mid (x:B \rightarrow Q(x))$. In this case, $P \mid\mid Q(a)$ is the only possible continuation. Here, $proc(a);P$ has influenced the behavior of $(x:B \rightarrow Q(x))$. If some information can be encoded in the event that occurs then this information can be the content. There is also not reason why we could not also let part of the message refer to a *channel* along which the message travels. The solution that Hoare adopts is to introduce a class of events that is an ordered pair consisting of a channel and a message.

Let *CHAN* denote the set of available channels and *MSG* denote the set of possible messages. Then communication events *COMM* is just the set $CHAN \times MSG$. Moreover, it should be possible to express that the behavior of the receiver depends only on the message received, regardless of which channel. This is accomplished by parameterized processes of the form $P(m)$, where m is a message. This allows us to rewrite our example as $(proc(<c,m>);P) \mid\mid (<c,x>:COMM \rightarrow Q(m))$, where $c \in CHAN$, $m \in MSG$, and x is a variable that varies over messages. Note that we have introduced a notion of pattern matching by writing $<c,x>:COMM$ rather than $<x,y>:(\{c\} \times MSG)$. This presents no problem since one can view the former as an abbreviation for the latter.

This occurs so frequently that a special notation is introduced.

DEF If $<c,m> \in COMM$, then $proc(<c,m>) = c!m$

DEF Let $c \in CHANN$, $T \subseteq MSG$, and x and y be variables ranging over channels and messages, respectively.

$(c?y:T \rightarrow P(y)) = (<x,y>:(\{c\} \times T) \rightarrow P(y))$

T is referred to as the *type* of the message. It will be omitted if it is understood from the context. This will frequently happen when $T=MSG$.

The example can now be written as $(c!m ; P) \mid \mid (c?x \rightarrow Q(x))$.

3.4.10 Recursive Definitions

Since **PROC** is a domain, one can define processes with recursive definitions using continuous operators and be guaranteed to have a solution. Therefore, the μ operator is introduced into the CSP formalism. It has the usual meaning as a least fixedpoint operator.

We can use it to define the *RUN* process. *RUN* satisfies the equation:

$$RUN = (x:A \rightarrow RUN)$$

If $(x:A \rightarrow RUN)$ is a continuous function of *RUN*, then $\mu p(x:A \rightarrow p)$ is a solution. $(x:A \rightarrow P)$ is in fact a continuous function of P if $P \in PROC$.

All of the operators introduced so far are in fact continuous. This is not the case with all the operators that Hoare introduces into CSP. In particular, the *hiding* operator, which allows the removal of all occurrences of a specified set of events from a process, introduces the possibility of non-continuity. This is quite reasonable. Continuity tells us that we can determine the value of a function for an infinite input

so long as we can get all the finite pieces that make up that input. If we are allowed to obscure some of the necessary finite pieces, then we should not expect to get a complete value for the function. We still can expect the function to be monotonic. If we provide more pieces of an infinite argument then we should expect a better approximation of its value, even if we are prevented from ever getting it correct.

3.5 Reasoning about CSP

One of the major goals of Hoare was not only to provide a means for defining processes. He also wanted tp provide a way to reason about them. The algebraic treatment that it is given suggests that a major part of the reasoning process would involve the application of algebraic laws to simplify or otherwise transform process expressions. Indeed, a major portion of the space in [HO85] is devoted to presenting, deriving, and justifying such laws. We will simply use such laws as the need arises.

In addition to the algebraic aspects, which involves the manipulation of expressions, there is also the problem of correctness. This does not involve relations of equivalence between expression; rather it involves a relation between the behavior of a process, derived from its definition, and the intended behavior, expressed, somehow, as a specification.

To express this relation Hoare introduces a new predicate **sat**. The idea is that a specification will consist in describing what properties the traces of a process must satisfy. Thus, a specification says something about what we must be able to observe after any finite amount of time that a process has been active. Formally:

DEF Let $s \in A^*$, q be a predicate involving traces, i.e. $q : A^* \rightarrow Bool$, and P a process. Then

$$P \textbf{ sat } q \equiv (\forall s)(s \in traces(P) \Rightarrow q(s))$$

Thus we see that satisfaction of a specification involves more than just a CSP expression. Satisfaction is defined as a relation between CSP expressions and first order predicates involving traces. Note that we are treating q as a two valued and not a three valued predicate.

In addition to this definition, Hoare also provide the following laws involving **sat**.

1. P **sat** *true*
2. P **sat** q **and** P **sat** $r \Rightarrow P$ **sat** $(q$ **and** $r)$
3. $(\forall s)(P$ **sat** $q(s)) \Rightarrow P$ **sat** $((\forall s)q(s))$
4. P **sat** S **and** $(S \Rightarrow T) \Rightarrow P$ **sat** T

Since recursion is an allowable technique for defining processes, induction will have a useful place in a system to reason about processes. In particular, the Fixedpoint Induction Theorem holds for CSP. Hoare introduces a special case of this which is useful when dealing with CSP. See Theorem 9 and Theorem 10 in [BHR84].

4. Logic Programs

4.1 Horn Clause Logic Programs

Although the implementation of full first-order logic is an interesting and important task, there are reasons to consider a more modest result. These reasons concern both conceptual clarity and efficiency. It is sometimes just as important to discover useful restrictions on some computational model as it is to investigate its full power.

Consider the following examples:

- Rather than using arbitrary classes of formal languages, the compiler writer learns to use LR grammars, in return for which he gets automatically generated, linear time parsers.
- Programmers get increased modularity and clarity by restricting themselves to accessing data only indirectly through procedures designed specifically for this purpose.
- Programmers also get increased clarity and verifiability if they restrict themselves to clean control structures and avoid unrestricted jumps.

The point of these examples is that although restrictions are imposed, the programmer really loses nothing but gets a great deal in return. I suggest that this is also the case with the Horn clause subset of logic. Kowalski's book, among others, demonstrates their adequacy for a great many purposes. In addition, Horn clauses are easier to implement. It is just this combination of general adequacy and ease of

implementation that has led to the great interest in Horn clause logic programs. From now on we will restrict ourselves to this formalism.

In this section we summarize some standard results about Horn Clause logic programs over the Herbrand Universe. In Section 3.4 we will turn to a formal account, due to Prof. M. Fitting. A growing body of definitions and results is forming, with contributions from many different sources. Many of the most important results are collected by John Lloyd in [LL88]. The remainder of this section draws heavily on the first two chapters of that book.

In this section we will be informal regarding the syntax of logic programs. The syntax used here is standard PROLOG syntax. In Section 3.4 we will be more formal about syntax.

A logic program is a collection of expressions of the form:

$$A :- B_1, \cdots, B_k, k \geq 0.$$

where the A and each B_i are atoms. Such expressions are called *Horn clauses* or *program clauses*. An *atom* has its usual definition.

If $k>0$ then the clause is an *implication* and if $k=0$ then the clause of a *unit clause*. The atom to the left of the :– is referred to as the *head* of the clause and the list of atoms on the right side is referred to as the *body* of the clause. Thus a unit clause is a program clause with no body.

A logic program is *activated* by presenting it with a *goal clause* or *query*, which is an expression of the form:

$$:\!-B_1, \cdots, B_n, n \geq 0$$

where each B_i are atoms. This will sometimes be written as:

$$?\!-B_1, \cdots, B_n$$

The *empty clause* is a goal clause for which $n=0$. It will be written as:

$$\square$$

To be a *Horn* clause, a clause must have a head. Thus, goals are non-Horn clauses. We define the clause

$$A:\!-B_1, \cdots, B_k$$

as truth-functionally equivalent to

$$(B_1 \textbf{ and } \cdots \textbf{ and } B_k) \rightarrow A$$

and hence equivalent to

$$A \textbf{ or } \neg B_1 \textbf{ or } \cdots \textbf{ or } \neg B_k.$$

As a result, a clause is true if and only if one or both of the following occur:

- The head is true.
- At least one clause in its body is false.

These facts allow one to use the usual methods in reducing the truth value of

complex expressions to the truth values of the atoms that it contains. First note that each clause is treated as though it were *universally closed*, i.e. prefixed with a universal quantifier for each of its variables. The domain is always the *Herbrand universe*, which means that each clause is true iff the result of every *ground substitution* is true. Thus, evaluating any clause depends on being able to evaluate ground atoms. The *Herbrand Base* of a set of clauses S, denoted B_S, is the set of all ground atoms that may be constructed using function symbols and predicate symbols that occur in S. If one can evaluate each member of B_S then one can evaluate each member of S.

Such an evaluation of B_S can be used to separate it into two pieces, one subset containing exactly the members of B_S which are true according to this evaluation and another subset containing exactly the false members.

Alternatively, one can determine an evaluation of B_S by giving a subset of it which contains exactly those members of B_S which are to be true according to this evaluation. An *interpretation* of S is any subset of B_S used for this purpose and the truth or falsity of clauses is defined only with respect to some interpretation.

Some further definitions:

DEF An interpretation I of a set of clauses S is a *model* of S iff ever member of S is true *wrt* I.

DEF S is *satisfiable* iff S has at least one model.

DEF A clause C is a *consequence* of S *iff* C is true *wrt* every model of S.

A result that follows immediately is:

C is a consequence of S *iff* $S \cup \{\neg C\}$ is unsatisfiable.

It is now possible to define what a logic program computes:

DEF Let C be a clause and θ be a substitution. Then θ is an *answer substitution* for C iff it assigns values only to variables occurring in C.

DEF Let P be a logic program. Then θ is a *correct answer substitution* for $P \cup \{:-A_1, \cdots, S_k\}$ iff θ is

- an answer substitution for $:-A_1, \cdots, A_k$
- the universal closure of $(A_1 \& \cdots \& A_k)\theta$ is a consequence of P.

To put it simply, if a logic program is given a goal as input, then it computes a correct answer substitution as output. Since, in general, there may be more than one correct answer substitution, a logic program is nondeterministic.

In addition to describing what a logic program computes, we may also describe *how* it computes it. This is done by procedure called *SLD resolution*. It is described as follows:

DEF A *computation rule* is a function which when given a goal returns a *selected* atom from that goal.

DEF Let G_i=:-$A_1, \cdots, A_k$ and C_i=A :-$B_1, \cdots, B_n$. Then G_{i+1} is a *resolvent* of G_i and C_i iff for some A_j, there is an *mgu* θ_{i+1} such that

- $A\theta_{i+1}=A_j\theta_{i+1}$
- G_{i+1}=:-$(A_1, \cdots, A_{j-1}, B_1, \cdots, B_n, A_{j+2}, \cdots, A_k)\theta_{i+1}$

DEF G_{i+1} is *derived* from G_i and C_i using θ_{i+1} via computation rule R if, in addition to being a resolvent of C_i and G_i, A_j is the atom selected by R.

DEF Let P be a logic program, G a goal, and R a computation rule. Then an *SLD derivation* of $P \cup \{G\}$ *via* R is a sequence of:

- goals $G, G_1, \cdots$
- alphabetic variants of horn clauses $C, C_1, \cdots$
- mgu's $\theta, \theta_1, \cdots$

where each G_{i+1} is derivable from G_{i+1} and C_{i+1} using θ_{i+1} via R.

DEF An *SLD refutation* of $P \cup \{G\}$ via R is an SLD derivation which ends in $\square$.

DEF A goal is said to be *solved* iff and SLD *refutation* has been found for it.

DEF Let $\theta, \theta_1, \cdots, \theta_n$ be the sequence of mgu's used in a refutation of $P \bigcup \{G\}$ using R. Then the composition of these substitutions, restricted to variables occurring in G, is an *R–computed answer substitution.*

Since there may be more than one clause applicable at each step of the derivation there may be more than one possible R-computed answer substitution. Thus, SLD derivations are nondeterministic.

It is desireable to be able to say that a logic program finds correct answer substitutions by implementing SLD resolution to find R-computed answer substitutions. This is justified by the following three theorems, due to Clark and given in Lloyd [LL88].

Soundness Theorem for SLD Resolution

Every R-computed answer substitution is also a correct answer substitution.

The exact converse of this is not true since R-computed answer substitutions are most general while correct ones need not be. However, Clark obtained the following close result:

Completeness Theorem for SLD resolution

Ever correct answer substitution is an instance of an R-computed answer substitution, for some computation rule R.

This last result could use some strengthening since it requires us to find both a computation rule R and an R-computed answer substitution. Clark provided the

necessary strengthening with the following theorem:

Strong Completeness Theorem for SLD resolution

For every computation rule R, every correct answer substitution is an instance of some R-computed answer substitution.

Correct answer substitutions provide the declarative semantics of logic programs, R-computed answer substitutions provide the operational semantics, and the soundness and completeness theorems provide their equivalence.

In establishing the *Strong Completeness Theorem* a result that will be referred to later is proven. It says, in effect, that all computation rules are equivalent modulo renaming of variables.

Independence Theorem for Computation Rules

If R and R_1 are computation rules and θ is a correct answer substitution derived for a goal G using R, the there exists a derivation of a substitution θ_1 for G using R_1 such that θ_1 is correct for G and θ and θ_1 are equivalent modulo renaming of variables.

4.2 The Control of Logic Programs

In this chapter we consider some of the issues that are involved in understanding the control of parallel logic programs. These issues fall into two main categories. First, what are some possible execution models other than the stack based model for sequential execution and what kind of issues generally need to be addressed when considering such models. The second issue is how to actually express control. These are the subjects of the two sections in this chapter.

4.2.1 Kahn/MacQueen Networks

The first model considered is not a logic programming model but an imperative one proposed by G. Kahn and D. MacQueen. See [KAMQ77]. The relevance of this model is that it had a significant influence on the logic programming models that came later. The system that Kahn and MacQueen propose is the extension of conventional imperative languages by the addition of a *process*. A process is similar to a procedure. It has its own data environment and interacts with other processes via parameters. Moreover, there are declarations which characterize processes and process calls which create actual processes as instances of these declarations. Unlike procedures, however, more than one process may be active at the same time. *Active* does not necessarily mean *executing;* such processes can be implemented on a single sequential processor. Rather, it means that an active process has a statement available as the next statement that the system may execute. For sequential languages there is never more than one such statement while for concurrent languages there may be more than one.

Kahn and MacQueen present their system by giving the syntax and an operational semantics for the extensions that they propose. Kahn presents a fix-point semantics in a separate paper [KA74]. The extensions themselves are quite simple. A means for declaring processes, analogous to procedure declarations, must be provided. In addition, a means for processes to interact must be provided. As Kahn and MacQueen say:

The key concepts are **processes** *and structures called channels which interconnect*

> *processes and buffer their communications. Channels carry information in one direction only from a* **producer process** *to a* **consumer process,** *and they behave like unbounded FIFO queues.*

And further on:

> *From an operational point of view, a process network is a collection of independent machines which interact by making demands upon or sending data along communication channels.*

To accomplish this processes, which are syntactically similar to procedures, may have special parameters called *port parameters*. Such formal parameters are declared to be either input or output. The corresponding actual parameters must be variables. A channel is formed by binding the input parameter of a consumer process to the output parameter of a producer process. This binding occurs in a *reconfiguration* statement which is, in effect, an operation which forms complex processes from simpler ones. An instance of a process declaration, and its associated channels, is created when a reconfiguration instruction is activated via an *activation* instruction. At this time a channel is created from process P_o to process P_i if an output parameter of P_o is bound to an input parameter of P_i. In general, a channel can terminate on more than one process but a channel can have only one source.

In order to transmit data along channels two primitive operations are defined. *get* is a function which takes as input a channel and returns the next value on that channel. *put* is a procedure with two parameters - the first is an output port and the second a

value to be placed on the end of its associated channel. Thus, processes can interact by exchanging information. But this is not the only way they interact. If as process executes a *get* on an empty channel then it suspends until the channel becomes non-empty. Thus, a processes rate of progress may also be subject to the behavior of other processes. As Kahn and MacQueen say:

> *A single constraint regulates the activity of processes: if a process requests input data from an empty channel, it must stop and wait until that data is provided by the channel's producer, which must be activated if possible. Given this constraint, a range of scheduling strategies are possible, from* **pure coroutine** *execution where a single process is active at any time to* **full parallelism** *where all processes run except when they are waiting for input. These scheduling strategies all yield the same input/output behavior, because the exclusive use of channels for interprocess communication and the careful choice of data transmission primitives serve to insulate processes from scheduling-dependent information.*

This introduces the key idea of a *data driven computation*. The process that is chosen to execute next is determined by selecting, in some way, from among all those processes that both need to consume some data *and* for which there is some data that is ready to be consumed. The degree of parallelism is equal to the degree that more than one such process can be selected at once. Thus, their semantics allows control to be distributed; it is something which occurs between processes rather than to the system as a whole. This does not preclude an implementation which maintains a

global view, rather it allows such a global view to be ignored by the programmer.

Related to this is the notion of a *dynamic data structure*. Such a structure is one which, at any given moment, is only partially determined. A channel is such a structure. Alternatively, we can think of it as a potentially unbounded structure which, at any given moment, is only partially examined. Such structures are called streams. One may think of the action of a process on such streams as the incremental generation or consumption of a single complex object (the stream) rather than than the manipulation of a multitude of simple objects (objects in the stream).

These streams are a special case of the streams we saw in the section on domains. If M is the set of messages that can be sent, then the set of these streams is M^{ω}. Thus, Kahn and MacQueen have introduced elements of M^{ω} as objects that can be manipulated by their programs. Indeed one of the major thrusts of Kahn and MacQueen is to consider the result of restricting processes to having only a single output port. The semantics of such a subset can be given in terms of functions on streams. This is elaborated in [KA74]. A major theme in the development of parallel logic programs has been the generalization of linear streams to tree structures and the generalization of functions to relations.

4.2.2 The Process Interpretation of Logic Programs

It was van Emden and de Lucena [VED82] who applied the Kahn/MacQueen concepts to logic programs. They introduced the process interpretation of logic programs, which was intended to extend Kowalski's procedural interpretation of logic programs in the same way that Kahn and MacQueen's processes extended imperative

procedures. As they say:

> *In Kowalski's* **procedural** *interpretation of Horn clause logic a goal statement is interpreted as the stack of a single sequential computation. In the* **process** *interpretation, a goal statement must represent the state of a network of sequential computations. As a result,* **in the process interpretation, a goal statement is interpreted as a network of stacks connected by channels with contents given by terms of the goal statement.**

Their system does not alter the declarative semantics; an execution still produces a correct answer substitution. However the programmer may partition the body of a clause into separated stacks by the use of brackets. Each bracketed component corresponds to a sequential query as well as a single process of Kahn and MacQueen. The complete body corresponds to a reconfiguration statement in that it creates a more complex process which produces a single answer substitution. The corresponding processes are created when the clause is invoked.

If a variable occurs in more than one stack then it represents a channel connecting these stacks. Such variables may occur as a sole parameter or at the end of an open list. An open list is one which has a variable at its end rather than null. This means that such a list has not had its last element determined yet; it corresponds exactly to the concept of a stream. If a variable shared between stacks occurs exactly once as a single variable (i.e. as a term by itself) and at least once as the end of a list, then the single occurrence is taken to be the output port and the other occurrences are taken to be the input ports. Otherwise, it is nondeterministic which occurrences are input and

which is output. Thus one difference between Logic processes and Kahn/MacQueen networks is that the input/output role of ports is not declared but determined when needed. This agrees with the case of sequential logic programs where the input/output mode of parameters may not be determined until execution.

4.2.3 AND/OR Processes

One way of providing an operational semantics for parallel logic programs is to construct a process-based model to describe the possible transformations that a goal can undergo. Conery and Kibler present such a model. See [COKI83, COKI85, CON87]. Their processes are of two kinds: AND processes and OR processes. Each process manages the solution of a goal in the SLD derivation tree.

An OR process manages the alternative solutions of an atomic goal. It spawns processes to solve each possible descendant and relays a stream of correct answer substitutions to its parent. An AND processes manages the solution of a non-atomic goal, combining alternatives from each of its literals into a single correct answer substitution. We would like to attempt a solution of each of its literals in parallel. However, the possibility that there may be variables shared between literal means that care must be taken to ensure that any solution does not give the same variable inconsistent bindings.

There are two possibilities - either generate all possible solutions for each literal and attempt to combine them into a consistent one or to incrementally generate a solution and backtrack when failure is unavoidable. Conery and Kibler are only concerned with the latter case. This means that there must be two mechanisms to

handle AND processes: *forward execution* and *backward execution*. Forward execution controls the initiation of subprocesses and coordinates the generation of a single solution. If it ever reaches a point where the generation of a single solution is impossible then backward execution takes over and restarts the search along a different path after it first undoes some of the work of the forward execution mechanism. This is of course typical of any PROLOG system.

In order to proceed with forward execution it is necessary to determine, for each variable in a goal, a single literal which can bind a value to that variable. Such a literal is said to be a **generator** for that variable. Any other literals which contain that variable are said to be **consumers** of that variable. If a literal consumes a variable then it cannot begin to execute before the generator of that variable has completed. Therefore, any literals which do not consume variables may safely be executed together. Such literals either contain no variables or are generators for any that they contain.

In general, it is possible to designate generators in many ways. However, care must be taken. It is possible to choose them in such a way that every literal that generates one variable also consumes another. If this happens then deadlock has occurred since there is no literal that can be chosen for execution. Fortunately this can always be avoided. Conery and Kibler present and algorithm, called the **ordering algorithm,** which partially orders the literals in an AND process. The partial order is such that earlier literals never consume variables from later ones. This means that for all minimal literals an OR process may be started immediately.

This partial order can be represented by a labelled directed graph, which Conery and Kibler call a **dataflow graph.** The nodes in this graph are labelled by literals and there is an arc connecting two nodes if the literal at the tail generates that variable for the literal at the the head. By convention the graph is drawn with generators towards the top. Note that because of the partial order the graph is acyclic.

After a literal in this graph is solved it may be possible to start one of the literals for which it generates variables. Since solutions may contain a variable, it is possible that these new candidate literals still share a variable. As a result, generators may again need to be designated. This can be done by re-applying the ordering algorithm, excluding solved literals, and starting an OR process for any new minimal literals. Since the resulting dataflow graph is smaller, we may view forward execution as the successive reduction of the original dataflow graph.

If the OR process for a literal fails, then we must reject the binding for at least one of the variables that it consumes. Managing this is the purpose of backward execution. The generator for that variable is called the backtrack literal. The backtrack literal is re-inserted and the graph reorganized and restarted.

4.3 Metalevel Inference

When executing logic programs, one cannot just go about blindly generating derivations, even if one has a fast parallel processor. The resulting search space is either needlessly large due to the exploration of irrelevant paths or intractable due to combinatorial explosion. Thus, there is need for a control mechanism. This suggests the following two aspects of logic programs:

1. Describing the user domain with Horn clauses.

2. Controlling how the SLD refutation procedure uses the domain logic.

This leads to Kowalski's equation:

ALGORITHM = LOGIC + CONTROL

The user expresses his domain's logic through Horn clauses. How is the control component expressed? One possibility is to use a procedural language to describe the SLD refutation procedure. This has the disadvantage that the program is now described in two fundamentally different ways. An alternative is to express both in the same formalism. If control is expressed in the same Horn clause logic, then it becomes a metalogic with the logic of the object level as the domain of the control level.

This representation of a formalism within itself is well known, going back to Goedel's arithmetization of metamathematics and Turing's Universal Machine. It enters computer science with McCarthy's definition of LISP in LISP [MAC60]. He did this by first showing how an interpreter for LISP could be constructed from a restricted class of LISP expressions and control structures. Then all that was needed was to represent LISP objects in a computer and implement the basic operations and control structures.

Similar ideas have been explored by researchers in the logic programming community, see [BOKO82, BUWE81, GALA82, KO79a, KO79b, STER82, STER84],

and the production system community, see [DA80a, DA80b, GEO82].

For logic programs, the top-level of such an interpreter can have a very simple recursive structure. Gallarie and Lassere give essentially the following:

```
solve(G) :-
        is_empty(G).
solve(G) :-
        select_atom(G,A),
        select_clause(A,C),
        substitute(G,A,C,G'),
        solve(G').
```

This can be understood as follows:

- A is the atom selected by the computation rule.
- C is the clause selected for A by the search strategy.
- G' is derived from G and C via the current computation rule and search strategy.
- The search process is recursively applied to G'.

The control component consists of the definition of *select_atom* together withe *select_clause*. By providing definitions for these the user may specify control. However, since the control component is also a logic program, it too needs its own control component! Thus the problem of controlling the object level has been pushed

up one level. One can also solve this problem by providing a meta-metalevel. To prevent an infinite regress there must be some level that does not express its control component in this way; some method, other than a higher order logic, must be used at some top level. Such features include extralogical predicates, such as in standard PROLOG, or annotations, such as in IC-PROLOG, PARLOG, or Concurrent PROLOG.

As a further example, consider the following interpreter for PROLOG which is itself written in PROLOG:

```
solve(true).
solve((A,G)) :-
        solve(A),
        solve(G).
solve(A) :-
        atom(A),
        clause(A,B),
        solve(B).
```

This program makes use of the following properties of standard PROLOG implementations:

- *true* is a built-in predicate of arity 0 which always succeeds.
- (P,Q) is equivalent to logical (P **and** Q). This means that the second clause in the program peels of the leftmost atom, solves it and then proceeds to solve

the rest of the goal.

- Since PROLOG solves atoms in a goal from left to right, *solve*(*A*) is done before *solve*(*G*), in the second clause.
- *clause* is a built-in predicate which unifies its first argument with the head the first available program clause. The second argument of *clause* is the body of that program clause. In addition, any bindings generated by the unification are applied to both arguments and upon backtracking *clause* may be resatisfied with the next available program clause, if there is one.

It should be stressed that this reproduces the exact sequential behavior of PROLOG. This will not do for parallel execution of logic programs. In order to achieve this with a logic programming metalanguage there must be extralogical primitive predicates which allow the expression of parallel execution. These will be similar to those found in parallel imperative languages. In addition, selecting a single atom will not do; we must be able to select a group of atoms which may be solved in parallel.

Given these tools we can write an interpreter like the following:

```
solve(Goal) :-
        is_empty(Goal).
solve(Goal) :-
        not_empty(Goal),
        select_group(Goal,Group,Remainder),
```

par_solve(Group),

solve(Remainder).

The key aspect of this interpreter is *par_solve*, which solves the selected group, in parallel, before it solves the remainder of the original goal. While this illustrates the point it is not ideal. We would really like to try to begin solving the *Remainder* as soon as the first atom in *Group* is solved.

4.4 Domains for Logic Programs

The semantics that is presented here is due to Melvin Fitting. It was developed in response to two objections regarding the conventional semantics of logic programs. First, the introduction of a third truth value **bottom** allows for a more natural expression of nonterminating programs. Second, commercial logic programming systems provide more than just terms as data types. The resolution of these objectives is critical to our purposes since we wish to construct metalogic programs which allow the state of execution of object level logic programs to occur as data. Moreover, such programs do not necessarily terminate and we thus are in the situation of computing an infinite data structure, viz. the *final* state of the computation. Indeed the motto of this section should be

The Herbrand Universe provides an Apt model but domains are more Fitting.

The concepts underlying this semantics were designed to fit easily into the theory of domains. Relations are usually defined as sets of tuples taken from some universe. However, one can also define a relation in terms of a *characteristic function*. Such a

function takes a tuple as input and returns *true* if the tuple is in the relation and *false* if it is not. In order to fit this into our theory of domains one should require that the universe from which the tuples are taken be a domain and that the characteristic function be continuous. Moreover, the range of characteristic functions should be a three valued *Bool*, with **bottom**, if nonterminating computations are to be accounted for.

DEF $Char(D,n)=[D^n \rightarrow Bool]$

The semantics for a logic program in some language **L** assigns a relation to each of its predicate symbols via a characteristic function. Such an assignment is called an *interpretation*. In general, there are many possible interpretations for a logic program. The main problem is selecting a single one that captures all the necessary information. How one gets this information for a given predicate will depend upon whether or not it is defined by the given logic program. What it means to be *defined by a logic program* will be defined shortly. If it is not defined then one expects to be given its corresponding relation outright. We will refer to such relations as *given*. In computational terms we expect given relations to be built-in. Indeed one of the benefits of the domain theory approach is that it easily accounts for such relations. If a function is defined then one expects to be able toconstruct its characteristic function from the logic program and the given relations. This is easily accomplished by assigning each logical connective a function on *Bool* and then using a typical recursive procedure for composing such operations. The fact that relations may be defined recursively poses no problems since, being defined by allowable functions, they have

unique least fixed-points which capture all the necessary information about what they could reasonably compute.

4.4.1 Syntax of Logic Programs

In this section we present a description of the syntax of logic programs. It is a simple variation of typical logic program syntax and standard first order notation. It is designed to facilitate semantic analysis rather than programmability.

1. A *literal* is a relation symbol followed by its arguments in parentheses.
2. A *head* is a literal which has only variables as parameters, each of which only occurs once.
3. A *query* or *goal* is a conjunction of literals. A goal may be existentially quantified.
4. A *body* is a disjunctions of goals.
5. A *clause* or *definition* is of the form *head* :– *body*.
6. A *logic program* is a collection of definitions.

In addition the following constraints apply:

- A relation symbol occurs as the head of a definition at most once. Those that do not occur in the head of some definition are *reserved*.
- If a variable occurs in the body of a definition then it is either existentially quantified or it also occurs in the head. The syntax of logic programs usually makes this implicit.

This can be made more formal with the following syntactic equations:

$$PROG = CLAUSE\ .\ PROG \mid CLAUSE$$

$$CLAUSE = HEAD :- DISJUNCT \mid HEAD$$

$$DISJUNCT = QUERY \mid QUERY\ ;\ DISJUNCT$$

$$QUERY = QUANTIFIER(CONJUNCT)$$

$$CONJUNCT = LITERAL \mid LITERAL\ ;\ CONJUNCT$$

Since terms other than single variables are not allowed to occur in the heads of clauses, the kind of parameter passing mechanism used in PROLOG does not apply. However, the variant that is used is not essentially different. For convenience, we will use either syntax. Suppose that a PROLOG program has the clause:

$$p(t) :- q(t)$$

where t is an arbitrary term. We can replace it by:

$$p(X) :- X = t, q(X)$$

where X is a variable that does not occur in t and = unifies X with t. The unification is now explicit rather than implicit. This can be generalized by transforming

$$p(t_1, \ldots, t_k) :- \phi(t_1, \ldots, t_k)$$

to:

$$p(X_1, \ldots, X_k) :- X_1 = t_1, \ldots, X_k = t_k, \phi(X_1, \ldots, X_k)$$

This requires that conjuncts consist of two part: an equational part and a non-equational. The equational part performs unification by an explicit use of an equation solving mechanism. It is possible to complete the transformation from PROLOG syntax to the one used here by noting that once we have remove all nonvariable terms from the heads of clause that we can then combine the bodies that define the same predicate by forming disjunctions. These ideas have been developed further in [JLM86] to include arbitrary equational logics. We restrict ourselves to the Herbrand Universe.

The following syntactic functions will also be useful:

$$defined : PROG \rightarrow Finite\,(PRED)$$

$$reserved : PROG \rightarrow Finite\,(PRED)$$

$$clause\_set : PROG \rightarrow Finite\,(CLAUSE)$$

$$disj\_set : CLAUSE \rightarrow Finite\,(DISJUNCT)$$

$$conj\_set : DISJUNCT \rightarrow Finite\,(CONJUNCT)$$

$$literal\_set : CONJUNCT \rightarrow Finite\,(LITERAL)$$

$$eq\_set : CONJUNCT \rightarrow Finite\,(LITERAL)$$

$$neq\_set: CONJUNCT \rightarrow Finite\ (LITERAL)$$

These have the following definitions:

$$clause\_set(P) = \{C : C \in CLAUSE \textbf{ and } C \text{ occurs in } P\}$$

$$head\_set(P) = \{L : L \in LITERAL \textbf{ and } ((\mathbf{E}X)(L :- X \in clause\_set(P)) \textbf{ or } L \in clause\_set(P))\}$$

$$defined(P) = \{R : R \in PRED \textbf{ and } R \text{ occurs in } head\_set(P)\}$$

$$reserved(P) = \{R : R \in PRED \textbf{ and } R \text{ does } \neg \text{ occurs in } head\_set(P)\}$$

$$disj\_set(C) = \{D : D \in DISJUNCT \textbf{ and } D \text{ occurs in } C\}$$

$$conj\_set(D) = \{K : K \in CONJUNCT \textbf{ and } K \text{ occurs in } D\}$$

$$literal\_set(K) = \{L : L \in LITERAL \textbf{ and } L \text{ occurs in } K\}$$

$$eq\_set(K) = \{L : L \in LITERAL \textbf{ and } L \text{ occurs in } K \textbf{ and } L \text{ is an identity}\}$$

$$neq\_set(K) = \{L : L \in LITERAL \textbf{ and } L \text{ occurs in } K \textbf{ and } L \text{ is } \neg \text{ an identity}\}$$

4.4.2 Semantics of Logic Programs

We now begin the process of formally defining our semantics. This will be in two stages. First, the semantics of Boolean operators on computable relations is defined. This is an extension, due originally to Kleene, to include **bottom**. These operators are defined via the following tables. From these tables its is easily verified that the functions that they describe are monotonic. Since they apply to a finite domain they are thus continuous as well.

and	*true*	*false*	bottom
true	*true*	*false*	bottom
false	*false*	*false*	*false*
bottom	bottom	*false*	bottom

or	*true*	*false*	bottom
true	*true*	*true*	*true*
false	*true*	*false*	bottom
bottom	*true*	bottom	bottom

:-	*true*	*false*	bottom
true	*true*	*true*	*true*
false	*false*	*true*	bottom
bottom	bottom	*true*	bottom

=>	*true*	*false*	bottom
true	*true*	*false*	bottom
false	*true*	*true*	*true*
bottom	*true*	bottom	bottom

≡	*true*	*false*	bottom
true	*true*	*false*	bottom
false	*false*	*true*	bottom
bottom	bottom	bottom	bottom

¬	*true*	*false*	**bottom**
	false	*true*	**bottom**

An operator is also needed to deal with existential quantification. In $(\mathbf{E}x)\phi(x)$, the existential quantifier tells one how the truth value of the open formula $\phi(x)$ depends on the values that can be given to x and the result of evaluating ϕ with this new value. One should thus be able to construct $\chi((\mathbf{E}x)\phi(x))$ from $\chi(\phi(x))$. However, x should be a parameter of $\chi(\phi(x))$ but not be a parameter of $\chi((\mathbf{E}x)\phi(x))$. In general, ϕ may have more than one free variable. Each of these other variables is universally quantified. Thus, to interpret existential quantifiers an operator is needed which takes characteristic functions of arity $n+1$ to characteristic functions of arity n, where $n>0$. Moreover, the variable quantified may be any of the open variables. This operator may be viewed either as a function that takes a position as a parameter or as a member of a family of functions indexed by the position of the quantified variable. We will choose the latter. This leads to the following:

DEF $(\mathbf{E}_i):[\mathbf{D}^{n+1}\rightarrow\mathbf{Bool}]\rightarrow[\mathbf{D}^{n}\rightarrow\mathbf{Bool}]$

where:

$$(\mathbf{E}_i\chi)(x_1,\ldots,x_n)=\begin{cases} \textit{true} & \text{if } \chi(x_1,\ldots,x_{i-1},a,x_i,\ldots,x_n)=\textit{true} \text{ for } \textit{some } a\in\mathbf{D} \\ \textit{false} & \text{if } \chi(x_1,\ldots,x_{i-1},a,x_i,\ldots,x_n)=\textit{false} \text{ for } \textit{every } a\in\mathbf{D} \\ \textbf{bottom} & \textit{otherwise} \end{cases}$$

Since we need $(\mathbf{E}_i f)$ to be allowable, we need the following theorem:

Allowability of $(\mathbf{E}_i\chi)$

If $\chi \in Char(D,n+1)$ then $(\mathbf{E}_i\chi) \in Char(D,n)$.

Proof See *Appendix I*.

For each given relation it is assumed that there is some pre-defined method of computing it. The purpose of a logic programming language will be to allow one to define more complex relations in terms of given relations and logical operators.

4.4.3 Domains for Logic Programs

The task now is to construct a suitable domain for a language **L**. Let $R_1, \cdots, R_n$ be the predicate symbols of **L**. Without loss of generality assume that $R_1, \cdots, R_k$ are the reserved predicate symbols and the rest, if any, are defined. One wants to be able to map each R_i onto a suitable χ_i. It is possoble to construct n domains, each consisting of a single characteristic function. Alternatively, again following Fitting, one can construct a single function which captures all the information of each of the individual characteristic functions $\chi_1, \cdots, \chi_n$.

This more convenient construction is easily accomplished using standard domain constructions. The single function that we construct has as its source the disjoint union of the sources of each of $\chi_1, \cdots, \chi_n$. We expect that if we give this function a tuple from one of the summands in its source that it will give the same result as the corresponding characteristic function. This single function will be referred to as an *interpretation* and its source as an *interpretation space*. Moreover, it should be restricted to interpretations that can be defined by logical operations on the set of given characteristic functions. To that end an object is introduced, called a *data*

structure, which consists of a domain and a set of given characteristic functions over that domain. A data structure, by itself, determines a kind of minimal interpretation which gives meaningful results when applied to tuples from the sources of the functions in the data structure but is undefined for all other tuples. A logic program which uses a data structure will then have the effect of extending this interpretation to a larger one which captures the meaning of the program.

More formally, fix a language as **L** and a base domain as **D**.

DEF An *interpretation space* is a domain of the form $\mathbf{D_1}+\cdots+\mathbf{D_n}$ where for each $\mathbf{D_i}$, $\mathbf{D_i}=\mathbf{D}^{\mathbf{arity(R_i)}}$. We refer to **D** as the *base* domain of the interpretation. Since the only requirement that we have placed on **D** is that it is a domain, it can be a complex construction from simpler domains. For example, **D** may be **NUM+STREAMS+TERMS+[TERMS→BOOL]**.

DEF An interpretation (over an interpretation space *IS*) is an allowable function from an interpretation space *IS* to *Bool*. We denote the set of all interpretations over D by **INT**. That is, **INT**=[*IS*→*BOOL*].

DEF $\langle \mathbf{D};\chi_1,\ldots,\chi_k\rangle$ is a *data structure* where:

- **D** is a non-empty domain.
- Each χ_i is a characteristic function on **D** of a *given relation of arity* $\alpha(i)$.

DEF Let DS be a data structure defined by $DS=<\mathbf{D};\chi_1, \cdots ,\chi_k>$. Then $INT(DS)=\{I \in \mathbf{INT} : I(x \text{ in } D_i)=\chi_i(x), \text{for} i \leq k\}$. That is, $INT(DS)$ is the set of interpretations that are consistent with DS.

It is $INT(DS)$ that we now focus on. It is from here that we must find *the* interpretation of a logic program. The way things have been defined, $INT(DS)$ is a domain. This follows from the fact that it was defined from domains using domain constructions. Since an interpretation is a function, this means that $INT(DS)$ is amenable to the kind of fixed point techniques that allow recursive definitions. We will take advantage of this fact by using the standard continuous approximation function for logic programs, *viz.* T_P. The least fixed point μT_P will give the desired interpretation. Notice that the definitions of interpretation, interpretation space, and data structure make no mention of programs; they only involve the language **L**. Thus a language and a data structure determine a broad range of interpretations. The selection of a program should then further select a single interpretation.

DEF ψ is a function which maps constant symbols to their corresponding elements in **D**. We extend ψ so that it maps tuples of constant symbols to tuples in $\mathbf{D}^n$ such that $\psi(<c_1, \cdots ,c_n>)=<\psi(c_1), \cdots ,\psi(c_n)>$. It is assumed that every finite element of **D** has a name.

DEF v is the function which evaluates formulas according to a given interpretation. v:**L**→**INT(DS)**→**BOOL** such that:

1. $\nu(R_i(\mathbf{a})) = \chi_i(\psi(\mathbf{a}))$, if R_i is reserved.

2. $\nu(R_i(\mathbf{a})) = \nu(\phi_i(\mathbf{a}))$, if R_i is defined by $\phi(\mathbf{a})$).

3. $\nu(F_1\ op\ F_2) = \lambda I.\ (\nu(F_1)(I)\ op\ \nu(F_2)(I))$

4. $\nu(\mathbf{E}x_i F) = \lambda I.\ (\mathbf{E}_i \nu(F)(I))$

Note that for each formula F, $\lambda I.\ (\nu(F)(I))$ is continuous. This is because all formulas are finite and $\nu(F)(I)$ maps into an expression that is composed only of continuous operators. It is in terms of ν that T_P is now defined.

DEF $T_P(I)(\psi(\mathbf{a})\ in\ \mathbf{D}_i) = \chi_i(\psi(\mathbf{a})\ in\ \mathbf{D}_i)$ if R_i is reserved.

$T_P(I)(\psi(\mathbf{a})\ in\ \mathbf{D}_i) = \nu(R_i(\mathbf{a}))(I)$ if R_i is defined.

Since $\nu(F)$ is continuous and each χ_i is continuous for reserved predicate symbols, T_P is a continuous function on $INT(DS)$. This means that μT_P exists. The crucial fact about μT_P is that it is a minimal model for P. This means that $\nu(C)(\mu T_P) = true$ for any clause C in P and that if this hold for any other interpretation I, that $\mu T_P \leq I$. A proof can be found in [LAM85] where that show that models of a logic program P are exactly the fixed points of T_P. Therefore, every model must be an extension of μT_P.

It is possible to define an evaluation function for closed formulas by:

DEF $eval_P = \lambda F.\ (\nu(F)(\mu T_P))$

A characteristic function for a complete logic program P can be defined as follows:

DEF χ_P:[[$\mathbf{D_1}$+ · · · +$\mathbf{D_n}$]*→**BOOL**] such that:

1. $\chi_P(<>) = true$
2. $\chi_P(<\mathbf{x}\ in\ \mathbf{D}_i>\hat{}Z) = \chi_i(\mathbf{x}\ in\ \mathbf{D}_i)\&\chi_P(Z)$

5. The Semantics of Control

5.1 Introduction

The declarative semantics for logic programs associates with each logic program, in the case of the classical treatment, the set of ground atoms in its minimal model or, in the three valued treatment, a characteristic function. By the completeness theorem, this allows one to understand *what* can be derived from a logic program. This type of semantics does not allow an adequate exposition of the theory of control. In order to arrive at such a theory one must also take into account *how* a goal is proved. It is not enough to consider solely the input/output relation defined by a logic program.

This is not to say that it is unimportant to understand how the input/output behavior is determined by the control component; it is just to say that such a determination is a by-product of control rather than control itself. The SLD derivation tree is a structure which characterizes all *possible* derivations but does not give an adequate account of how a given derivation behaves. In particular, the SLD tree shows the final stage of all possible derivations but does not give information regarding the various intermediate stages.

A similar scenario has occurred in the development of the semantics of imperative languages. Initially, denotational semantics assigned an input/output function as the meaning of a program. Subsequent work on concurrency led to methods that allowed reasoning about the ongoing behavior of programs and the interactions of concurrently executing modules, as well as their input/output behavior. Such methods involve the

theory of processes sketched in Chapter 3.

The execution of a logic program involves the search for a correct answer substitution. The search space examined corresponds to conjunctive queries. We will examine in this chapter a general theory of search based on a set of decomposable objects and a binary relation R, called a *reduction*, defined on such objects. The process of searching from a given point x in a state space corresponds to the incremental construction of enough of the reflexive and transitive closure of R, denoted R^*, to see if any member of some designated class of points is reachable from x.

In trying to characterize such searchs, it is typical to use a tree structure which is gradually developed as the search progresses. Much work has been done on the formal description of trees. One technique for describing trees involves labeling each node in a tree by a sequence of natural numbers and viewing the contents of a tree as being given by a function from these sequences to the values that can occur at a node in the tree.

Let us consider another analogy with imperative languages. In the denotational semantics of such languages it is common to represent the *store* of a computation as a function from locations to storeable values. Locations themselves are represented by the set of natural numbers N, while the storeable values are members of some specified domain D. The store is then represented as a function from N to D and a change to the store, such as resulting from an assignment statement, is represented by a change to this function.

Since N^* is countable, we can use it in place of N for defining a store. Thus a

tree can be viewed as a store with additional structure. Moreover, the structure of this tree could be designed to support the use of R in constructing R^*. This should allow one to define a relation on trees which is derived from R. The reflexive and transitive closure of this relation will then represent the global transitions that occur during a search.

Just as it is typical to focus on the possible sequences of transitions that a store can undergo, so we focus on the possible transitions that a search tree can undergo. However, since these transitions are defined with a relation rather than a function, we must account for the possibility of nondeterminism. For this we turn to the theory of processes. There will be two points of view from which to consider such process. There is a local process, which represents the transitions of individual nodes and a global process, which represents behavior of the tree as a whole.

How do we view these local and global processes? The activity that occurs at a given node is primarily concerned with the use of the reduction relation R. New nodes are spawned to solve tasks requested by parents. Thus, the global activity will involve interactions between ancestors and descendents. It is for this reason that we turn to *process trees* which are intended to mimic the local and global transitions of search trees. They provide a structure to manage the interactions of a process with its descendents. A process that spawns subproblems extends the process tree one level. It also adds communication links between the process and its descendents. These links will be used for the descendents to relay information to their parents regarding their success or failure at obtaining a solution.

Then it would be possible to distinguish the effects of different control components by the processes that they define. Furthermore, some aspect of the search mechanism **must** be represented since control is not a property of the state space alone but a relation between a state space and a search mechanism. One must consider not only states in the state space, but also states of the search mechanism. This is accomplished by allowing control information to be stored both globally and at each node in the search tree. Thus, the trees we will be considering have more information stored at each node than just a goal, as in an SLD tree.

5.2 Reductions

5.2.1 Definitions

The primary concept that we will use in giving a semantics for search is that of *reduction*. Very simply, a reduction is a finitely generated binary relation R on some countable set S. See [HU80]. A relation is finitely generated if, for each $x \in S$, there is only a finite number of distinct $y \in S$ such that xRy. The concept of reduction is also important since the procedural semantics of logic programs is defined in terms of reducing an initial goal to an empty goal. Moreover, we will define a parallel search as a reduction on process trees defined by reductions on its nodes. Members of R will be referred to as *reduction pairs*.

We define binary relations as sets of ordered pairs. There is another way or representing them using functions on $Power(S)$. Intuitively the functions takes a subset X of S as input and returns that subset Y of S reachable from X

DEF $\Delta: Power(S) \rightarrow Power(S)$ such that:

$$\Delta(Z) = \{\, y : (\mathbf{E}x)(x \in Z \textbf{ and } xRy)\}$$

It follows from this that:

$$xRy \equiv y \in \Delta(\{x\})$$

Moreover, it is easy to see that:

$$\Delta(Z_1 \cup Z_2) = \Delta(Z_1) \cup \Delta(Z_2)$$

This implies that Δ is determined by its behavior on singleton sets, i.e.

$$\Delta(Z) = \bigcup_{x \in Z} \Delta(\{x\})$$

The function *rel*, which maps a set of ordered pairs into its powerset representation, is defined by:

DEF $rel: Power(S^2) \rightarrow [Power(S) \rightarrow Power(S)]$ such that:

$$rel(R) = \{(Z_1, Z_2) : (\mathbf{A}x, y)((x \in Z_1 \textbf{ and } xRy) \equiv y \in Z_2)\}$$

Reachability in a finite number of steps is defined by:

DEF

$$\Delta^i(X) = \begin{cases} X & i=0 \\ \Delta(\Delta^{i-1}(X)) & i>0 \end{cases}$$

DEF $\Delta^*(X) = \bigcup_{i=0}^{i=\omega} \Delta^i(X)$

A third way that one can represent reductions is by monadic functions on S. Each reduction pair (x,y) defines a transformation that maps x into y but leaves other members of S unaffected. If we combine all such functions in all possible consistent ways then this new set of functions captures the same information as R.

DEF $reduce : S \times S \times S \times \rightarrow S$

where:

$$reduce(x,y,z) = \begin{cases} y & \text{if } x=z \\ z & \text{if } x \neq z \end{cases}$$

DEF $Trans_0(R) = \{f : (\mathbf{E}x,y)(xRy \textbf{ and } f=\lambda z.reduce(x,y,z))\} \cup \{\iota\}$

DEF $Trans(R) = \{lub(Z) : Z \in Direct(Trans_0(R))\}$

where *lub* is taken with respect to the transformation ordering. See Chapter 2.

This leads to the following:

$$y \in \Delta(\{x\}) \equiv xRy \equiv (\mathbf{E}f)(f \in Trans(R) \textbf{ and } f(x)=y)$$

DEF A *normal form* is an irreducible object. *NORM* is the set of such objects, and is defined by:

$$NORM = \{x : \Delta(\{x\}) = \varnothing\}$$

DEF The *normal forms* of subset Z of S denoted $\eta(Z)$ is defined by:

$$\eta(Z) = \Delta^*(Z) \cap NORM$$

Note that $NORM = \eta(S)$.

We have talked about reducing objects but we are interested in problem solving and searching. The composition of the sequence of reductions that is generated in reducing an object to a normal form can be taken as an object being searched for or as the solution to a problem so, long as one can take objects to represent problems. Those objects that are goals are problems that are *self-evidently solved* whereas non-goals need to be worked on. We will tend to speak interchangeably about objects and problems.

DEF $\psi: S \times Trans(R) \rightarrow Bool$

is defined by:

$$\psi(x, \rho) \equiv \rho(x) \in GOAL$$

A set Z is a *region* for an object $x \in S$ provided that there is a chain of reductions leading from x to any $z \in Z$. Formally:

DEF $region(x,Z) \equiv (\mathbf{A}z)(z \in Z \Rightarrow (\mathbf{E}y)(y \in Z \text{ and } xR^*y))$

DEF $REGIONS(S) = \{region(x,Z) : x \in S\}$

At this point we have enough machinery to define a simple type of state space search. Given some $x \in S$, find some ρ such that $\psi(x, \rho)$. However, there are a number of complexities that must be taken into account.

5.2.2 Reduction Decomposition

The reduction relation induces a structure on S. When we model logic programs we will want to interpret rules as corresponding to this structure. But the syntactic components of a logic program also have another structure induced by substitutions. We require that the domains that we use also be able to represent this structure. Now, substitutions map general terms to more specific ones. To this end the notion of a *generalized object* is introduced along with a domain of transformations, *SPEC*, which maps general objects to more specific ones.

When we consider rules we will consider the relationship between Θ and *SPEC*. However, one thing that is required is that a version of the *Lifting Lemma* for logic programs holds. See [LL88] for a discussion of the "Lifting^Lemma". In terms of reductions it is that for all $\sigma \in SPEC$:

$$\Delta(\sigma(\{x\})) = \sigma(\Delta(\{x\})) \quad (*)$$

We also require that *SPEC* contain an identity function, denoted ι.

The relation given by *SPEC* corresponds to the specialization ordering on terms. Let G denote the subsumption ordering. Then:

$$SPEC = Trans(G)$$

Similarly,

$$RED = Trans(R)$$

The definition of Δ will be modified to include a subscript to indicate whether it involves reduction or specialization, *i.e.*:

$$\Delta_r(Z) = \{y : (\mathbf{E}x)(x \in Z \textbf{ and } xRy)\} = \{y : (\mathbf{E}x, \rho)(x \in Z \textbf{ and } \rho \in RED \textbf{ and } \rho(x) = y)\}$$

$$\Delta_s(Z) = \{y : (\mathbf{E}x)(x \in Z \textbf{ and } xGy)\} = \{y : (\mathbf{E}x, \sigma)(x \in Z \textbf{ and } \sigma \in SPEC \textbf{ and } \sigma(x) = y)\}$$

We can now state (*) as:

$$\Delta_r * \Delta_s = \Delta_s * \Delta_r$$

What are we to make of the relation defined by Δ_s? Let $\Delta = \Delta_r * \Delta_s$. This new Δ also defines a reduction. Each reduction step step defined by Δ proceeds in two phase. First, we find a $\sigma \in SPEC$ to get some $\sigma(x)$. We then find some $\rho \in RED$ and obtain $\rho(\sigma(x))$. What kind of control do use to find σ? Well, if we're dealing with terms then we'll surely use unification. We can view unification as a search for a substitution that transforms a term to a left side of some reduction pair. Subsequent processing will then replace the transformed term by the right side of that reduction

pair. This replacement is given by some $\rho \in RED$. To avoid having to change the left side must have an infinite number of such pairs. This is alright since we are dealing with mathematical models. As a result, $\Delta_r * \Delta_s$ corresponds to the resolution relation. From this point of view unification is an implementation concept and not a semantic one.

Recall that logic programs allow the use of incomplete or partially determined data structures. Unification fills in just enough of the data structure to allow a resolution step to proceed. Term rewriting systems, and indeed most formalisms, require complete data structures. Partial evaluations systems are an exception. So what is usual for logic programs is exceptional for other formalisms. Viewing substitutions as a reduction relation allows one to view all of these formalisms in the same light. They are all definable by reductions, albeit a compound one in the case of logic programs.

Furthermore, when a logic program is executed, a transformation of the following form occurs, where $\rho_i \in RED$ and $\sigma_i \in SPEC$:

$$(\rho_1 * \sigma_1) * \cdots * (\rho_k * \sigma_k)$$

Since * is associative, ρ's commute with σ's, and each of *SPEC* amd *RED* is closed under *, this expression may be rewritten as:

$$\rho * \sigma$$

Where $\rho = \rho_1 * \cdots * \rho_k$ and $\sigma = \sigma_1 * \cdots * \sigma_k$.

For logic programs this σ yields a final substitution. This points out a connection between *computation* and *verification*. Each ρ_i corresponds to a step in a propositional derivation; no data is generated. Each σ_i corresponds to data that is generated but involves no propositional steps.

This shows that one can independently compute (or guess) σ and then verify using ρ. Alternatively, and more realistically, one can interleave data generation and verification. One may view resolution as a process which incrementally generates data that is guaranteed to allow an incremental step of verification.

5.2.3 Confluence and Nondeterminism

When we are searching a space using reductions, there is usually a number of different reductions applicable to any object. This results in choice points and alternate reduction paths. However, in order to be complete, it is necessary to be able to ultimately arrive at any solution, regardless of the path chosen. There is a property of reductions, called *confluence*, which says that for every choice point there is another point at which alternate paths will meet. See [RO73,HU80]. Formally stated:

DEF $confluent(R) \equiv (\mathbf{A}x,y_1,y_2)(xR^*y_1 \textbf{ and } xR^*y_2 \Rightarrow (\mathbf{E}z)(y_1R^*z \textbf{ and } y_2R^*z))$

We can define an even stronger property, *commutativity*, by:

DEF $commutative(R) \equiv (\mathbf{A}x,y_1,y_2)(xRy_1 \textbf{ and } xRy_2 \Rightarrow (\mathbf{E}z)(y_1Rz \textbf{ and } y_2Rz)$

This can also be expressed in terms of Δ as:

$$\textit{confluent}\,(\Delta) \equiv (\mathbf{A}x,y,z)(x,y \in \Delta^*(\{z\}) => \Delta^*(\{x\}) \cap \Delta^*(\{y\}) \neq \varnothing)$$

$$\textit{commutative}\,(\Delta) \equiv (\mathbf{A}x,y,z)(x,y \in \Delta(\{z\}) => \Delta(\{x\}) \cap \Delta(\{y\}) \neq \varnothing)$$

Clearly, commutativity implies confluence, but not *vice versa*. The importance of confluent reductions is that elements that have normal forms have unique normal forms. See [HU80] for a proof of this. This means that even though there is local nondeterminism, the system is globally deterministic. In fact, one may even define a function, which may be partial, such that $f(x)$ is the unique normal form of x. Furthermore, one may allocate different processors to different paths, accepting the first normal form reached, since all other normal forms are the same.

However, reductions are not, in general, confluent. In particular, logic programs considered as reductions are not confluent. One way of achieving confluence is to embed non-confluent reductions in confluent ones. Consider yet another analogy with sequential computations. The notions of a *continuation* is used to model jumps in the control flow of a program. Continuations represent *the rest of the program*.

Searching a previously ignored path is also a jump in control flow. The kind of structures needed will represent *the rest of the choice points* and will represent a phase in the search for solutions. Since such structures are themselves objects, one can construct a search system for continuations. Each object corresponds to a choice point, and at any given time a continuation will record any as yet unused reductions applicable to its choice point. Since the reductions it represents are not necessarily confluent, there may be many solutions that it discovers. Normal forms for

continuations will occur when all possible alternatives have been exhausted. We will use processes to represent such continuations. Transitions of such processes will correspond to the search process.

A different problem, also solvable by continuations, is looping. Searching using reductions is essentially a process of constructing enough of Δ^* to see if it contains a goal. This process is carried out incrementally. The relation that corresponds to Δ^* is R^*. While R^* is, by definition, reflexive and transitive, it is not necessarily anti-symmetric. Suppose that x and y are distinct and that xRy and yRx. Then we can follow a path that goes form x to y to x to y, etc. Such obvious loops must be eliminated. This can be done if there is some way of recording what points have been visited. Continuations can be used for this purpose.

The problem now to be solved is how to embed an arbitrary reduction system in a commutative one. The solution adopted is simple-minded and purely for theoretical purposes. A new reduction system is constructed upon $REGION(S)$ from the one on S. R will ambiguously denote both reductions, as the context allows. The idea is that if $Z_1, Z_2 \in REGIONS(S)$ then $Z_1 R Z_2$ provided that one can obtain Z_2 from Z_1 by reducing some $x \in Z_1$. That is:

$$Z_1 R Z_2 \equiv (\mathbf{E}x,y)(x \in Z_1 \textbf{ and } y \in Z_2 \textbf{ and } y \notin Z_1 \textbf{ and } xRy \textbf{ and } Z_2 = Z_1 \cup \{y\})$$

Clearly, $Z_1 R Z_2 \Rightarrow Z_1 \subset Z_2$, so there can be no looping. Moreover, R on $REGIONS(S)$ is commutative since no elements are removed. It is always possible, at any step, to add any object that is not already there. It will seen in the next section that this has to be complicated to account for problem decompositions.

5.2.4 Object Decomposition

A further complexity arises because when one is solving a complex problem, one often seeks to simplify it by decomposing it into simpler ones. These simpler problems can then be independently solved, ultimately combining these partial solutions into a single solution. This idea is certainly crucial in the execution of logic programs. Since problems are represented as objects in S, we turn to a discussion of decomposition.

After having decomposed a problem, it is not always possible to simply recompose results. There may be some constraints on how and when solutions can be combined. Systems in which there are such constraints are termed *partially decomposable*. For example, logic programs requires that substitutions to literals in a goal be unifiable and the unification of these substitutions yields the answer substitution.

To this end a decomposition function δ is introduced which maps from objects in S to decompositions. For the purposes at hand one can take decompositions to be finite subsets of S. Consider singletons $\{x\}$ from S. Then there should be some $y \in S$ such that $\delta(y)=\{x\}$. That is, decomposing this y yields $\{x\}$. We will require that in all such case $x=y$ and refer to all such objects in S as *atoms*. Moreover, it will required that the target of δ be finite subsets not only of S, but of the set of atoms. $ATOMS(S)$ will denote the set of atoms of S. Moreover, δ^{-1} should also be well defined; it corresponds to composition. This implies that δ is a bijection from S to $Finite(ATOMS(S))$. $Finite(ATOMS(S))$ will be abbreviated as $decomp(S)$.

In addition to defining δ, we also want to induce on $decomp(S)$ a reduction system

that mirrors the one on S itself. It should allow one to either do all work in S, all work in $decomp(S)$, or a mixture of the two. Thus, one can apply some $f \in RED$ to some $x \in S$. Let

$$\delta(x) = \{a_1, \cdots, a_k\}$$

It is required that there exists a subset $F = \{f_1, \cdots f_k\}$ of RED such that its members may be used to transform the a_i's and then be able to compose these results to get $f(x)$. It is not the case that each $f(a_i) \in decomp(S)$, *i.e.* atoms are not always transformed to atoms. Therefore, to compose the resultsr, it is necessary to first decompose each $f(a_i)$. But it is further required that there is some associative and commutative operation operation on RED, denoted +, such that

$$f(x) = \delta^{-1}((((\bigcup \delta)(f_1 +, \cdots, + f_k)(\{a_1, \ldots, a_k\}))$$

Put another way:

$$\delta * f = (\bigcup \delta) * (\Sigma F) * \delta$$

In general, it will not be the case that ΣF is defined for each $F \subseteq RED$. Those which do not represent sets of solutions that solve individual problems but cannot be combined to yield a common solution. Conversely, if one first decomposes an object and applies transformations which admit a common solution, then there should be some single transformation that could also be used without decomposition.

What kind of functions should be considered as candidates for F? At first thought any member of RED seems to be a good candidate. However, the point of considering

object decomposition is to avoid doing much of the work involved in reducing atoms. One wants to be able to consider just those functions that result in transforming atoms to normal forms. Following Conery, we will suppose that there are oracles which, for any atom, will provide such solutions. In fact there will be a separate process that computes these solutions as a service for each process.

We first extend the definition of ψ so that it takes two transformations as parameters by:

DEF $\psi: S \times Trans(G) \times Trans(R) \rightarrow Bool$

such that:

$$\psi(x, \sigma, \rho) \equiv (\rho * \sigma)(x) \in GOAL$$

Now suppose that $\delta(x)=\{a_1, \ldots, a_k\}$ and:

$$\psi(a_1, \sigma_1, \rho_1) \textbf{ and } \cdots \textbf{ and } \psi(a_k, \sigma_k, \rho_k)$$

Then it is postulated that:

$$((\rho_1 * \sigma_1) + \cdots + (\rho_k * \sigma_k))(x) \in GOAL$$

What we would like to be able to do is separate the σs from the ρs so that we can find a single σ and a single ρ such that $\psi(x, \sigma, \rho)$ can be derived from all of the $\psi(a_i, \sigma_i, \rho_i)$. In particular, we would like:

$$\sigma = \sigma_1 + \cdots + \sigma_k = \Sigma \sigma_i$$

$$\sigma = \rho_1 + \cdots + \rho_k = \Sigma\rho_i$$

This leads to the further postulate that when both $\Sigma\sigma_i$ and $\Sigma\rho_i$ are defined:

$$\psi(a_1,\sigma_1,\rho_1)\ \textbf{and}\ \cdots\ \textbf{and}\ \psi(a_k,\sigma_k,\rho_k) \Rightarrow (\mathbf{E}x)(x=\delta^{-1}(\{a_1,\ldots,a_k\})$$

Conversely, if

$$\psi(x,\sigma,\rho)\ \textbf{and}\ (\delta(x)=\{a_1,\ldots,a_k\})$$

then there exists

$$\sigma_1,\ldots,\sigma_k,\rho_1,\ldots,\rho_k$$

such that:

$$\sigma=\sigma_1+\cdots+\sigma_k$$

$$\rho=\rho_1+\cdots+\rho_k$$

5.3 Trees

The primary use that we will make of trees is to record the various subproblems that are generated during the course of solving state space search problems. It should be possible to inspect a tree and read off the relations between subproblems by examining the ancestor/descendent relations. Moreover, the incremental steps that occur during the search process should be definable in terms of functions that take as input trees and return as a value a transformed tree. In this section we provide a general definition for such trees and operations.

Let $t: N^* \rightarrow \mathbf{D}$ be a function from finite sequences of natural numbers to some domain **D**. When talking about trees, we shall refer to members of N^* as *locations* and members of **D** as nodes. In these cases we will denote N^* as *Loc* and **D** as *Node*. Among the locations in a tree there are some that have data associated with them and some which don't. Those which do are the *nodes* in the tree. They can be formally defined by:

DEF $node(w,t) \equiv t(w) \neq \mathbf{bottom}_D$

Not all functions from *Loc* to *D* are trees. Only those which are closed under ancestor and left sibling are to be considered trees. This can be formalized by stating the closure conditions which *node* must satisfy:

DEF *t* is a *tree over domain* **D** iff

- $node(w^{\wedge}\langle n\rangle,t) \Rightarrow node(w,t)$ (Closure under ancestor)
- $node(w^{\wedge}\langle n\rangle,t)$ **and** $m<n \Rightarrow node(w^{\wedge}\langle m\rangle,t)$ (Closure under left sibling)

The following subsets of *Loc* are also useful:

DEF $nodes(t) = \{w : node(w,t)\}$

DEF $leaves(t) = \{w : node(w,t)$ **and** $(w' > w \Rightarrow \neg node(w',t))\}$

We also define the following predicate:

DEF $leaf\,(w,t) \equiv w \in leaves\,(t)$

The following are some useful functions which involve various attributes of trees.

DEF $size\,(t) = \mid nodes\,(t) \mid$

DEF $degree(w,t) = \mid \{w\hat{}\langle n\rangle : node\,(w\hat{}\langle n\rangle, t)\} \mid$

DEF $degree(t) = \max\{degree(w,t) : w \in \mathbf{N}^*\}$

DEF $depth\,(t) = \max\{\mid w \mid : node\,(w,t)\}$.

DEF $t \leq t' \equiv t(w) \leq t'(w)$, for all $w \in \mathbf{N}^*$.

DEF Let n be a natural number. The $t \mid n$, the *truncation* of t at n is defined by

- $(t \mid n)(w) = t(v)$, if $\mid w \mid \leq n$ **and** $v = w\hat{}\langle m\rangle$ **and** $m \leq n$
- $(t \mid n)(v) =$ **bottom** otherwise.

Using this notation we can bound both the depth and the degree of an infinite tree. The usual definition of truncation bounds only the depth since the trees that it is applied to already have a finite degree. Our version is a two-dimensional version of the usual definition.

5.4 Operations on Trees

There are three classes of operations trees that are here defined. The first class requires that the nodes in trees are elements of some domain. These operations are thus valid for all trees. The second class of operations assumes that the nodes have a specific structure and consists of functions to manipulate individual nodes with this structure. Specific properties are postulated which members of this second class must satisfy with respect to that structure. Finally, a class of operations is defined that uses the first two types of operations.

5.4.1 Class I Operations

DEF $next: Loc \times Tree \rightarrow Loc$

where:

- $next(w,t) = \textbf{bottom}$, if $\neg node(w,t)$.
- $next(w,t) = \mu n.[\neg node(w \hat{} n,t)]$, if $node(w,t)$.

A *boundary* location is either a leaf or a location that can have a node added without ruining the tree property.

DEF $bound: Loc \times Tree \rightarrow Bool$

is defined by:

$$bound(w,t) \equiv leaf(w,t) \textbf{ or } (\neg node(w,t) \textbf{ and } (\textbf{E}v)(v = next(w,t)))$$

DEF *subtree* : $Loc \times Tree \rightarrow Tree$

subtree (w,t), the subtree of t at w, is defined by:

$$(subtree\,(w,t))(v) = t(w\hat{}v)$$

subtree (w,t) will usually be abbreviated as t/w.

DEF *contents* : $Tree \rightarrow Power\,(Node)$

is defined by:

$$contents\,(t) = \{t(w) : (\mathbf{E}w)(node\,(w,t))\}$$

DEF *assign* : $Loc \times Node \times Tree \rightarrow Tree$

is a function such that *assign* (w,v,t) results in a tree just like t except that the node at w is n. We will usually write *assign* (w,v,t) as $t[w:=v]$. It is defined by:

- $t[w:=v] = t$ if $\neg node\,(w,t)$ **and** $\neg bound\,(w,t)$, otherwise
- $t[w:=v](w) = v$
- $t[w:=v](u) = t(u)$ provided $w \neq u$

DEF *replace* : $Loc \times Node \times Tree \rightarrow Tree$

is a functions where *replace* (w,t,t') is the result of replacing the subtree of t with root at w (i.e. t/w), by t'. We will usually write *replace* (w,t,t') as

$t[w \leftarrow t']$. It is defined by:

- $t[w \leftarrow t'] = t$ if $\neg node(w,t)$ **and** $\neg bound(w,t)$, otherwise
- $t[w \leftarrow t'](u) = t(u)$, if w is not a prefix of u.
- $t[w \leftarrow t'](u) = t'(v)$, if $u = w \hat{} v$.

If we let s_i be of the form $w := v$ or $w \leftarrow t'$, then we can abbreviate expressions of the form:

$$(t[s_1])[s_2]$$

by:

$$t[s_1; s_2].$$

Using conditional expressions and fixpoint operators, we could then define a language of programs on trees. We defer to the future the full elaboration of this. Here these ideas are used informally.

DEF We can now use *assign* to define next kind of tree function:

$$extend : Loc \times Node \times Tree \rightarrow Tree$$

where

$$extend(w,v,t) = t[next(w,t) := v]$$

5.4.2 Class II Operations

We suppose that each node is constructed from some elements from a stock of components. Let

1. *Sort* be some finite set of names used for indexing.
2. Each of $D_1, \ldots, D_k$ be a domain with a unique associated index.
3. $Node = f(D_1, \ldots, D_k)$, where f is a domain-construction.
4. $Comp = D_1 \cup \cdots \cup D_k$ be the set of *components*.

That is, nodes are obtainable by applying domain constructions to components. The nature of these constructions are dependent on a give application, but the following functions must be defined:

DEF $select : Sort \times Node \rightarrow Comp$

where $select(i,v)$ is the element of the domain indexed by i that is chosen for processing.

DEF $add : Sort \times Comp \times Node \rightarrow Node$

where $add(i,d,v)$ is the result of adding a component d from the domain indexed by i to the node v.

DEF $delete : Sort \times Comp \times Node \rightarrow Node$

where $delete(i,d,v)$ is the result of removing a component d from the domain indexed by i in the node v.

These functions are the basis for manipulating individual nodes in a search tree. We need to consider further some aspects of search and what nodes should look like. They must obviously record the status of the problem that they are solving and any information needed to manage the data structures that record these problems. In order to model the interactions between subproblems they must also record which node in the tree is the immediate parent and which descendents have been assigned which tasks. Since we want to be able to distinguish successive steps in the process we separate the events associated with working on a problem from those associated with interactions with processes solvingn other problems. Such interactions will involve the flow of information both inward and outward from nodes. Thus, nodes will contain data structures that record messages that have been received but not yet processed and those which have been locally generated but not yet sent. This leads to the following requirement:

$$Node = Input \times Output \times Data$$

$$Sort = \{input, output, data\}$$

where each of *Input*, *Output*, and *Data* are domains. Therefore, *Node* is also a domain. *Data* is the local data, *Input* is messages that have been received but not yet processed, and *Output* is messages generated but not yet sent. The actual definition of these domains will depend upon a given problem. There are some general requirements:

DEF $sub\_node : Node \rightarrow Node$

sub_node is a primitive function that returns a node which represents a new subproblem to be solved.

DEF $R : Node \times Node \rightarrow Bool$

where xRy is a relation on *Node* that is intended to extend a reduction relation on some set S to *Node* and *Node* represents some data structure that manages the reduction of objects in S.

5.4.3 Class III Operations

Given these primitive functions it is possible to define the necessary operations on trees. These operations form a heterogeneous algebra, with the domains involved as the carriers.

DEF $up : Loc \times Tree \rightarrow Tree$,

$up(w,t)$ results in a tree which has transferred the next waiting message from the output structure of the node at $w\hat{}n$ to the input structure of the node at w. It is defined by:

$$up(w\hat{}n,t) = t[w\hat{}n := delete(output,m,t(w\hat{}n)); w := add(input,m,t(w))]$$

where:

$$m = select(output,t(w\hat{}n))$$

DEF *down* :$Loc \times Tree \rightarrow Tree$,

down is similar to *up* except that the message moves in the opposite direction. It is defined by:

$$down(w\hat{}n,t) = t[w := delete(output,m,t(w)); w\hat{}n := add(input,m,t(m\hat{}n))]$$

where:

$$m = select(output,t(w))$$

DEF *attach* :$Loc \times Node \times Node \rightarrow Tree$

extend is a that inserts a new node into a tree. It is also important to introduce a function which incorporates a new node into a tree. *attach* (w,x,y) results in tree where the node y at location w is attached to the node at x in the sense that x will have its data structures updated so that it acts as the parent for y. The exact details will depend on the definition of the underlying nodes.

DEF *spawn* :$Loc \times Tree \rightarrow Tree$

is defined by:

$$spawn(w,t) = (extend(w,t,v))[w := attach(next(w,t),t(w),v)]$$

where:

$$v = sub\_node(t(w))$$

DEF $read : Loc \times Tree \rightarrow Tree$

defined by:

$$read(w,t) = t[w := add(data, m, delete(input, d, t(w)))]$$

where:

$$m = select(input, w, t(w))$$

DEF $write : Loc \times Tree \rightarrow Tree$

is defined by:

$$write(w,t) = t[w := add(output, m, delete(data, m, t(w)))]$$

where:

$$m = select(data, w, t(w))$$

DEF $step : Loc \times Tree \times Tree \rightarrow Bool$ is defined by:

$$step(w,t,t') \equiv R(t(w),x) \textbf{ and } t' = t[w := v]$$

where:

$$v = add(data, x, t(w))$$

We now use these functions to extend the relation R to the domain of trees as follows

$$R(x,y) \equiv (\textbf{E}w)(step(w,x,y) \textbf{ or } y = f(w,x))$$

where f is one of *up*, *down*, *spawn*, *read*, *write*.

5.4.3.1 An Algebra of Trees

What we have done is to formally define a many-sorted algebra over trees. We have:

$$Carrier = \{Trees, Node, Message, Data, Loc, Bool\}$$

The signature of the algebra is given here:

- Class I

$$next : Loc \times Tree \rightarrow Loc$$

$$assign : Loc \times Node \times Tree \rightarrow Tree$$

$$replace : Loc \times Node \times Tree \rightarrow Tree$$

$$extend : Loc \times Node \times Tree \rightarrow Tree$$

- Class II

$$add : Sort \times Comp \times Node \rightarrow Node$$

$$delete : Sort \times Comp \times Node \rightarrow Node$$

$$sub\_node : Node \rightarrow Node$$

$$select : Sort \times Node \rightarrow Comp$$

$$R : Node \times Node \rightarrow Bool$$

- Class III

$$up : Loc \times Tree \rightarrow Tree,$$

$$down : Loc \times Tree \rightarrow Tree,$$

$$attach : Loc \times Node \times Node \rightarrow Tree$$

$$spawn : Loc \times Tree \rightarrow Tree$$

$$read : Loc \times Tree \rightarrow Tree$$

$$write : Loc \times Tree \rightarrow Tree$$

$$step : Loc \times Tree \times Tree \rightarrow Bool$$

5.4.4 Reduction Trees

One way of implementing searches with processes is to allow a process which represents an object in a state space to spawn a new process for each of its successors. The original process need not remain. Its like a pond full of amoebas endlessly dividing. The notion of decomposition makes this implementation impossible because there must be *something* to perform the re-composition. Even if we get fancy and design some protocol that allows us to distribute the work to the descendents themselves, then the execution of this protocol constitutes a process which is itself the *something*.

It is for this reason that we turn to process trees. They provide a structure to manage the interactions of a process with its descendents. A process that spawns subproblems extends the process tree one level. It also add to a communication network to include links between the process and its descendents. These links will be used for the descendents to relay information to their parents regarding their success or failure at obtaining a solution.

Each node in such a tree will correspond to a scheduled to be solved. The problem will be represented in decomposed form. The solutions to subproblems contained in these decompositions will be accomplished by subtrees of this node spawned just for this purpose. Each node will be a continuation and will thus represent, through time, the evolution of this subproblem. If one took a global look at the tree then one would see two kinds of evolution. The first is the evolution of a single node. The second is the spawning of new nodes to solve subproblems. Incidentally we will not bother to incorporate a mechanism for trees to shrink. Failing nodes will simply become inactive; an actual implementation would deallocate resources from such nodes.

To formalize this we again develop a new reduction system derived from a previous one. Here we will consider two such systems. The first is more elaborate version of reductions on continuations derived from reductions on decompositions. The second is a reduction system derived on trees that is in turn derived from the reduction system on continuations. The structure of continuations will be considered in the next chapter.

5.4.5 Process Trees

The execution of logic programs is essentially a search process and searches are clearly represented by trees. Since we wish to use processes to model such searches, we consider trees of processes. Furthermore, we want such trees to be themselves processes. Since *PROC* is a domain, *Trees*(*PROC*) is also a domain. But we need to be a little more selective in the types of trees we consider. In particular, we want specific nodes in the tree to be able to behave independently of other nodes in the tree. This requires that each node have its own distinct set of events. Moreover, the behavior of the tree itself should depend on the behavior of its nodes and subtrees.

We can easily achieve distinct event sets by requiring that all events of a process at node w have w somehow encoded in them. This is accomplished if we have a set of events which are ordered pairs, the first component being w and the second being used as necessary. To facilitate this we define a renaming function which prefixes a string to the first component in a pair and extend this function to sequences and processes.

DEF Let $A=(\mathbf{N}^* \times S) \cup S$ Then $PTREE \subseteq PROC$ such that:

1. $PTREE \subseteq Trees(PROC)$
2. $events(t(w)) \subseteq \{(w,x) : w \in \mathbf{N}^* \textbf{ and } x \in S\}$

where

$events(P) = \{a : (\mathbf{E}w,x)((w,x) \in P \textbf{ and } a \textit{ occurs in } w)\}$

3. $trans(P,s,Q) => trans(t[w:=ins(w,P)], ins(w,s), t[w:=ins(w,Q)])$

4. $t_1, t_2 \in PTREE$ **and** $trans\,(t_1, s, t_2)$ =>

trans(t[w<-ins(w,t_1)],ins(w,s),t[w<-ins(w,t_2)])

The way that process trees are defined, the only interactions that can occur between members in the trees is communication between parents and children. The third clause in the definition of a process tree says that if a process P can evolve to a process Q by way of event sequence s, then, with the appropriate renaming, any process tree that contains P at some node can evolve to a tree that contains Q at the same node by way of the same event sequence s. The last clause says the same with regard to subtree replacement.

5.5 Mapping Confluent Reductions to Processes

In this section we consider how to characterize searches as processes. It is assumed that all reductions considered are confluent. To specify a search process, it is necessary to specify how the behavior of the process depends on the current state of the search. The state of a reduction can be specified by using parameterized processes. Using the guard construct, it is possible to specify processes whose behavior is influenced by using set membership as a test. That is, the construct:

$$P = (x : T \rightarrow Q(x))$$

specifies that P can only engage in an event from the set T and that it will continue as $Q(x)$. Thus, specifying T and $Q : T \rightarrow PROC$ determines the behavior of P.

If characteristic functions are used to determine T then logic programs can be used to specify the behavior of P. Let χ_i be a characteristic function that has a set of events

as at least one of its parameters. Without loss of generality assume that it is the first parameter. Its corresponding predicate symbol R_i s introduced into CSP by the following postulate:

$$trans((R_i(x, \mathbf{y}) \rightarrow Q(x, \mathbf{y})), a, Q(a, \mathbf{b})) \equiv \chi_i(a, \mathbf{b}))$$

Thus, $(R(x,y) \rightarrow Q(x,y))$ is equivalent to:

$$(x : T \rightarrow Q(x, f(x)))$$

where $T = \{x : R(x,y)\}$ and $(\mathrm{A}x)(x \in T \Rightarrow R(x, f(x)))$. So we have not really added any expressiveness to CSP. We have, however, made it more convenient for our purposes and allowed the incorporation of logic programs into the formalism. Note that this is very much in the spirit of logic programming in that we have a query which has an unbound variable in it which becomes bound during the course of execution.

The use we will make of this addition is to allow us to describe processes which examine their state, make a decision based on this examination of how to proceed, and then act on this decision. Such processes will have the following structure:

$$P(Z) = (decide(x, Z) \rightarrow ACT(x, Z))$$

where *decide* is a predicate symbol, x is a parameter ranging over some predetermined repertoire of actions, and *ACT* is a processes which which is defined so it is equivalent to the result of *doing* x. Typically *ACT* will be defined by cases and will eventually evolve to P, and thus the system as a whole is recursively defined. What is involved in evaluating *decide* can depend on Z itself. It may be that *decide* simply checks some

easily accessible aspect of Z or it may have to perform a complex evaluation. This will depend on the situation.

We can try to eliminate *ACT* by unwinding the recursive definitions. The result will be either:

$$ACT(a,Z) = E_a(f(a,Z)) \text{ provided } decide(a,Z)$$

or:

$$ACT(b,Z) = E_b(g(b,Z)) \text{ provided } decide(b,Z)$$

where E_a is a process expression that contains P and E_b is a process expression that doesn't. This implies that:

$$P(Z) = (decide(x,Z) \rightarrow E_a(f(a,Z))) \text{ provided } test(a,Z)$$

$$P(Z) = (decide(x,Z) \rightarrow E_b(g(b,Z))) \text{ provided } test(b,Z)$$

This can be shortened to:

$$decide(a,Z) => E_a(f(a,Z))$$

$$decide(b,Z) => E_b(g(b,Z)))$$

We can formally introduce => into CSP by the following postulate:

$$trans(R_i(a,y) => Q(a,y), a, Q(a,y)) \equiv \chi_i(a,y)$$

and can also be incorporated on the basis of the following definition suggested by the

Law of Importation in propositional logic. That is:

$$P \Rightarrow (Q \Rightarrow R) = P \textbf{ and } Q \Rightarrow R$$

Note that this is **not** a logical formula (= is used, not ≡) and that in this context **and** is not commutative - the order is important. The language obtained by adding such Horn Clause defined guards will be referred to as *HCSP*. The purpose of this augmented notations is to allow tests to occur in guards. Moreover, if guards are used to control how a process executes, then processes which are defined with such Horn Clause guards can have a control component specified by Horn Clause programs.

Curly braces are added to *HCSP* so that the following is true:

$$\{R_1 \Rightarrow Q_1, \ldots, R_k \Rightarrow Q_k\} = cond(R_1 \Rightarrow Q_1, \ldots, R_k \Rightarrow Q_k)$$

HCSP programs will thus be specificed by giving a set of *HCSP* clauses. In addition a PROLOG syntax will be adopted for the Horn Clause component of *HCSP*, even though our "official" syntax is the Fitting version. Since they're equivalent there is no problem.

It may also be desirable to allow P to interact with other processes as part of its repertoire of actions. This will allow it to be influenced by external as well as internal events. If *in.c* and *out.c* are channels leading to some external process, then we can allow *ACT* to have a clause like the following:

$$ACT(<c,m>,Z) = (in.c\ !m \rightarrow out.c\ ?x \rightarrow P(x,Z))$$

Thus, P may send a message m on $in.c$ and act upon the response. If we restrict Z to being a finite element, then there are only a countable number of possible state. Thus Z itself may be part of a message and we could have that following expression:

$$ACT(c,Z) = (in.c\,!Z \rightarrow out.c\,?Z_1 \rightarrow P(Z_1))$$

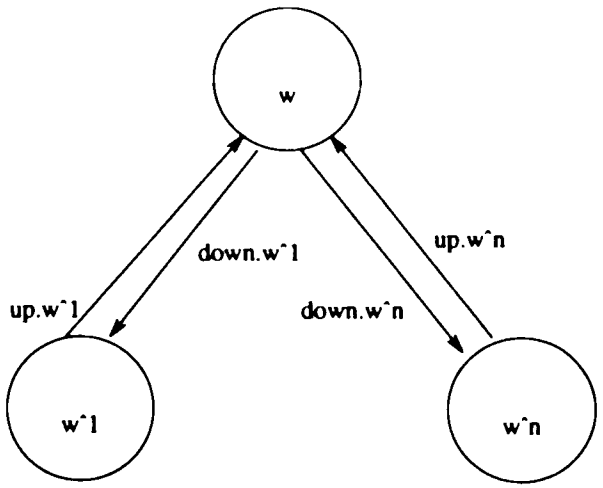
By doing this we permit Z_1 to depend on an arbitrary amount of information regarding Z. The purpose of events within the CSP formalism is to focus on those changes which do not *need* to be broken down further even if they *may* be broken down further. It is then justifiable to send any finite amount of information over channels so long as all we are concerned about is how the information contained in the message is treated as a whole. This implies that we can have a local decision procedure which is part of P and only has access to information contained in Z and a global decision procedure which may have access to more than just Z.

We can now provide a process description of a node in a search tree. It is a process with two parameters. The first parameter is the location of the process within the process tree. The second is the data contained in that node and is thus an element of a domain. Moreover, it is a finite element. There is an input and an output channel connecting each node to each of its descendents and to its parent (except for the root node). We need some uniform way of naming these channels. Each will be of the form:

direction.location

where *direction* is either *up* or *down* and *locations* is a location within the process

tree. These channels only connect parents and children and will follow the convention that *location* is the location of the child. *up* or *down* will be determined by noting where the channel provides input. The following diagram shows these conventions.



The following notation is introduced, where *w* is the location of the process that the notation occurs in:

$$to\_p(m) = up.w\,!m$$

$$to\_c(k,m) = down.w\hat{}k\,!m$$

$$from\_p(x) = down.w\,?x$$

$$from\_c(k,x) = up.w\hat{}k\,?x$$

So if P is at location w and Q is at location $w\hat{}n$ where:

$$P = (to\_c(n,m) \rightarrow P_1) = (down.w\hat{}n\,!m \rightarrow P_1)$$

$$Q = (from\_p(x) \rightarrow Q_1(x)) = (down.w\hat{}n\,?x \rightarrow Q_1(x))$$

then:

$$trans\,(w\_sync\,(P,Q), <down.w\hat{}n.m>, w\_sync\,(P_1, Q_1(m)))$$

Messages between nodes in a tree are sent asynchronously and the following process is used to express this:

$$asynch\_send\,(C,X,P) = par\,((C\,!X \rightarrow STOP),P)$$

In addition, there is a control process which has an input and an output channel for every node in the tree. These channels are denoted *in.c_chan.w* and *out.c_chan.w*. Messages will be sent synchronously between the control process and nodes in the tree. However, a node process always suspends and waits for a reply from the control process. In this way the control process does not have to wait for messages. If an

input message is there it may take it or choose not to. If it wants to output a message, then it does not have to wait because the process it wishes to send to is always waiting.

There are two predicates, *local* and *global*, which correspond to the local and global decision procedures, respectively.

Finally, there is process called *ANSWER* that is used to collect the answers generated by the tree. It has a single input channel *answer* that the root of the tree uses for an output channel.

$$SOLVE(G) = w\_sync(ANSWER(\varnothing), NODE(\Lambda, G), CONTROL(\textbf{bottom}))$$

$$ANSWER(S) = (answer\ ?x \rightarrow ANSWER(S \cup \{x\}))$$

NODE is defined by the following *HCSP* specification:

1. *up(X)* is an event that causes a node to send a message *X* from its output buffer up the tree. The root process outputs an answer on the channel *answer*.

$$local(up(X), \Lambda, Z) =>$$

$$asynch\_send(answer, X, NODE(\Lambda, delete(output, X, Z)))$$

$$local(up(X), W\hat{}N, Z) =>$$

$$asynch\_send(W, X, NODE(W, delete(output, X, Z)))$$

2. *down* (N,X) is an event that causes a node to send a message X to its *Nth* child.

$$local\,(down\,(N,X),W,Z) =>$$

$$asynch\_send\,(W\hat{}N,X,NODE\,(W,delete\,(output,X,Z)))$$

3. *read* (X) causes a transfer of a message X from the input buffer to the *data* area.

$$local\,(read\,(X),W,Z)) =>$$

$$NODE\,(W,add\,(data,X,\,(delete\,(input,X,Z))))$$

4. *write* (X) is analogous to *read* (X).

$$local\,(write\,(X),W,Z) =>$$

$$NODE\,(W,add\,(output,X,\,(delete\,(data,X,Z))))$$

5. *spawn* (N,X) causes a subnode X to be generated and a new process started started as the *Nth* child.

$$local\,(spawn\,(N,X),W,Z) =>$$

$$w\_sync\,(NODE\,(W,attach\,(N,Z,X)),NODE\,(W\hat{}N,new\_state\,(X)))$$

6. *control* cause the node to suspend operation while it awaits the result of a query of the *CONTROL* process.

$$local\,(control,W,Z) =>$$

$$(in.c\_chan.W!Z \rightarrow out.c\_chan.W?Z_1 \rightarrow NODE(W,Z_1))$$

7. *stop* simply causes a node to halt. We could alternately cause it to become a process that always responded to any message by saying it was not operable. This would not get us anything useful.

$$local(stop,W,Z) =>$$

$$STOP$$

CONTROL will be defined by *HCSP* definitions of the following form:

$$global(X,Z) => CONTROL(E(X,Z))$$

where X is an event, Z is the control state and $E(X,Z)$ may involve one or both of W and X. *read* and *write* will be among $events(CONTROL(Z))$ and will have definitions analogous to that of *NODE*. In addition the following clause for *global* will define how *NODE* and *CONTROL* communicates:

```
global(in.c_chan.W.M,Z) :-
        location(W),
        c_message(M),
        c_state(Z).
```

where *location*, *c_message*, and *c_state* are predicates the are true only if there arguments are locations, control messages, or control states, respectively. Also, we

require that *global*(*X*, **bottom**) is always *false*. Thus, at the start all *CONTROL*(**bottom**) can do is wait for its first input message.

6. Rules and Logic Programs

Reductions can now be expressed as processes in CSP. This gives a domain-theoretic characterization of the process of reducing objects to normal forms. The goal of this chapter is to show how to map from logic programs to reductions. This gives a mapping from logic programs to processes via reductions.

Let **L** be a logic programming language, **P** a logic program **L**, and **R** be the reduction system that **P** is mapped to. The literals of **L** will be mapped to atoms in **R**. Conjunctions correspond to non-atomic objects. Thus, after applying a decomposition operator, conjuncts map to sets of atoms. Disjunctions map to sets of objects. After decomposition, these become finite sets of finite sets of atoms. Each clause will be understood as contributing to the reduction function on the decomposition space of **R**. Thus, clauses will be mapped to relations from atoms to sets of atoms. The union of all such relations from each clause included in a logic program gives a reduction relation only for those generalized objects that are explicitly mentioned in the program.

6.1 Reductions for Logic Programs

A rule system is a finite, syntactic representation of a reduction system. Each rule determines a possibly infinite set of reductions pairs. The union of all such sets for all rules in a rule system gives the reduction relation. We will not be too specific about the general syntax of a rule system since there are many possible variations, such as production systems, grammars, Markov algorithms, and logic programs. In general, each rule has a right side and a left side joined by some connective. Each side is

further composed of sequences of objects which will generically be referred to as *items*. In the case of logic programs, the items are the literals.

The following semantic functions map from a logic programming language to objects of a reduction system:

DEF $atom : LITERALS \rightarrow ATOMS(S)$,

where *atom* is a given function. If $ATOMS(S) = LITERALS$ then the mapping is to the Herbrand Base.

DEF $conj : CONJUNCTS \rightarrow Finite(ATOMS(S))$,

where $conj(L_1, \ldots, L_k) = combine(\{atom(L_1), \ldots, atom(L_k)\})$. For the moment think of $ATOMS(S)$ as the Herbrand Base and *combine* as the identity function.

DEF $disj : DISJUNCTS \rightarrow Finite(Finite(ATOMS))$,

where $disj(C_1; \cdots; C_n) = \{conj(L_1), \ldots, conj(L_n)\}$

DEF $clause : CLAUSE \rightarrow ATOM(S) \times Finite(ATOMS(S))$,

where $clause(Q(\mathbf{x}) :- \phi(\mathbf{x})) = \{atom(Q(\mathbf{x}))\} \times conj(\phi(\mathbf{x}))$

DEF $prog : PROG \rightarrow Finite(ATOMS(S) \times Finite(ATOMS(S))$,

where $prog(P) = \{clause(X) : X \in clause\_set(P)\}$

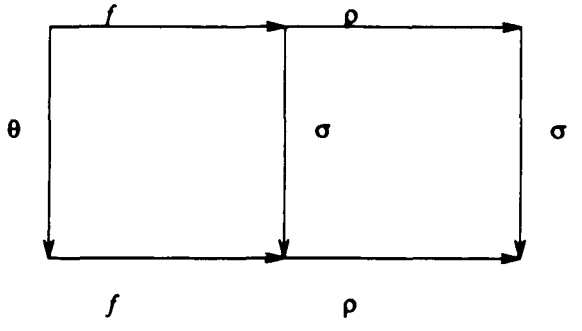
In addition to the syntax of programs, there is usually some notion of pattern

matching, which involves substitutions. Such substitutions map items to items. This can be naturally extended to include mappings from item sequences to item sequences, rules to rules, and sets of rules to sets of rules. Using these substitutions one can generate, from a rule, a set of derived rules. The function *map* maps substitutions to transformations in *SPEC*. It is defined by:

DEF $map: \Theta \rightarrow SPEC$ such that *map* is a bijection and if f is one of *atom*, *conj*, *disj*, *clause*, or *prog*, and $map(\theta) = \sigma$, then:

$$(*) \quad f * \theta = \sigma * f$$

This can be expressed, along with the previous postulates for the commutativity of σ and ρ, by postulating that the following diagram commute:



prog expresses how to map from programs to specific generalized states. This means that one cannot immediately use *prog* to define the resolution relation; it must be modified to include instances of literals that appear in rules. The following

definitions are useful for this:

DEF $inst : X \rightarrow power(X)$

where X is a literal, conjunctions, disjunction, or clause, and $inst(X)$ is all the substitution instances of X. It is defined by:

$$inst(X) = \{X\theta : \theta \in \Theta\}$$

DEF $spec : X \rightarrow power(X)$

spec is analogous to *inst*, except it applies to the set S rather than a set of syntactic objects. It is defined by:

$$spec(X) = \{\sigma(X) : \sigma \in SPEC\}$$

where $X \in S$ or $X \in S^2$.

Because of (*) we can derive from this the following:

$$spec * prog = prog * inst$$

Let $f = spec * prog = prog * inst$. One can think of f as being derived by first expanding a finite logic program into an infinite one and then mapping this infinite program to a reduction system. Alternatively, one can view it as a mapping from a finite logic program to a finite reduction system, which is then mapped to an infinite reduction system. Inspecting the types of the semantic functions shows that this is a reduction system defined in a decomposition space. Moreover, this reductions system

only applies to singleton sets of atoms; any non-singleton set is a normal form. This is rectified by extending this system to include non-singletons in the following way:

DEF $\nu: PROG \rightarrow Power(Finite(ATOMS(S)) \times Finite(ATOMS(S)))$

where

$\nu(\mathbf{P}) = \{(Z \cup x, Z \cup y) : Z \in Finite(Atoms(S))$ **and** $(x,y) \in (spec * prog)(\mathbf{P})\}$

We now have a reduction system defined on $Power(S)$. This reduction relation is given by $\nu(\mathbf{P})$. In order to use the techniques from the previous sections it is necessary to have a confluent reduction system, which $\nu(\mathbf{P})$ may not be. In the next section we will provide an alternative definition for S and an alternative definition for ν.

There are three types of rules that we are interested in. The reduction system $\nu(\mathbf{P})$ describes the *Logic* part of Kowalski's equation. In addition there will be rules that describe the *Control* part through the predicates *local* and *global* described in the previous chapter. They will also be given by Horn Clauses but the domain to which they apply will not necessarily be the Herbrand Universe. The domain will be that of continuations for local control and whatever domain is needed to record and manipulate control information.

6.2 Continuations for Logic Processes

When a logic program is activated it is given a goal G and expected to produce a correct answer substitution θ. This is defined so that:

$$correct(\theta, G, \mathbf{P}) \equiv (\mathbf{A}X)(X \in [G\theta] => eval_{\mathbf{P}}(X) = true)$$

That is, given a goal, an implementation produces a substitution, all of whose instances are true. Thus, one may choose to view a logic program as a computable relation between formulas and substitutions. We must say *relation* rather than *function* since θ need not be unique. This is why logic programs are nondeterministic.

The view that we will adopt here is a variant of this idea. In reviewing the PROLOG implementation, it can be seen that the essential operations are those involving managing the environment. This environment records the current value of the variables in the goal's derivation. A model of logic programming implementation should have as a prominent feature some object which represents this. In fact it is already there in the notion of substitutions.

When queried, a PROLOG implementation transforms an initial stack into a final stack which has an environment that records a correct answer substitution for the query. It is possible to request that this query be re-satisfied, *i.e.* a different correct answer substitution be computed. Thus, an initial query determines an initial stack which may be transformed into one or more final stacks. Its formal analogue is a relation between an initial substitution and a stream of final substitutions. Different, but equally correct, implementations may yield different orderings of the same substitutions. It is substitutions which are the key elements manipulated within continuations. It is the reduction of queries, represented by substitutions, that is to be made confluent.

A continuation is meant to record enough information so that we can implement a confluent relation. It must be able to record and pursue new branch points and to proceed from old ones. This will be done by maintaining two kinds of information. The first will consist of *all* the subtasks that have been generated for a given node. The second will contain whatever extra information is deemed useful to manage the first kind of information. For example, in order to identify which is the next available descendent process, one can store an index. Also, if one wants to overlay a data structure on the subtasks, such as a directed graph, then one can store the edges in this database.

The first step in working out the details of this is seeing how we move from relations between formulas and substitutions to an equivalent notion of relations between substitutions. The relation will be that of input substitution to output substitution. Let a *predicate skeleton* be an atom that contains only non-repeated variables and no constants or function symbols. *PSkeleton* denotes the set of all predicate skeletons. The predicate skeletons for a given relation symbol are all equivalent modulo renaming of variables. Clearly, any atom can be represented as a predicate skeleton and a substitution. A conjunction of predicate skeletons which share no variables will be called simply a *skeleton*. The set of all skeletons is denoted *Skeleton*. Thus, given a relation symbol, we can focus on substitutions alone. Note that the syntax we have given for logic programs facilitates using these ideas since the heads of clauses are predicate skeletons.

We now begin the task of defining the structure *Node* and the control predicates *local* and *global*. First, we provide the details about continuations and what the state

of a node consists of. The basic object that we will manipulate will include a skeleton and a substitution. It is called a *frame*. Additional information that is stored along with them concerns the status of their solution, such as whether it has been solved, whether it has failed, whether an answer is pending or whether no resources have been allocated for it. Moreover, if it is pending, which descendent is working on an answer.

Consider non-atomic frames. As we provide partial solutions for complex frames, we must update any information recording the status of solutions. What we know is that the solution obtained must be applicable to all members of a decomposed goal. This goes for both propositional reductions and data generation reductions. Thus, partial solutions must distribute among frames just as substitutions. A *cell* records this informations for literals. A frame combines this information into a structure which records this information both for conjunctive queries and for the individual literals that make it up.

DEF $Phase = \{waiting, solved, failed, pending, reported\}$

DEF $Cell = PSkeleton \times \Theta \times Phase \times RED \times (Single(\mathbb{N}) \cup \varnothing)$

DEF $Frames = Finite(Cell) \times \Theta \times Phase \times RED \times Finite(\mathbb{N})$

DEF $Atoms = Single(Cells) \times \Theta \times Phase \times RED \times Single(\mathbb{N})$

The function *make_cell* turns a literal into a cell. The function *make_atom* turn a cell into an atomic frame. The function *combine* turns a set of atomic frames into a

single frame. The function *decompose* turns a single frame into a set of atomic frames.

Suppose:

$$c_i = (s_i, \theta_i, p_i, \rho_i, n_i)$$

$$C = (\{c_1, \ldots, c_k\})$$

$$a_i = (\{c_i\}, \theta_i, p_i, \rho_i, n_i)$$

$$Z = \{a_1, \ldots, a_k\}.$$

Then:

DEF $make\_cell(P(\mathbf{t})) = (P(\mathbf{x}), \theta, waiting, id, \varnothing)$

where no variable in x occurs in t and $\theta(P(\mathbf{x}))=P(\mathbf{t})$

DEF $make\_atom(c_i) = a_i$

DEF $combine(Z) = (C, \Sigma\theta_i, \Sigma p_i, \Sigma\rho_i, \bigcup n_i\})$

provided that all the sums are defined. Otherwise,

$combine(Z) = (C, \mathbf{bottom}, failed, \mathbf{bottom}, \bigcup n_i)\}$

DEF $decompose((C, \theta, p, \rho, N)) = \{a : (\mathbf{E}c)(c \in C \textbf{ and } a = make\_atom(c))\}$

The function *combine* is defined so that it unifies substitutions, takes upper bounds

of reductions (according to the transformation ordering), combines the status of each of its cells into a status for the frame as a whole, and takes the union of all of the descendents. Since each cell contains the information necessary to form an atomic frame, the function *decompose*(F) merely strips out the cells from F and constructs the appropriate atomic frames.

A status for non-atomic frames is determined as follows:

1. If one of its components is *waiting* and none of them have *failed* then it is *waiting*. This indicates that it can spawn descendent.
2. If one of its components has *failed* then it has *failed*.
3. If all have been *solved* then it is *solved*.
4. If all have been *reported* then it is *reported*.
5. Otherwise, it is *pending*. If a frame is *pending*, then it cannot spawn a descendent.

The function *atom* which maps from literals to objects in our intended reduction system is defined by:

$$atom = make\_atom * make\_cell$$

The definition of *combine* then makes possible the definition of *conj*. The definitions of *disj*, *clause*, and *prog* follow immediately. Applying the function *conj* to a conjunction results in a frame of the following form:

$$F = (C, \theta, waiting, id, \varnothing)$$

where C is the set of its component cells. Subprocesses will be spawned to solve component cells for F. When this happens a new frame is created to record this fact. Upon the receipt of a message from the subprocess spawned to solve a component cell we update the information in the appropriate cell. This also updates the information in the frame which contains this cell. The status of this cell will then become either *failed* or *solved* and will thus not be available to spawn new processes. One can define a relation on cells that expresses these possible transitions and extend it naturally to atomic frames sets of atomic frames, and finally to non-atomic frames. This relation will be called *succ*. In the next section we will consider its definition.

The reason we are considering continuations is that we want them to allow us to be able to record all choice points that are available at any given stage in the solution of a goal. Frames represent such points. Sets of frames then represent sets of choice points. So long as we do not discard any frames we do not discard any choice points. Thus, we will store at each node the complete set of frames that have been generated. This implies that as a node evolves its set of associated frames can only become larger and more inclusive; no information is lost. If we extend *succ* to sets of frames then it does not loop. That is, the reflexive closure of *succ* is a partial order. This was our other reason for introducing continuations. This leads to the following definition for the set of continuations, denoted *Cont*:

DEF $Cont = Finite(Frame)$

The state that is maintained by a node will also contain in input set, an output set, and the auxiliary database mentioned above. The I/O sets will contain messages. The auxiliary database will be a finite set containing objects in the form of ground terms. So the next available descendent will be stored as $next\_child(k)$, where $k \in \mathbf{N}$. Let *Data* denote the database. Then states are defined by:

DEF $States = Finite(Messages) \times Finite(Messages) \times Data \times Cont$

Thus, the components of a state are the input set, the output set, the auxiliary database, and a continuation, respectively. A node is then a process which has a location and a state as a parameter. That is:

$$Node : Loc \times States \rightarrow PROC$$

6.3 The SLD Relation

In this section we extend the SLD relation on conjucts to a relation on frames. Each conjunct $C(\mathbf{x})$ can be split into an equational part, $E(\mathbf{x})$, and a non-equational part $P(\mathbf{x})$. This information can be used to construct two relations R_σ and R_ρ. Consider the clause:

$$Q(\mathbf{x}) :- E(\mathbf{x}), P(\mathbf{x})$$

The role of $E(\mathbf{x})$ is to provide a substitution based on unification. Therefore, the following should hold if $E(\mathbf{x})$ is not empty:

$$R_\sigma(<Q(\mathbf{x}), \theta, p, \rho, n>, <Q(\mathbf{x}), unify(\theta(E(\mathbf{x}))), p, \rho, n>)$$

and if $E(\mathbf{x})$ is empty:

$$R_{\sigma}(<Q(\mathbf{x}),\theta,p,\rho,n>,<Q(\mathbf{x}),\theta,p,\rho,n>)$$

i.e. no substitution needs to be generated.

The role of R_{ρ} is to characterize propositional steps. Consider the following rule of inference known as *modus tollens*:

$$\frac{P \Rightarrow Q,\ \neg Q}{\neg P}$$

This can be extended to:

$$\frac{P \Rightarrow Q,\ \neg(Q \textbf{ and } R)}{\neg(P \textbf{ and } R)}$$

Using :– instead of => yields the rule:

$$\frac{Q{:-}P,\ \neg(Q \textbf{ and } R)}{\neg(P \textbf{ and } R)}$$

This replacement within a negated conjunction is what is meant as a *propositional step*. Indeed one may describe a step of SLD resolution as applying the rule:

$$\frac{Q{:-}P,\ \neg(Q' \textbf{ and } R),\ \theta(Q)=\theta(Q')}{\neg(\theta(P \textbf{ and } R))}$$

We are breaking this single rule into two parts. $Q(\mathbf{x}){:-}E(\mathbf{x}),P(\mathbf{x})$ has some $\rho \in RED$ associated with it. ρ should be such that

$$\rho(Q(\mathbf{t})) = \theta(P(\mathbf{x}))$$

if and only if $\theta = unify(E(\mathbf{t}))$ is defined. Otherwise, $\rho(X)=X$. Thus, ρ generates a new goal if unification is successful. This allows us to define R_ρ.

If $P(\mathbf{x})$ is not empty then:

$$R_\rho(\langle Q(\mathbf{x}),\theta,p,\rho',n\rangle,\langle Q(\mathbf{x}),\theta,p,\rho*\rho',n\rangle)$$

only if $\theta(E(\mathbf{x}))$ is unifiable and where ρ is the transformation associated with that rule.

If $P(\mathbf{x})$ is empty:

$$R_\rho(\langle Q(\mathbf{x}),\theta,p,\rho,n\rangle,\langle Q(\mathbf{x}),\theta,solved,\rho_\square*\rho,n\rangle)$$

where $\rho_{\square}$ is a constant function whose value is $\square$, *i.e.* we have arrived at a contradiction, which is the goal of our search. Thus,

$$GOALS = \{<Q, \theta, solved, \rho_{\square} * \rho, \{n\}> : \theta \in \Theta \textbf{ and } \rho \in RED \textbf{ and } n \in \mathbf{N}\}$$

The relation *sld* is defined by:

DEF $sld = R_{\rho} * R_{\sigma}$

sld, R_{σ}, and R_{ρ}, are defined in terms of cells. We describe the extension to frames using *combine*. Let F be an atomic frame where $F = make\_atom(c)$. Then:

$$R(F,F') \equiv R(c,c') \textbf{ and } F' = make\_atom(c')$$

where R is one of *sld*, R_{σ}, or R_{ρ}. Let F now be a possibly non-atomic frame, where $F = combine(Z)$. Then:

$$R(F,F') \equiv (\mathbf{E}a)(a \in Z \textbf{ and } R(a,a') \textbf{ and } F' = combine((Z - \{a\}) \cup \{a'\}))$$

where R is again one of *sld*, R_{σ}, or R_{ρ}.

It is now possible to re-define the function ν. Let ν_{σ} be the function that maps form $\mathbf{P}$ to R_{σ} and let ν_{ρ} do the same for R_{ρ}. Then we define ν such that:

$$\nu(\mathbf{P}) = \nu_{\sigma}(\mathbf{P}) * \nu_{\rho}(\mathbf{P})$$

That is, ν yields *sld*. Furthermore, we can define the function *map* such that:

$$map(\theta)((s, \theta')) = (s, \theta * \theta')$$

6.4 Specifying Control

The predicate *local* has the following simple definition:

```
local(Action,Location,State) :-
        components(Input,Output,Data,Frames,State),
        ready(Action,Location,Input,Output,Data,Frames),
        should_do(Action,Location,Input,Output,Data,Frames).
```

components is a given relation that satisfies:

components (*Input*,*Output*,*Data*,*Frames*, (*Input*,*Output*,*Data*,*Frames*))

The predicate *ready* is true iff its first argument is an action which is ready to be performed in the state that corresponds to its second argument. For example, *read*(*X*) is an action that can be performed only if the input set is not empty. *should_do* applies further checks. For example, it may be possible to read in a given state, but it may be specified that reading is only done when there are no messages to write. Thus, *ready* may succeed but *should_do* may fail. *ready* specifies what may be done while *should_do* specifies what should be done.

ready is defined by:

```
ready(read(X),Loc,In,Out,Data,Frames) :-
        member(X,In).

ready(write(<Loc,<solved,Sigma,Rho>>),Loc^N,In,Out,Data,Frames) :-
```

```
        member(F,Frames),
        status(solved,F),
        /* psi is a given relation that examines F */
        psi(F,Sigma,Rho).

ready(write(<Loc,failed>),Loc^N,In,Out,Data,Frames) :-
        /* all_failed is a given search */
        all_failed(Frames).

ready(up(<Loc,X>),Loc^N,In,Out,Data,Frames) :-
        member(<Loc,X>,Out).

ready(down(N,X),Loc,In,Out,Data,Frames) :-
        member(<Loc^N,X>,Out).

        /* Spawn a new process only if there is a frame with an atom waiting
and
        it does not create a frame that is already in Frames */
        ready(spawn(K,X),Loc,In,Out,Data,Frames) :-
        member(F,Frames),
        decompose(F,Z),
        member(A,Z),
        status(waiting,A),
        next_child(K,Data),
        sld(A,X),
```

```
        replace(A,X,Z,Z'),          /*Replace A by X in Z to obtain
Z'*/
        combine(Z',F'),
        ¬member(F',Frames).

ready(stop,Loc,In,Out,Data,Frames) :-
        ¬member(X,In),
        ¬member(X,Out),
        ¬member(X,Data),
        ¬member(X,Frames).

ready(control,Loc,In,Out,Data,Frames) :-
        true.
```

The definition of *ready* is standard and would not be part of a users program. The definition of *should_do* should be provided by the user, although a default one could certainly be provided. The last clause says that a node is always to consult the control process. It is the predicate *should_do* that determines when this happens.

Parts of *local*'s definition could be by default. For example, a user might want to determine how to select subgoals from within a node but no care to worry about how messages should be handled. On the other hand, another user might want to maximize the handling of messages and to avoid spawning any subprocesses. The semantics we have provided can handle any version of *should_do* that the user provides.

We now consider the functions that perform the actions on states.

1. *delete* remove an object from a designated area.

   - $delete\,(output,X,(I,O,D,K)) = (I,O-\{X\},D,K)$
   - $delete\,(input,X,(I,O,D,K)) = (I-\{X\},O,D,K)$

In addition, if one deletes from the input a message from the parent of the form:

$$<W,kill>$$

this indicates that the parent wishes to stop the receiving process. As result the receiving process should send the same message to each of its descendents and change each part of the state to $\varnothing$. The only action that is then possible, according to the definition of *ready*, is to stop. If a *failed* message is deleted from a descendent, then the frame that spawned it is found and updated as failed. If a success message is deleted, then the appropriate frame is updated.

2. *add* adds an object to the designated area.

   - $add\,(output,X,(I,O,D,K)) = (I,O\cup\{X\},D,K)$
   - $add\,(input,X,(I,O,D,K)) = (I\cup\{X\},O,D,K)$
   - $add\,(data,solve\,(N,X),(I,O,D,K)) = (I,O,D,update\,(X,K))$

If a *failed* message is added to the output, then there is nothing for this node to do. Thus, the state should be made empty. Also, all active descendents should be killed. If a success message is added to the output buffer then we mark the

appropriate frame as *reported*.

3. *attach*(*N*,*State*,*X*) adds *X* as a new frame to the state. It has *N* as the descendent designated to solve it. The frame responsible for generating it is marked as pending and its descendent is *N*.

4. The function *new_state* initializes a new state from a frame passed down from a parent.

The definition of *global* leads to interactions between a node and global control information. We will take the global data structure to be a data base of fact, represented as a set of assertions about the states of the nodes in the tree of processes. Each such fact should include at least the location of the node to which it refers.

For example, depth-first search could be specified by requiring that each node process consult the control process before it begins. If any of its older siblings or their descendents are still active then this process waits. The status of active processes can be kept in the global data base. The process can be suspended by simply delaying to respond to it until it is appropriate. This is possible because when a node process consults the control process it voluntarily suspends until it receives a message from the control process.

The same strategy can be applied to specify a breadth-first search. Moreover, one can use any kind of prioritizing mechanism so long as we store enough information in the global data base. This priority can also be dynamic, that is changing every time new information is added. Priority then becomes a relation between the various goals

ready to be worked on, rather than a static property of goals. This allows for data driven control flows, such as is used in Concurrent Prolog. In the case of Concurrent Prolog, the dynamic information is determined by looking at which goals have variables that have just had read-only variable become bound.

Thus, we have seen that the process *SOLVE*, defined in the last chapter, provides a basis for the semantics of parallel logic programs. The function v maps from logic programs to relations. These relations, when incorporated as part of the process *SOLVE*, drive the evolution of nodes in a process tree. Further, given definitions of the predicates *local* (via the predicate *should_do*) and *global* it is possible to influence the behavior of these nodes. The *Logic* part of Kowalski's equation is thus a standard logic program over the Herbrand Universe. The *Control* component is a logic program over the data structures defined to manage the execution of *Logic*. Thus, we have controlled deduction where the control is itself specified by a deductive mechanism.

7. Appendix 1 - Proofs

7.1 Proofs for Chapter 3

We need to prove the following:

Basis Theorem for Proc:

$P \in PROC$ *iff* there is some $Z \subseteq A$ such that Z is independent and $P = CHAOS - \uparrow Z$

Proof: Immediate from the First and Second Main Lemmas, proved below.

It will be useful to state the original axioms in terms of $\bar{P}$ rather than P.

$\bar{P}$:

$$\bar{P}_1 : (s,X) \in \bar{P} \Rightarrow is\_finite(X)$$

$$\bar{P}_2 : (\langle\rangle, \varnothing) \neg\in \bar{P}$$

$$\bar{P}_3 : (s, \varnothing) \in \bar{P} \text{ and } s \leq t \Rightarrow (t, \varnothing) \in \bar{P}$$

$$\bar{P}_4 : (s,X) \in \bar{P} \text{ and } X \subseteq Y \Rightarrow (s,Y) \in \bar{P}$$

$$\bar{P}_5 : (s, X \cup \{c\}) \in \bar{P} \text{ and } (s^\frown\langle c \rangle, \varnothing) \in \bar{P} \Rightarrow (s,X) \in \bar{P}$$

Equivalence of P and $\bar{P}$

Proof: Clearly, axiom $\bar{P}_1$ is taken care of by since all P and $\bar{P}$ are subsets of $CHAOS$, and hence $A^* \times finite(A)$. Moreover, *independence* will be defined so as to exclude $(\langle\rangle, \varnothing)$ from any independent set. $(\langle\rangle, \varnothing)$ will also be defined as the bottom element in the poset $\langle CHAOS, \leq \rangle$. Thus, if x is independent $(\langle\rangle, \varnothing) \neg\in x$ and hence

$(\langle\rangle,\varnothing)\neg\in \uparrow x$. This takes care of axiom $\bar{P}_2$. **QED**

Poset Lemma

$\langle CHAOS, \leq\rangle$ is a poset.

Proof: Recall that the partial order defined on *CHAOS* is defined by:

$$(s,X)\leq(t,Y) \equiv [(s=t \textbf{ and } X\subseteq Y) \textbf{ or } (s\leq t \textbf{ and } X=\varnothing)]$$

We must show reflexivity, anti-symmetry, and transitivity.

A *Reflexivity:* Clearly, by the first disjunct in the definition of $\leq$, $(s,X)\leq(s,X)$.

B *Anti-symmetry:* Suppose $(s,X)\leq(t,Y)$ **and** $(t,Y)\leq(s,X)$. There are four possibilities to consider:

1. $(s,X)\leq(t,Y)$ because $s=t$ **and** $X\subseteq Y$ *and* $(t,Y)\leq(s,X)$ because $t=s$ **and** $Y\subseteq X$. Clearly, this implies $X=Y$ and so $(s,X)=(t,Y)$
2. $(s,X)\leq(t,Y)$ because $s=t$ **and** $X\subseteq Y$ *and* $(t,Y)\leq(s,X)$ because $t\leq s$ **and** $Y=\varnothing$. Now, since $X\subseteq Y$ and $Y=\varnothing$, $X=\varnothing$. Therefore, $X=Y$ and $s=t$.
3. $s\leq t$ **and** $X=\varnothing$ *and* $t=s$ **and** $Y\subseteq X$. This is similar to case 2.
4. $s\leq t$ **and** $X=\varnothing$ *and* $t\leq s$ **and** $Y\subseteq X$. This is similar to case 1.

C *Transitivity:* Suppose $(s,X)\leq(t,Y)$ **and** $(t,Y)\leq(u,Z)$. Again there are four cases and the the following implications are readily seen to be valid.

1. $[(s=t \textbf{ and } X\subseteq Y) \textbf{ and } (t=u \textbf{ and } Y\subseteq Z)] \Rightarrow [s=u \textbf{ and } X\subseteq Z]$

2. $[(s=t \text{ and } X \subseteq Y) \text{ and } (t \leq u \text{ and } Y=\varnothing)] => [s \leq u \text{ and } X=\varnothing]$

3. $[(s \leq t \text{ and } X=\varnothing) \text{ and } (t=u \text{ and } Y \subseteq X)] => [s \leq u \text{ and } X=\varnothing]$

4. $[(s \leq t \text{ and } X=\varnothing) \text{ and } (t \leq u \text{ and } Y=\varnothing)] => [s \leq u \text{ and } X=\varnothing]$

QED

Using the definition of $\leq$, we can replace axioms 3 and 4 by the following single axiom:

$$\bar{P}_{3,4}: f \in \bar{P} \text{ and } f \leq f' => f' \in \bar{P}$$

$\bar{P}_{3,4}$ is equivalent to $\bar{P}_3$ and $\bar{P}_4$.

Proof: To see this, first note that $\bar{P}_3$ can be replaced by:

$$(s, \varnothing) \in \bar{P} \text{ and } s \leq t => (t, X) \in \bar{P}$$

This is easily seen by transitivity of => and axioms $\overline{P_3}$ and $\overline{P_4}$. Moreover, since $s \leq s$, $\overline{P_3}$ is a special case of this axiom.

Using simple facts about equality, we can thus rewrite axioms $\overline{P_3}$ and $\overline{P_4}$ as:

A: $(s,X) \in \bar{P} \text{ and } [s \leq t \text{ and } X=\varnothing] => (t,Y) \in \bar{P}$

B: $(s,X) \in \bar{P} \text{ and } [s=t \text{ and } X \subseteq Y] => (t,Y) \in \bar{P}$

Note that each of **A** and **B** has a disjunct from the definition of $\leq$ in its antecedent. Using the definition of $\leq$ and the following first-order theorems:

i) $(\mathbf{A}x)(P(x)\ \mathbf{and}\ Q(x)) \equiv (\mathbf{A}x)(P(x))\ \mathbf{and}\ (\mathbf{A}x)(Q(x))$

ii) $[(P\ \mathbf{and}\ Q_1 \Rightarrow R)\ \mathbf{and}\ (P\ \mathbf{and}\ Q_2 \Rightarrow R)] \equiv [P\ \mathbf{and}\ (Q_1\ \mathbf{or}\ Q_2) \Rightarrow R]$

we obtain:

$$\mathbf{A\ and\ B} \equiv (s,X)\in \bar{P}\ \mathbf{and}\ (s,X)\leq(t,Y) \Rightarrow (t,Y)\in \bar{P}$$

Obviously, this is equivalent to $\bar{P}_{3,4}$.

QED

First Main Lemma

If $\bar{P}$ satisfies the axioms $\bar{\mathbf{P}}$, then $\lfloor \bar{P} \rfloor$ is independent.

Proof: Let $M=\lfloor \bar{P} \rfloor$. By $\bar{P}_1$ and $\bar{P}_2$, $\bar{P}\subseteq CHAOS-\{(<>,\varnothing)\}$ clearly holds for $\bar{P}$ and hence for M, since $M\subseteq\bar{P}$. Moreover, by the definition of min, no two elements of M can satisfy $\mathbf{I}_1$. Now, suppose that $(s,X\cup\{c\}),(s^\frown<c>,\varnothing)\in M$. Then they are also elements of $\bar{P}$, and so, by $\bar{P}_5$, $(s,X)\in\bar{P}$. However, since $(s,X)<(s,X\cup\{c\})$, this contradicts the assumption that M is minimal. Therefore, since not both of (s,X) and $(s^\frown<c>,\varnothing)$ can be in M, M is independent.

QED

The Second Main Lemma is the converse of this, *viz.* if M is an independent set, then $M=\lfloor \bar{P} \rfloor$, for some $\bar{P}$ that satisfies axioms $\bar{\mathbf{P}}$. We make use of the following two lemmas in establishing this.

Lemma 1 Suppose:

1. $Z \subseteq CHAOS$
2. $(s^\frown\langle c\rangle, \varnothing) \neg\in Z$
3. $(s^\frown\langle c\rangle, \varnothing) \in \uparrow Z$

then for all $X \subseteq A$, $(s,X) \in \uparrow Z$.

Proof: Since $(s^\frown\langle c\rangle, \varnothing)$ is not in Z but is in $\uparrow Z$, then there must be some $f \in Z$ such that $f < (s^\frown\langle c\rangle, \varnothing)$. From the definition of $\leq$, f must be of the form $(t, \varnothing)$, where $t < s^\frown\langle c\rangle$. This implies $t \leq s$, which in turn implies that $(s, \varnothing) \in \uparrow Z$. Therefore, since for all $X \subseteq A$, $(s, \varnothing) \leq (s,X)$, we can conclude that $(s,X) \in \uparrow Z$.

QED

Lemma 2 Suppose:

1. Z is independent
2. $(s^\frown\langle c\rangle, \varnothing) \in Z$
3. $(s, X \cup \{c\}) \in \uparrow Z$

then: $(s,X) \in \uparrow Z$.

Proof: **Since Z is independent and $(s^\frown\langle c\rangle, \varnothing) \in Z$, then $(s, X \cup \{c\})$ cannot also be in Z. Since it is in $\uparrow Z$, then there must be some $(t,Y) \in Z$ such that $(t,Y) < (s, X \cup \{c\})$.** Now there are two ways in which this can happen. Either [$t < s$ **and** $Y = \varnothing$] or [$s = t$ **and** $Y \subset X \cup \{c\}$].

Case 1 If $t < s$ **and** $Y = \varnothing$, then $(t,Y) < (s,X)$, so (s,X) must also be in $\uparrow Z$.

Case 2 On the other hand, suppose $s=t$ **and** $Y \subset X \cup \{c\}$, *i.e.* $(t,Y)=(s,Y)$. Then it must be the case that $Y \subset X$ as well. Otherwise, Y must equal some set $X' \cup \{c\}$. But $(s,X'\{c\}) \in Z$ contradicts the assumptions that Z is independent *and* $(s^\frown\langle c\rangle,\varnothing) \in Z$. Therefore, since $Y \subset X$, $(t,Y) \leq (s,X)$, so $(t,Y) \in Z \Rightarrow (s,X) \in \uparrow Z$.

In either case, $(s,X) \in \uparrow Z$.

QED

We can now prove the following:

Second Main Lemma

If Z is an independent set, then $CHAOS - \uparrow Z$ satisfies axioms $\bar{P}$.

Proof:

Clearly, axiom $\bar{P}_1$ is satisfied since $(\langle\rangle,\varnothing)$ is not in any independent set and is minimal in $\langle CHAOS, \leq\rangle$, and hence not is also not in any filter generated by an independent set. It is also clear from the definition of filters that that axiom $\bar{P}_{3,4}$ is satisfied.

This leaves $\bar{P}_5$. Suppose that both $(s,X \cup \{c\})$ and $(s^\frown\langle c\rangle,\varnothing)$ are in $\uparrow Z$. We must show that (s,X) is also in $\uparrow Z$. Since Z is independent, then either $(s,X \cup \{c\})$, or $(s^\frown\langle c\rangle,\varnothing)$ is not in Z. If $(s^\frown\langle c\rangle,\varnothing) \neg\in Z$, then by Lemma 1, $(s,X) \in Z$. This true regardless of whether $(s,X \cup \{c\})$ is or isn't in $\uparrow Z$, so this also covers the case when both of $(s^\frown\langle c\rangle,\varnothing)$ and $(s,X \cup \{c\})$ are not in Z.

This only leaves the case where $(s^\frown\langle c\rangle,\varnothing) \in Z$ and $(s,X \cup \{c\}) \neg\in Z$. By lemma 2, this also implies that $(s,X) \in \uparrow Z$.

QED

7.2 Proofs for Chapter 4

Allowability of $(\mathbf{E}_i\chi)$

If $\chi \in Char(D,n+1)$ then $(\mathbf{E}_i\chi) \in Char(D,n)$.

Proof To aid readability, in the proof we assume that $i=1$ and omit the subscript for **E**. The modification for the general case is easy.

Suppose:

$$Z \in Direct(\mathbf{D}^n) \text{ and } lub(Z) = \langle d_1, \ldots, d_n \rangle.$$

We must show that for each $x \in Bool$:

$$(\mathbf{E}f)lub(Z) = x \text{ iff } lub((\mathbf{E}f)(Z)) = x$$

We show the case for $x=true$; the others are analogous.

For each $a \in \mathbf{D}$, let

$$Z(a) = \{\langle a, x_1, \ldots, x_n : \langle x_1, \ldots, x_n \rangle \in Z\}$$

Clearly, $Z(a)$ is directed and $lub(Z(a)) = \langle a, d_1, \ldots, d_n \rangle$.

Now by definition of **E** we have for some $a \in \mathbf{D}$:

$$(\mathbf{E}f)(d_1, \ldots, d_n) = true \text{ iff } f(a, d_1, \ldots, d_n) = true$$

From this we can deduce that for some $a \in \mathbf{D}$:

$$(\mathbf{E}f)(d_1, \ldots, d_n) = true \text{ iff } f\,(lub\,(Z\,(a))) = true$$

But since f is continuous we also have for some $a \in \mathbf{D}$:

$$(\mathbf{E}f)(d_1, \ldots, d_n) = true \text{ iff } lub\,(f\,((Z\,(a))) = true$$

By the definition of the ordering on *Bool*, for any $X \subseteq Bool$:

$$lub\,(X) = true \text{ iff } \neg(false \in X) \textbf{ and } true \in X$$

Hence, for each $\mathbf{a} \in Z\,(a)$:

$$f\,(\mathbf{a}) = true \textbf{ or } f\,(\mathbf{a}) = \textbf{bottom}$$

but for at least one $\mathbf{a} \in Z\,(a)$

$$f\,(\mathbf{a}) = true$$

This in turn implies the same for $lub\,((\mathbf{E}f)(Z))$. Similar reasoning thus allows us to establish that

$$lub\,(f\,(Z\,(a))) = lub\,(\mathbf{E}f)(Z))$$

QED

7.3 Proofs for Chapter 5

We now show that *Trees* (D) is in fact a domain under the pointwise ordering on functions. A suitable basis of finite elements is needed. A tree has *finite structure* if its size is finite and it has *finite content* if its range contains only finite elements of D.

A tree is *finite* if it has both a finite structure and finite content. We will show that *Trees* (D) is a *cpo* and that the set of finite trees in *Trees* (D) forms a countable basis for *Trees* (D)

Lemma 1 The set of finite trees of *Trees* (D) is countable.

Proof: We show that there is a one-to-one map of the set of finite trees into the set of natural numbers. It is not onto but this is not necessary to show countability. Since D is a domain it has a countable basis. Let d be a one-to-one function from this basis to **N**. The graph of t can be characterized by restricting ourselves to those elements $<w,t(w)>$ for which $t(w) \neq$ **bottom**. For trees with finite structure this set of such elements is always finite. We can now define our coding function as follows:

$$code(t) = set(\{pair(seq(w), d(t(w))) : node(w,t)\}$$

Since *set*, *pair*, *seq*, and d are one-to-one, then *code* is clearly one-to-one. Therefore, the set of finite trees in *Trees* (D) is countable.

Lemma 2: If **D** is a domain then *Trees* (**D**) is a *cpo* under the pointwise ordering. Moreover, if $Z = \{t_1, \ldots, t_k \cdots\}$ is any directed set of trees in *Trees* (D), then its least upper bound is the tree t defined by:

$$t(w) = lub(\{t_i(w) : t_i \in Z\}).$$

Proof: Clearly the function which satisfies, for all w, $t(w) =$ **bottom**$_D$, is the least element of *Trees* (**D**).

Let $Z = \{t_1, \ldots, t_k \cdots\}$ be an arbitrary directed set of trees in *Trees* (D), Then,

for each w, $\{t_1(w), \ldots, t_k(w) \cdots\}$ is a directed set. Let t be the tree defined by:

$$t(w) = lub(\{t_i(w) : t_i \in Z\}).$$

t is well-defined since D is a domain. Thus, all the necessary upper bounds do in fact exist. We must show that t is the least upper bound of Z. It is clear from the definition that it is an upper bound. Suppose that t' is also an upper bound which is strictly less than t. We can then derive the contradiction that t' is not an upper bound. If it is less than t then there must be some w such that $t'(w) \lhd t(w)$. But then from the definition of t, there must be a $t_i \in Z$ such that $t'(w) \lhd t_i(w)$. Therefore t' cannot be an upper bound. Thus, directed sets of trees have least upper bounds.

Lemma 3 The set of finite trees in $Trees(D)$ is a basis for $Trees(D)$.

Proof: We must show that for any $t \in Trees(D)$, there is a directed set of finite trees whose least upper bound is t.

Since D is a domain, then for every w there is a directed subset of finite elements of D, which we will denote Z_w, such that $lub(Z_w) = t(w)$. We sill further assume that **bottom** is always an element of Z_w. The set of finite elements that we want to construct is that set which has finite trees such that for each of its trees t', $t'(w) \in Z_w$.

Let $F(t) = \{t' : (\mathrm{A}w)(t'(w) \in Z_w)\}$. $F(t)$ is the set of trees with finite content that are less than or equal to t. Members of $F(t)$ do not necessarily have finite structure. This is remedied by the following definition:

$$approx(t) = \{t' \mid n : t' \in F(t)\}$$

Clearly, $approx(t)$ contains only finite trees. It is also directed. To see this, suppose $t_1, t_2 \in approx(t)$. Then for each w, $t_1(w), t_2(w) \in Z_w$. Since Z_w is directed, there is some $d \in Z_w$ which is an upper bound for $\{t_1(w), t_2(w)\}$. Let t_3 be the tree defined by such that for each w, $t_3(w)$ is the upper bound of $\{t_1(w), t_2(w)\}$. Clearly, t_3 is an upper bound of $\{t_1, t_2\}$ and in $approx(t)$. Therefore, $approx(t)$ is directed. Moreover, for each $t' \in approx(t)$, $t' \leq t$. What we must now show is that $lub(approx(t)) = t$. We do this by showing that for all w, $lub(approx(t))(w) = t(w)$. Then the result follows from Lemma 2.

To see this note that for any $t' \in approx(t)$ and for any w $t'(w) \in Z_w$i. Furthermore, for each $d \in Z_w$ there is a $t' \in approx(t)$ such that $t'(w) = d$. Therefore,

$$\{t'(w) : t' \in approx(t)\} = Z_w.$$

QED

We also have the following theorems regarding the continuity of subtree replacement:

THEOREM Let Z be a directed set of trees over **D**. Then

$$(lub(Z))[w \leftarrow t] = lub\{t_i[w \leftarrow t] : t_i \in Z\}$$

THEOREM Let Z be a directed set of trees over or elements of **D**. Then

$$t[w \leftarrow lub(Z)] = lub\{t[w \leftarrow x] : x \in Z\}$$

8. References

AV82 Apt, K.R. and vanEmden, M.H.
"Contributions to the theory of logic programming"
JACM 29,3 (July 1982), 841-862

BOKO82 Bowen,K. and Kowalski,R.
"Amalgamating language and metalanguage in logic programming"
in [CLTA82], 153-172

BHR84 Brookes, S.D.,Hoare,C.A.R.,Roscoe,A.W.
"A theory of communicating sequential processes"
JACM 31,3 (July 1984), 560-599

BUWE81 Bundy, A. and Welham, B.
"Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation"
Artificial Intelligence 16 (1981), 189-212

CAM84 Campbell, J.A., ed.
Implementations of PROLOG
Ellis Horwood Limited, West Sussex, 1984

CLGR81 Clark, K.L., and Gregory,S.
"A relational language for parallel programming"
Functional Programming and Computer Architectures, ACM, New York 1981,

CLGR86 Clark, K.L., and Gregory,S.

"PARLOG: parallel programming in logic"

TOPLAS 8,1 (January 1981), 1-49

CLTA82 Clark, K.L. and Tarnlund, S.-A.,eds.

Logic Programming

Academic Press, London, 1982

CM81 Clocksin,W.F. and Mellish, C.S.

Programming in PROLOG

Springer-Verlag, New York, 1981

CODD70 Codd, E.F.

"A relational model of data for large shared data banks"

CACM 13,6 (June 1970), 337-387

COL82 Colmerauer, A.

"PROLOG and infinite trees"

in [CLTA82], 231-252

COKI83 Conery, J.S. and Kibler, D.F.

"Parallel interpretation of Logic programs"

Proceedings of the Conference on Functional Programming Languages and Computer Architecture, ACM, (October 1983), 163-170

COKI85 Conery, J.S. and Kibler, D.F.

"AND parallelism and nondeterminism in logic programs"

New Generation Computing, 31 (1985), 43-70

CON87 Conery, J.S.

Parallel Execution of Logic Programs

Kluwer Academic Publishers, Norwell, 1987

CUCU85 Cuadrado,C.Y. and Cudrado, J.L.

"PROLOG goes to work"

BYTE 10,8 (August 1985), 151-158

DALE84 Davis, R. and Lenat, D.B.

Knowledge Based Systems in Artificial Intelligence

McGraw-Hill, New York, 1982

DA80a Davis, R.

"Meta-rules: Reasoning about control"

Artificial Intelligence 15,3 (December 1980), 179-222

DA80b Davis, R.

"Meta-rules: Content reference"

Artificial Intelligence 15,3 (March 1980), 223-240

DGLI86 Degroot, D. and Lindstrom G., eds.

Logic Programming, Functions, Relations, and Equations

Prentice-Hall, Englewood Cliffs, 1986

DIJ75 Dijkstra, E.W.

"Guarded commands, nondeterminacy,and the formal derivation of programs"

CACM 18,8 (August 1975), 453-457

DIJ76 Dijkstra, E.W.

A Discipline of Programming

Prentice-Hall, Englewood Cliffs, 1976

ES85 Eisenbach, S. and Sadler, C.

Declarative languages: an overview"

BYTE 10,8 (August 1985), 181-197

FIT85 Fitting, M.

"A Kripke-Kleene semantics for logic programs"

Journal of Logic Programming, 2:293-312 (1985)

FIT86 Fitting, M.

"Logic programming semantics using a compact data structure"

in Proceedings of the International Symposium of Methodologies for Intelligent Systems

Elsevier, 1986

GALA82 Gallaire, H. and Lassere, C.

"Metalevel Control for Logic Programs"

in Clark and Tarnlund, 173-185

GG85 Genesereth, M.R. and Ginsberg, M.L.

"Logic Programming"

CACM 28,9 (September 1985), 933-941

GEO82 Georgeff, M.P.

"Procedural control in production systems"

Artificial Intelligence 18,2 (March 1982), 175-202

GO79 Gordon, M.J.C.

The Denotational Description of Programming Languages

Springer-Verlag, New York, 1979

GR81 Gries, D.

The Science of Programming

Springer-Verlag, New York, 1981

GUES Guessarian, Irene

Algebraic Semantics

Springer-Verlag, New York, 1981

HO69 Hoare, C.A.R.

"An axiomatic basis for computer programming"

CACM 12,10 (Oct. 1969),576-580,583

HO78 Hoare, C.A.R.

"Communicating Sequential Processes"

CACM 21,8 (August 1978), 666-676

HO84 Hoare, C.A.R.

"Programming: Sorcery or Science"

IEEE Computer, (July 1984)

HO85 Hoare, C.A.R

Communicating Sequential Processes

Prentice-Hall International, London, 1985

HOG82 Hogger, C. J.

"Concurrent logic programming"

in [CLTA82], 199-213

HOG84 Hogger, C.J.

Introduction to Logic Programming

Academic Press, London, 1984

HU80 Huet, G.

"Confluent reductions: abstract properties and applications to term rewriting systems"

JACM 27,4 (October 1980), 797-821

JLM86 Jaffar, J., Lassez J.-L. and Maher, M. J.

"A logic programming language scheme"

in [DGLI86]

KA74 Kahn, G.

"The semantics of a simple language for parallel programming"

Proc. IFIP 1974, North-Holland, 471-475

KAMQ77 Kahn, G. and McQueen, D.B.

"Coroutines and networks of parallel processes"

Proc. IFIP 1977, North-Holland, 993-998

KO79a Kowalski, R.

"Algorithm = Logic + Control"

CACM 22,7 (July 1979), 424-436

KO79b Kowalski, R.

Logic for Problem Solving

North Holland, New York, 1979

KO85 Kowalski, R.

"Logic Programming"

BYTE 10,8 (August 1985), 161-177

LAM85 Lassez, J.-L. and Maher, M. J.

"Optimal fixedpoints of logic programs"

Theoretical Computer Science, 39 (1985) 15-23

LL88 Lloyd, J.W.

Foundations of Logic Programming second edition.

Springer-Verlag, New York, 1988

LO84 Lusk, E. and Overbeek, R.

"A portable environment for research in automated reasoning"

Proceedings of the Seventh International Conference on Automated Deduction, LNCS 170, Springer-Verlag, New York, 1984, 43-52

MAC60 McCarthy, J.

"Recursive functions of symbolic expressions and their computation by

machine, part 1"

CACM 3,4 (April 1960),184-195

MCC83 McCorduck, P

"Introduction to the Fifth Generation"

CACM 26,9 (September 1983), 629-630

MAMO Martelli, A. and Montanari, U.

"An efficient unification algorithm"

TOPLAS 4,2 (April 1982), 258-282

NIL80 Nilsson, N.J.

Principles of Artificial Intelligence

Tioga Publishing Co., Palo Alto, 1980

PERL84 Pereira, L. M.

"Logic control with logic"

in [CAM84], 177-193

POR84 Porto, A.

"Epilog: A language for extended programming in logic"

in [CAM84], 268-278

ROB65 Robinson, J. A.

"A machine-oriented logic based on the resolution principle"

JACM 12,1 (January 1965), 23-41

RO73 Rosen, B. K.

"Tree manipulating systems and Church-Rosser theorems"

JACM 20,1 (January 1973), 160-187

SHA83a Shapiro, E. Y.

"A subset of Concurrent Prolog and its interpreter"

TR-003, ICOT, Tokyo, 1983

SHA83b Shapiro, E.Y.

"The Fifth Generation Project - a trip report"

CACM 26,9 (September 1983), 673-641

STE85 Stefik, M. E.

"Strategic computing at DARPA: overview and assessment"

CACM 28,7 (July 1985), 690-704

STE82 Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., and Silver, B.

"Solving symbolic equations with PRESS"

Computer Algebra, LNCS 144, Springer-Verlag, New York, 1982, 109-116

STE84 Sterling, L.

"Logical levels of problem solving"

Journal of Logic Programming, 1:151-163 (1984)

STO77 Stoy, J. E.

Denotational Semantics

MIT Press, Cambridge, Mass., 1977

VED82 van Emden, M. H. and de Lucena Filho, G. J.

"Predicate logic as a language for parallel programming"

in [CLTA82], 189-198

VEM84 van Emden, M. H.

"An interpreting algorithm for PROLOG programs"

in [CAM84], 93-110

VK76 van Emden, M. H. and Kowalski, R.

"The semantics of predicate logic as a programming language"

JACM 23,4 (October 1976), 733-742