AN ENVIRONMENT THEORY WITH PRECOMPLETE NEGATION OVER PAIRS

by

James Harold Andrews

Technical Report 86-23

November 1986

AN ENVIRONMENT THEORY WITH PRECOMPLETE NEGATION OVER PAIRS

By

JAMES HAROLD ANDREWS

B.Sc., University of British Columbia, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

.....

.....

THE UNIVERSITY OF BRITISH COLUMBIA

September 1986

Abstract

A formal semantics of Voda's Theory of Pairs is given which takes the natural-deduction form of Gilmore's first-order set theory. The complete proof theory corresponding to this semantics is given. Then, a logic programming system is described in the form of a computational proof theory for the Gilmore semantics. This system uses parallel disjunction and the technique of precomplete negation; these features are shown to make it more complete than conventional logic programming languages.

Finally, some alternative formulations are explored which would bring the logic programming system described closer to conventional systems. The semantic problems arising from these alternatives are explored.

Included in appendices are the proof of completeness of the complete proof theory, and the environment solution algorithm which is at the heart of precomplete negation over pairs.

Acknowledgement

My family, friends, and workmates have all contributed to making these last two years enjoyable and educational. In particular, for interesting and often crucial discussions about my thesis work I would like to thank Karl Abrahamson, Peter Apostoli, Paul Gilmore, Rick Morrison, and of course Paul Voda.

This work has been supported by scholarships from the Natural Sciences and Engineering Research Council and from Bell-Northern Research Ltd., for which I am greatly appreciative.

Chapter 1

Introduction

Kowalski [Kow74] expressed the paradigm of logic programming as "the interpretation of sentences in predicate logic as programs, of derivations as computations and of proof procedures as feasible executors of predicate logic programs." This paradigm is now generally accepted. However, since the popularization of Prolog the connection between logic programming and mathematical logic has sometimes seemed vague. Research has largely diverged into two styles: the implementation of systems which are computationally powerful but theoretically far from formal logic, and the study of systems that are formally well-founded but computationally uninteresting.

Recent research has attempted to reassert the connection between traditional logic and useful programming. In Paul Voda's work, a Theory of Pairs (TP) is developed as an axiomatic first-order theory, in order to provide a logical foundation for computations over a simple, recursively-defined domain. Then, programming languages are embedded in that theory, in the sense that each programming language is defined by a proof theory which can prove a subset of the theorems of TP.

The main contribution of this part of Voda's work is not the particular programming languages set forth, but rather the enclosing structure given by the Theory of Pairs, and the simple but rigourous association of languages with logic.

This thesis makes three main contributions. Firstly, it expresses the relatively informal semantics of Voda's original papers by a natural deduction Gilmore semantics. Secondly, it provides a complete proof theory in the style of Gilmore, which can be used as a tool to prove properties of programs and programming systems. Thirdly, it describes a language which handles negation in a way which comes much closer to complete classical negation than does negation as failure, the most popular treatment of negation in logic programming.

This *precomplete* negation was defined by the author following suggestions by Voda, and is the main feature of the language. Voda has described precomplete negation

over the domain of integers in [Vod86c]; Gilmore [Gil86a] has described a very similar technique for computing negation in a database-oriented set theory.

The domain of this paper is that of the pairs; however, in the formal semantics and the proof theories described in these pages, all variables are collapsed into a single variable which takes values ranging over the *environments*. So it seems a good characterization to describe the theory as "an environment theory with precomplete negation over pairs".

Chapter 2

Research Background

The present thesis could be seen as making contributions to applications of natural deduction first-order logic with truth-value gaps; programming language semantics; and the treatment of negation in logic programming. In this chapter we trace some of the background of these areas of research.

2.1 Natural Deduction Semantics and Proof Theories

Until the time of Gentzen, the proof theories most studied were those classified as "axiomatic", that is characterized by many axioms and few rules of deduction. The proof theory presented by Hilbert and Ackermann [HA38], for instance, contains modus ponens as the sole rule of deduction. Such proof theories are consistent and complete with respect to the standard semantics, as proved by Gödel (see for instance [Men64]); however, the rules of deduction are not necessarily clearly related to the semantics.

Gentzen's approach [Gen69] was to define a proof theory using few axioms and many rules of deduction, and to emphasize the purely syntactic, proof-theoretic component of formal first-order logic. The basic elements of his proof theory were sequents, rather than individual formulae. He proved consistency of his system by showing that the empty sequent (the basic form of contradiction) could not be derived in the proof theory. The *Hauptsatz*, or Main Theorem, leading to this conclusion was his proof of cut elimination. This was the proof that a derivation using applications of the cut rule, the only rule of inference which decreased the number of formulae in a sequent, could be manipulated to remove all such applications.

The rules of deduction of Gentzen's system looked very much like the semantic entailment rules – hence the description of it as a "natural deduction" system. In

such a system, consistency and completeness of the proof theory with respect to the semantics can be proven in a simple manner, and without the use of the cut-elimination result. Such proofs, however, still resort to the intuitive notions of set and number which form the basis of model theory.

Following on this work, Beth [Bet66] and Smullyan [Smu68], among others, expressed proofs in the form of "semantic" or "analytic tableaux", which are equivalent to Gentzen-style sequent proofs. These presentations considered the basic elements of the proof theory as trees of formulae rather than as structured sequences.

Gilmore, in his presentation of first-order logic [Gil86b], adopts a style of proof theory much closer to that of Gentzen. Sentences are given a "sign" denoting truth value, as in Smullyan. Sequents are once again the basic proof-theoretic elements, but are considered as sets of signed sentences rather than sequences of formulae. This leads to a straightforward definition of validity based on intuitive notions of set theory.

The main purpose of Gilmore's work, however, is to present extensions of classical first- and second-order logic which define a formal set theory. This set theory resolves the standard set-theory paradoxes (see for instance [Bet66]) by refusing to assign a truth value to paradoxical sentences. It is on the first-order version of Gilmore's set theory, for a specific domain of constants (the domain of pairs) that the present work is largely built. The analogues of set terms are programmatic predicate definitions. Definitions can be made which would lead the language into infinite computation. Calls to predicates with such definitions are not assigned the same truth value in all bases, and therefore are effectively of indeterminate truth value.

2.2 Truth-Value Gaps

Kripke, in his presentation of a truth-value semantics for language [Kri75] which is similar to Gilmore's for set theory, discusses these truth-value gaps, arguing from a philosophical viewpoint that it is not always possible to determine the truth of an assertion. He points out that the existence of a truth-value *gap* does not imply the existence of a *third* truth value, only that some formulae cannot be assigned a truth value. He proposes a transfinite hierarchy of truth definitions for a hierarchy of abstract languages, one of which languages (at some ordinal) will be able to refer to its own truth definition.

Around the same time, Scott [Sco75] proposes a similar hierarchy for a logic based on the lambda-calculus. However, he describes the hierarchy only up to ω. Naturally only the first of the hierarchy of truth predicates corresponds to computable truth.

Fitting [Fit85] accepts the gappy first-order truth definition as a given. He formulates the standard fixpoint semantics of logic programs (see below) in a three-valued setting, although the third value is described as meaning only "undefined".

In the sequel we will present the extended first-order logic of environment with only two truth values. As Scott points out, "[an undefined formula] could indeed be given a truth value if we would want to be perverse, but there is no need to be so". However, we will sometimes act as if this third truth value exists for the purpose of clarity; we will say that a sentence takes the undefined truth value, or takes the ? sign, when that sentence's truth value is not determinate in all bases.

2.3 Programming Language Semantics

The traditional method of defining the semantics of algorithmic languages is that of *operational* semantics, in which an abstract machine is defined which computes the language. One problem with operational semantics is the possible need to define the semantics of the abstract machine, and so on down to the bit flow level; the other problem is deeper.

What is actually defined by operational semantics is simply a characterization of the computations performed by the system. This characterization could be expressed either semantically or syntactically; the syntactic expression leads to a proof theory, while the semantic expression leads to a model theory with respect to which the proof theory is trivially complete (trivially because the semantics is no more than a transcription of the proof theory into semantic concepts). *Denotational* semantics [Sto77] solves the first problem of the operational approach by basing the language on well-studied notions of set and function, but fails to solve the problem of semantic triviality.

An operational semantics for logic programming is given by van Emden and Kowalski [vEK76], but they consider it useful only as a characterization of the computations. The *fixpoint* semantics they give has become the standard semantical model for Prolog and logic programming in general. Fixpoint semantics arises from the observation that the set of all Herbrand interpretations for a set of Horn clauses (a Prolog program) form a complete lattice under the subset relation. Thus each program is seen as defining a separate first-order theory with a separate semantics.

The goal of computation is seen as finding the least fixpoint of a monotone transformation on this lattice (the predicate-application transformation). This is then proved to be the least upper bound of all the interpretations, on the lattice, of the clauses of the goal.

The standard proof theory for logic programs is some application of the resolution principle, such as SLD-resolution [AvE82]. Although SLD-resolution is complete, it is not deterministic, *i.e.*, not systematic, to use the terminology of Smullyan. Therefore, if it is to be considered as a proof theory for a computational system there must be a further description of the method of generating proofs for the system to be fully characterized.

Lloyd, therefore, adds to a fixpoint semantics in the style of van Emden and Kowalski, and the resolution-based proof theory in the style of Apt and van Emden, a "procedural semantics" for finding resolution proofs, which is really a form of operational semantics.

Voda presents a markedly less complex view of logic programming. In a series of papers ([Vod84], [Vod85], [Vod86b]), he describes a first-order theory of basic data types, the Theory of Pairs; he then presents logic programming systems as proof theories which can prove subsets of conservative extensions of that theory. Predicate definitions are conservative extensions; queries are theorems to be derived; and solutions are the completely deterministic derivations of the queries which are defined by the programming systems.

Logical semantics is not discussed in any depth in Voda's work. The traditional truth-functional semantics, in which individual sentences are assigned truth values based on their constituent subformulae, suffices. What computer scientists often refer to as "semantics" – that is, the flow of data and control in a computation – is absorbed into the proof theory and ceases to be a semantic issue at all.

No lattice theory is needed because the concepts of program, query, and solution are completely proof-theoretic; no procedural semantics is needed because the proof theory is deterministic and completely describes the computation.

The domain of the fixpoint semantics, in most treatments, is the set of constant symbols which appear in the program. The program is in this sense like a database which specifies all the objects known about in the theory. In the Theory of Pairs, on the other hand, the domain is the set of terms generated by the 0-ary generator 0 and the binary generator $[-,-]$ of pairing. These pairs correspond to the S-expressions of Lisp. The constant symbols can then be defined as being equivalent to specific pairs.

The present thesis builds on Voda's concepts of semantics by expressing the Theory of Pairs as the basis for a family of non-deterministic natural deduction proof theories. It presents on the one hand the natural deduction semantics corresponding to that family, and on the other hand a deterministic proof theory which is sound with respect to the semantics, and which proves a large subset of the theorems of that family.

2.4 Negation in Logic Programming

Van Emden and Kowalski's original conception of a logic program was as a set of Horn clauses to which some variant of resolution or Modus Ponens could be applied [vEK76]. This restriction to Horn clauses has been recognized, at least since [Cla78], as being insufficient to express all databases of formulae of first-order logic. Accordingly, most Prolog systems have retained a Horn clause-like syntax but have moved outside the domain of Horn clauses by adding some ability to negate a formula.

This negation is generally computed by the inference rules collectively called "negation as failure" by Clark [Cla78]. If a formula **A** is not true under any variable substitution, the formula $\neg$**A** is inferred to be true. The main problem with this approach is that if the formula **A** is true under some non-null variable substitution, nothing can be said about the formula $\neg$**A**. Only in the case of a null substitution can $\neg$**A** safely be inferred to be false.

This logical pitfall was apparently not avoided by many Prolog implementations. In some systems bindings occurring within negations are not retracted when other clauses using the same variables are computed. In others, the bindings are retracted but the negated goal is considered to have failed; the effect is that by finding one substitution, the system assumes that all substitutions succeed.

Clark recognized the drawbacks of negation as failure. Since he was studying in the domain of logic databases, his solution was to maintain the restriction of the database (the analogue of a program) to Horn clauses, and to restrict queries to those in which all negative conjuncts contained only variables which had been fully instantiated by other, positive conjuncts. He informally proved the completeness of computations under this restriction.

An alternate solution (expressed within a Lloyd-style theory of logic programming) is to delay the evaluation of any negative conjunct in a goal or predicate body until all the terms in the conjunct are ground; that is, until the computation of other conjuncts has restricted all variables in the conjunct to stand for single terms. Lloyd [Llo84] has proven that this scheme is sound, and some Prolog interpreters such as MU-Prolog [Nai84] have implemented it.

Voda's formalism of logic programming, in itself, does not solve the problem of handling of negation. Proof theories such as that involving one-variable environments [Vod86b] contain a version of negation as failure which does not use delayed negation (and therefore is less powerful than IC-Prolog), but which blocks when variable bindings are changed within the negated formula (ensuring soundness).

However, recent work by Voda and the author has led to a "precomplete" negation which is significantly more powerful than negation as failure. This result has been achieved in part because of the clarity of the truth-functional semantics of Voda's system. Precomplete negation may be characterized as the computational technique of treating negated identity formulae as constraints to be retained as variable bindings are retained, and of using those constraints to compute whether given additional bindings or constraints can successfully be added.

As presented here, negated formulae are solved by pushing the negation down through the formula by De Morgan's laws. When the negation reaches an equality subformula, an algorithm (similar to but more general than unification) is used which solves for equations of the form $\mathbf{a} \neq \mathbf{b}$ as well as of the form $\mathbf{a} = \mathbf{b}$.

Gilmore [Gil86a] uses a similar technique for computing negation over database queries; however, the emphasis there is on the solution of either variable-free queries or queries involving enumeration of a finite domain. Gilmore's theory, being a general set theory, is also able to handle such things as quantification over sets, which is not attempted here. We concentrate, rather, on the details of finding all terms which satisfy recursively-defined predicates over a specific domain, which Gilmore's work does not explore deeply.

This method therefore decides almost all positive formulae containing negations. Some formulae it does not decide are of the form $\mathbf{A} \vee \neg\mathbf{A}$, in which $\mathbf{A}$ contains a reference to a predicate causing infinite computation. Although the truth value of each subformula in such a sentence is indeterminate, the entire sentence should be true in a logic with classical excluded middle. Since the language described here does not have such an excluded middle, it is similar to intuitionistic logic, but it might not prove the same set of theorems.

The system could be made classically complete by letting it decide excluded-middle formulae. Voda argues, however [Vod86a], that decreased completeness is desirable for logic programming to avoid inefficient computation, and is in fact needed only for full theorem-proving.

Chapter 3

Environment Theory

This chapter gives the formal details of the system we call environment theory. The elementary syntax, that is the linguistic structure of proof-theoretic objects, is described in section 3.1; due to the natural-deduction structure of the system, many of these syntactic elements form the basis of its formal semantics, described in section 3.2. As usual, some naive set theory is assumed in the presentation of the semantics.

Section 3.3 presents a proof theory which is complete, in a strong sense, with respect to the semantics. This proof theory is therefore a valuable tool with which to analyze other proof theories. It is used as such in section 3.4, which presents a strong, though not complete, proof theory. It is this latter proof theory, R, which attains a high level of determinism by the use of precomplete negation.

The main point of this presentation is to provide some underpinning to the proof theory R, which is implementable as a logic programming system. The structure of all of environment theory is therefore dependant, in a retrograde way, on decisions taken in the design of R.

Rather than interleave a discussion of these design considerations with the discussion of the theory, we have chosen to expound on the primary design in this chapter. In the next chapter we will discuss some important variants that would have arisen from following different paths in the design process, and their relative merits.

3.1 Elementary Syntax

The elementary syntax will be used in both the logical semantics and the proof theory. In classical model theory, a mapping must be defined between model-theoretic and proof-theoretic elements; in the tradition followed here, such a mapping is unnecessary. Many of the concepts here are explained more fully in [Vod86b] and [Gil86b], and are simply summarized here.

As the basic syntactic units of our theory, we shall have an infinite set of parameters $p_1, p_2, \ldots$; an infinite set of predicate names $P_1, P_2, \ldots$; the constant symbol 0; and the variable name w. We shall use boldface **p**, **q**, **r** and **P**, **Q**, **R**, possibly subscripted, as meta-variables ranging over parameters and predicate names, respectively.

Informally, terms will be built up from the parameters and w, using the pairing function (which builds up data structures) and the projection functions (which take them apart). We follow Voda's one-variable formulation, in which all quantified entities are parts of a single environment, denoted by the variable w. The domain of this variable is the terms. However, in the deterministic proof theory R, we will use only terms which belong to the subclass of environments; this subclass will be seen to have some useful properties.

While w is called a variable here, it may be more useful to think of it as a distinguished parameter, since it will never fully be within the scope of any quantifier.

The projection functions are the symbols h (head) and t (tail). These functions correspond to the CAR and CDR functions of Lisp.

Definition 3.1.1 A *projection* is a (possibly empty) sequence of symbols h and t.

We shall use Greek letters as meta-variables ranging over projections. The empty projection will be denoted by ϵ. Projections can be concatenated in the manner of character strings; if γ contains the sequence of symbols represented by α, followed by those represented by β, then we can say that $\alpha\beta$ is identical to γ.

Definition 3.1.2 **a** is a *term* iff either

1. **a** is a parameter, the variable w, or the constant 0; or
2. **a** is of the form $\mathbf{b}\alpha$, where **b** is a term; or
3. **a** is of the form $[\mathbf{b}, \mathbf{c}]$, where **b** and **c** are terms.

We shall use boldface lower-case letters **a**, **b**, **c**, possibly subscripted, as meta-variables ranging over terms. Similarly, we shall use **x**, **y**, **z**, possibly subscripted, as meta-variables ranging over the terms of the form $w\alpha$; these terms will be called *pointers*.

The functions h and t take apart terms built up by pairing. The effect of these functions can be seen best in the definition of the metatheoretic notion of a part of a term, which we will need in later discssion.

Definition 3.1.3 For a term **a** and projection α, *part* α *of* **a**, denoted $\mathbf{a}/\alpha$, is defined as follows.

1. $\mathbf{a}/\epsilon$ is **a**;

2. $[\mathbf{a}, \mathbf{b}]/h\beta$ is $\mathbf{a}/\beta$;
3. $[\mathbf{a}, \mathbf{b}]/t\beta$ is $\mathbf{b}/\beta$;
4. Otherwise, $\mathbf{a}/\alpha$ is undefined.

The characterization of the terms $w\alpha$ as "pointers" is useful primarily when we think of terms as data structures within computer memory, or as abstract trees (see [Vod86b]). Pointers appearing within a term to be bound to w can be seen as referring to other parts of that term; that is, pointing to the left or right, or to themselves.

Definition 3.1.4 If, for a term $\mathbf{a}$, $\mathbf{a}/\alpha \equiv w\alpha$, that occurrence of the pointer $w\alpha$ is said to be a *self-pointer*. If $\mathbf{a}/\alpha h\beta \equiv w\alpha t\gamma$, that pointer is said to point to the *right*. If $\mathbf{a}/\alpha t\beta \equiv w\alpha h\gamma$, that pointer is said to point to the *left*. (Note: we use the symbol $\equiv$ to mean "is identical to", that is in the sense of metatheoretic or syntactic identity.)

Definition 3.1.5 A *w-term* is a term built up by pairing from the term 0 and the pointers $w\alpha$.

Definition 3.1.6 A *proper environment* is a w-term in which every pointer is either a self-pointer or points to the right. An *environment* is either a proper environment or the term $[w, 0]$, which is often written as fail.

Environments have two very useful properties. For every term $\mathbf{a}$, if $w = \mathbf{a}$ then there exists an environment $\mathbf{a}'$ such that $w = \mathbf{a}'$ [Vod86b]. Thus, for the purposes of computing over terms, it is sufficient to compute over environments.

For every environment $\mathbf{a}$, no infinite chain of parts $\alpha_1, \alpha_2, \ldots$ can be constructed such that $\alpha_i \equiv w\alpha_{i+1}$, unless they are all identical. Proofs about algorithms which compute with environments, such as the Environment Solution Algorithm we shall encounter later, can use this property to prove such things as termination and computational complexity.

Definition 3.1.7 $\mathbf{A}$ is a *formula* iff either:

1. $\mathbf{A}$ is of the form $\mathbf{a} = \mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are terms; or
2. $\mathbf{A}$ is of the form $\mathbf{P(a)}$, where $\mathbf{a}$ is a term; or
3. $\mathbf{A}$ is of the form $\mathbf{B} \vee \mathbf{C}$ or $\mathbf{B}$ & $\mathbf{C}$, where $\mathbf{B}$ and $\mathbf{C}$ are formulae.
4. $\mathbf{A}$ is of the form $\neg\mathbf{B}$, $\exists\mathbf{B}$, or $\forall\mathbf{B}$, where $\mathbf{B}$ is a formula; or

If $\mathbf{A}$ is of one of the first two forms, it is also an *atomic* formula. If all terms appearing in the formula are w-terms, it is also a *w-formula*.

We shall use boldface capital letters **A**, **B**, **C**, possibly subscripted, as meta-variables ranging over formulae.

Note the absence of quantified variables after the quantifiers $\exists$ and $\forall$. Since we are using Voda's one-variable formulation, a quantifier itself can be thought of as generating a new variable to be referred to within its scope. The new variable is the head of the environment w, and the environment outside the scope of the quantifier is referred to as wt within the quantifier.

Definition 3.1.8 Because what a pointer signifies depends on the formula it appears in, each pointer is actually an *alias* of some other pointer within a given formula.

1. An occurrence of a pointer $w\alpha$ is an alias of itself within an atomic formula.
2. If an occurrence of $w\alpha$ is an alias of $w\beta$ within **A** or **B**, it is also an alias of $w\beta$ within **A** & **B**, **A** $\vee$ **B**, or $\neg$**A**.
3. If an occurrence of $w\alpha$ is an alias of $wt\beta$ within **A**, it is an alias of $w\beta$ within $\exists$**A** or $\forall$**A**.

Readers may think of the parameters as being free in any formula in which they appear, and the terms $w\alpha$ as being free within the scope of n quantifiers if and only if α begins with n or more t's. But since a formula with such "free variables" will still take on a truth value in the semantics, without substitutions for the free variables, notions of free and bound variables are largely irrelevant. We have the following definitions.

Definition 3.1.9 A *sentence* is a formula. (We will use this word when we wish to emphasize the lack of free variables in any well-formed formula.) A *signed sentence* is a sentence preceded by a + or − sign.

Definition 3.1.10 The *complexity* of a formula **A**, *comp*(**A**), is the following.

1. The complexity of **a** = **b** and **P**(**a**) are 0.
2. The complexity of $\neg$**A**, $\exists$**A**, and $\forall$**A** are *comp*(**A**) + 1.
3. The complexity of **A** & **B** is the greater of *comp*(**A**) and *comp*(**B**).

The complexity of a signed sentence $\pm$**A** is *comp*(**A**).

Definition 3.1.11 **a**{**b** := **c**} represents the term **a**, with all occurrences of the subterm **b** substituted by occurrences of the subterm **c** (*strong* substitution). Formally, **b**{**b** := **c**} $\equiv$ **c**, and if the term being substituted in is not identical to **b**, then:

1. $0\{\mathbf{b} := \mathbf{c}\} \equiv 0$

2. $[\mathbf{a}_1, \mathbf{a}_2]\{\mathbf{b} := \mathbf{c}\} \equiv [\mathbf{a}_1\{\mathbf{b} := \mathbf{c}\}, \mathbf{a}_2\{\mathbf{b} := \mathbf{c}\}]$
3. $\mathbf{a}h\{\mathbf{b} := \mathbf{c}\} \equiv (\mathbf{a}\{\mathbf{b} := \mathbf{c}\})h$
4. $\mathbf{a}t\{\mathbf{b} := \mathbf{c}\} \equiv (\mathbf{a}\{\mathbf{b} := \mathbf{c}\})t$

Strong substitution in a formula, $\mathbf{A}\{\mathbf{b} := \mathbf{c}\}$, is defined similarly.

1. $(\mathbf{a}_1 = \mathbf{a}_2)\{\mathbf{b} := \mathbf{c}\} \equiv (\mathbf{a}_1\{\mathbf{b} := \mathbf{c}\} = \mathbf{a}_2\{\mathbf{b} := \mathbf{c}\})$
2. $\mathbf{P}(\mathbf{a})\{\mathbf{b} := \mathbf{c}\} \equiv \mathbf{P}(\mathbf{a}\{\mathbf{b} := \mathbf{c}\})$
3. $(\neg\mathbf{A})\{\mathbf{b} := \mathbf{c}\} \equiv \neg\mathbf{A}\{\mathbf{b} := \mathbf{c}\}$
4. $(\mathbf{A} \,\&\, \mathbf{B})\{\mathbf{b} := \mathbf{c}\} \equiv \mathbf{A}\{\mathbf{b} := \mathbf{c}\} \,\&\, \mathbf{B}\{\mathbf{b} := \mathbf{c}\}$
5. $(\mathbf{A} \vee \mathbf{B})\{\mathbf{b} := \mathbf{c}\} \equiv \mathbf{A}\{\mathbf{b} := \mathbf{c}\} \,\&\, \mathbf{B}\{\mathbf{b} := \mathbf{c}\}$
6. $(\exists\mathbf{A})\{\mathbf{b} := \mathbf{c}\} \equiv \exists\mathbf{A}\{\mathbf{b} := \mathbf{c}\}$
7. $(\forall\mathbf{A})\{\mathbf{b} := \mathbf{c}\} \equiv \forall\mathbf{A}\{\mathbf{b} := \mathbf{c}\}$

Note that the last two subcases of strong substitution in a formula do not always adequately handle the renaming of parts of the environment; that is, they make the same mistake with renaming often made in early axiomatic first-order theories. Often, we will want to remedy this by substituting for all aliases of w in a term or formula. We will also want to do so in a contextual manner, with wt being substituted by the tail of the substituting term, and so on. Such a substitution, while achieving the desired semantic intent, will also preserve the form of a term as a w-term or an environment [Vod84].

Definition 3.1.12 The *contextual substitution* of a term $\mathbf{b}$ for w in another term $\mathbf{a}$, written $\mathbf{a}(\mathbf{b})$, or in a formula $\mathbf{A}$, written $\mathbf{A}(\mathbf{b})$, is defined as follows.

1. $0(\mathbf{a}) \equiv 0$
2. $[\mathbf{a}, \mathbf{b}](\mathbf{c}) \equiv [\mathbf{a}(\mathbf{c}), \mathbf{b}(\mathbf{c})]$
3. $w\alpha(w\beta) \equiv w\beta\alpha$
4. $w\alpha(0) \equiv 0$
5. $w([\mathbf{a}, \mathbf{b}]) \equiv [\mathbf{a}, \mathbf{b}]$
6. $wh\alpha([\mathbf{a}, \mathbf{b}]) \equiv w\alpha(\mathbf{a})$
7. $wt\alpha([\mathbf{a}, \mathbf{b}]) \equiv w\alpha(\mathbf{b})$
8. $(\mathbf{a} = \mathbf{b})(\mathbf{c}) \equiv \mathbf{a}(\mathbf{c}) = \mathbf{b}(\mathbf{c})$
9. $\mathbf{P}(\mathbf{a})(\mathbf{b}) \equiv \mathbf{P}(\mathbf{a}(\mathbf{b}))$

10. $(\neg\mathbf{A})(\mathbf{a}) \equiv \neg\mathbf{A}(\mathbf{a})$
11. $(\mathbf{A}\ \&\ \mathbf{B})(\mathbf{a}) \equiv \mathbf{A}(\mathbf{a})\ \&\ \mathbf{B}(\mathbf{a})$
12. $(\mathbf{A} \vee \mathbf{B})(\mathbf{a}) \equiv \mathbf{A}(\mathbf{a}) \vee \mathbf{B}(\mathbf{a})$
13. $(\exists\mathbf{A})(\mathbf{a}) \equiv \exists\mathbf{A}(wh, \mathbf{a}(wt))$
14. $(\forall\mathbf{A})(\mathbf{a}) \equiv \forall\mathbf{A}(wh, \mathbf{a}(wt))$

The basic elements of the proof theory will be sequents. These are defined as sets, following Smullyan, rather than as Gentzen's purely syntactic entities, in order to simplify the relationships between the syntax and semantics.

Definition 3.1.13 A *sequent* is a possibly infinite set of signed sentences with either the + or the − sign. A *finite* sequent is a sequent with a finite set of elements.

We will often use a notation similar to that of Gentzen to represent sequents. Every sequent S is the union of a set $\{+P_1, +P_2, +P_3, \ldots\}$ of sentences with a + sign, and a set $\{-M_1, -M_2, -M_3, \ldots\}$ of sentences with a − sign. We can therefore represent S uniquely with the notation $\{M_1, M_2, M_3, \ldots \rightarrow P_1, P_2, P_3, \ldots\}$. The Greek letters Γ and Δ, possibly subscripted, shall be used to represent sequences of (unsigned) sentences; thus, the standard form for a sequent shall be $\{\Gamma \rightarrow \Delta\}$.

Definition 3.1.14 If $\mathbf{P}$ is a predicate name and $\mathbf{A}$ is a sentence, then $\mathbf{P}(w) \leftrightarrow \mathbf{A}$ is a *predicate definition.* We shall refer to a finite set of predicate definitions as a *program.*

3.2 Formal Semantics

The model-theoretic part of environment theory consists of a definition of a base, that is one assignment of signs to atomic sentences, and rules for constructing the set of sentences entailed by a base. As with the definition of sequent in the last section, much of the formal semantics relies on informal notions of sets and natural numbers.

This material is derivative of the semantic definitions of [Gil86b], but differs in several important respects. Bases are restricted to those which satisfy identity over pairs, which has special properties in addition to those of standard identity. The quantifiers take the one-variable interpretation described informally in the last section. And, most important to the structure of the semantics, we will be interested only in bases which are interpretations of a given program Π, in the sense that they satisfy a substitutivity criterion derived from Π.

Definition 3.2.1 A *base* $\mathcal{B}$ is a set of signed atomic sentences such that the following conditions hold.

1. For every atomic sentence $\mathbf{A}$, exactly one of $+\mathbf{A}$ and $-\mathbf{A}$ is a member of $\mathcal{B}$. (Completeness and atomic excluded middle.)
2. For every atomic sentence $\mathbf{A}$, terms $\mathbf{a}$ and $\mathbf{b}$, and parameter $\mathbf{p}$, whenever $\pm\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ and $\mp\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ are both in $\mathcal{B}$, $-\mathbf{a} = \mathbf{b}$ is in $\mathcal{B}$.
3. For all terms $\mathbf{a}$, $+\mathbf{a} = \mathbf{a}$ is in $\mathcal{B}$. (This clause and the one preceding define standard identity.)
4. For all terms $\mathbf{a}$ and $\mathbf{b}$, the following signed sentences are in $\mathcal{B}$:
   (a) $+0 = 0\alpha$, $0\alpha = 0$
   (b) $-[\mathbf{a},\mathbf{b}] = [\mathbf{a},\mathbf{b}]\alpha$, $-[\mathbf{a},\mathbf{b}]\alpha = [\mathbf{a},\mathbf{b}]$ for all nonempty α
   (c) $+[\mathbf{a},\mathbf{b}]h = \mathbf{a}$, $+\mathbf{a} = [\mathbf{a},\mathbf{b}]h$
   (d) $+[\mathbf{a},\mathbf{b}]t = \mathbf{b}$, $+\mathbf{b} = [\mathbf{a},\mathbf{b}]t$
5. For all terms $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, whenever all of $+\mathbf{a}h = \mathbf{b}, +\mathbf{a}t = \mathbf{c}$, and $-\mathbf{a} = 0$) are in $\mathcal{B}$, $+\mathbf{a} = [\mathbf{b},\mathbf{c}]$ is in $\mathcal{B}$.
6. For all terms $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, whenever one of $-\mathbf{a}h = \mathbf{b}, -\mathbf{a}t = \mathbf{c}$, or $+\mathbf{a} = 0$) is in $\mathcal{B}$, $-\mathbf{a} = [\mathbf{b},\mathbf{c}]$ is in $\mathcal{B}$. (This clause and the two preceding define the special properties of identity between pairs.)

Bases give the truth value for atomic sentences; the truth value for non-atomic sentences can be derived inductively. We use the concepts of semantic entailment, semantic successor, and closure of a base to collect all the semantic consequences of the sentences in a base.

Definition 3.2.2 We shall say that a set of signed sentences *entails* another signed sentence ($\{\pm\mathbf{A}, \pm\mathbf{B}, \ldots\} \vdash\!\!\!\rightarrow \pm\,\mathbf{C}$) in the following cases, assuming that $\mathbf{A}$ and $\mathbf{B}$ are formulae.

1. Conjunction:
   (a) $\{+\mathbf{A}, +\mathbf{B}\} \vdash\!\!\!\rightarrow +\,\mathbf{A}\,\&\,\mathbf{B}$
   (b) $\{-\mathbf{A}\} \vdash\!\!\!\rightarrow -\,\mathbf{A}\,\&\,\mathbf{B}$
   (c) $\{-\mathbf{B}\} \vdash\!\!\!\rightarrow -\,\mathbf{A}\,\&\,\mathbf{B}$
2. Disjunction:
   (a) $\{+\mathbf{A}\} \vdash\!\!\!\rightarrow +\,\mathbf{A} \vee \mathbf{B}$
   (b) $\{+\mathbf{B}\} \vdash\!\!\!\rightarrow +\,\mathbf{A} \vee \mathbf{B}$
   (c) $\{-\mathbf{A}, -\mathbf{B}\} \vdash\!\!\!\rightarrow -\,\mathbf{A} \vee \mathbf{B}$

3. Negation: $\{\pm\mathbf{A}\} \vdash\!\!\!\rightarrow \mp\neg\mathbf{A}$
4. Existential quantifier:
   (a) $\{+\mathbf{A}([\mathbf{a}, w])\} \vdash\!\!\!\rightarrow +\exists\mathbf{A}$
   where $\mathbf{a}$ is some term
   (b) $\{-\mathbf{A}([\mathbf{a}_1, w]), -\mathbf{A}([\mathbf{a}_2, w]), \ldots\} \vdash\!\!\!\rightarrow -\exists\mathbf{A}$
   where $\mathbf{a}_1, \mathbf{a}_2, \ldots$ is the enumeration of all the terms
5. Universal quantifier:
   (a) $\{+\mathbf{A}([\mathbf{a}_1, w]), +\mathbf{A}([\mathbf{a}_2, w]), \ldots\} \vdash\!\!\!\rightarrow +\forall\mathbf{A}$
   where $\mathbf{a}_1, \mathbf{a}_2, \ldots$ is the enumeration of all the terms
   (b) $\{-\mathbf{A}([\mathbf{a}, w])\} \vdash\!\!\!\rightarrow -\forall\mathbf{A}$
   where $\mathbf{a}$ is some term

Definition 3.2.3 The *semantic successor* of a set of signed sentences S, $succ(S)$, is the set of signed sentences σ for which either $\sigma \in S$, or there exists a subset T of S such that $T \vdash\!\!\!\rightarrow \sigma$. We define the n^{th} *semantic successor* of S, $succ^n(S)$, for all $n \geq 0$, as follows: $succ^0(S)$ is S, and $succ^{n+1}(S)$ is $succ(succ^n(S))$.

Definition 3.2.4 The *closure* of a base $\mathcal{B}$, $Cl(\mathcal{B})$, is the union of all the semantic successors of $\mathcal{B}$. That is, a signed sentence is in $Cl(\mathcal{B})$ iff it is in $succ^i(\mathcal{B})$ for some finite i.

Definition 3.2.5 A base $\mathcal{B}$ is an *interpretation* of a program Π if, for every predicate definition $\mathbf{P}(w) \leftrightarrow \mathbf{A}$ in Π, and every term $\mathbf{a}$, whenever $\pm\mathbf{A}(\mathbf{a})$ is in $Cl(\mathcal{B})$, $\pm\mathbf{P}(\mathbf{a})$ is in $\mathcal{B}$.

Definition 3.2.6 A base $\mathcal{B}$ is a *model* of a sequent S (equivalently, S is *true in* $\mathcal{B}$) iff $S \cap Cl(\mathcal{B}) \not\equiv \emptyset$; that is, if there is a signed sentence σ such that $\sigma \in S$ and $\sigma \in succ^i(\mathcal{B})$ for some i.

Definition 3.2.7 A sequent S is *valid* with respect to a program Π iff all interpretations of Π are models of S; that is, iff S is true in all interpretations of Π.

Note that these definitions of truth and validity obtain precisely because intuitive set theory is used for defining both the model-theoretic notions of base and entailment, and the proof-theoretic notion of sequent.

Theorem 3.2.8 (Closure Completeness and Consistency) For every base $\mathcal{B}$ and every sentence $\mathbf{A}$, exactly one of the signed sentences $\pm\mathbf{A}$ appears in $Cl(\mathcal{B})$.

Proof. By induction on the complexity of **A**. All atomic sentences (complexity 0) are in $\mathcal{B}$ with exactly one sign, and the n^{th} semantic successor of $\mathcal{B}$ contains all the signed sentences of complexity $n+1$; the semantic entailment rules determine uniquely the sign of each sentence. □

Bases vary in the assignment of sign to individual atomic sentences, and the assignment of values to parameters and the variable w. The set of interpretations of an individual program is smaller than the set of all bases; the only variability is in the assignment of values to the parameters and w, and the assignment of sign to predicate calls for which the corresponding defining sentence is never computed.

For every predicate name **P** defined by a program Π, there is a set of predicate call formulae $\mathbf{P(a)}$ whose sign must be the same in all interpretations of Π. If the definition $\mathbf{P}(w) \leftrightarrow 0 = 0$ is in Π, for example, $+\mathbf{P(a)}$ must be in all interpretations of Π. However, if the definition $\mathbf{P}(w) \leftrightarrow \mathbf{P}(w)$ is in Π, then $\mathbf{P(a)}$ can be given any sign in interpretations of Π; and if $\mathbf{P}(w) \leftrightarrow \neg\mathbf{P}(w)$ is in Π, then no interpretations of Π exist!

As we will see, the predicate call formulae whose signs are not determined correspond exactly to those predicate calls whose computation causes a search down an infinite branch of the solution tree; that is, in programming language terms, infinite recursion. Some complex predicate definitions may result in success for some calls, failure for some calls, and infinite recursion for others; the first class corresponds to formulae assigned a $+$, the second to formulae assigned a $-$, and the third to formulae assigned different signs in different bases.

Since the sign of some predicate calls is not determined, it may be helpful to imagine the existence of a third truth value, "unknown", represented by the sign ?. Then a truth table with this third sign, for the above definitions of conjunction and disjunction, may be given as follows.

Conjunction:

&	+	−	?
+	+	−	?
−	−	−	−
?	?	−	?

Disjunction:

∨	+	−	?
+	+	+	+
−	+	−	?
?	+	?	?

These tables illustrate the fact that a formula's sign is determined in all bases if and only if there is enough information to determine that sign. The sign of both immediate subformulae is not always necessary for this determination, but at least one is.

For the purposes of this exposition, it is more useful to assume the existence of only two truth values, because this gives such properties as excluded middle for all formulae, which aid in some proofs. However, readers may find it helpful to relate the material that follows to the formulation with three truth values.

3.3 Proof theory C: Complete

We present here a complete natural deduction proof theory for the environment theory semantics. This proof theory cannot be used to directly implement a theorem-prover or logic programming system; the high degree of non-determinacy means that there must be additional information about how such things as eigenvalues of existential formulae are searched for. The next section can be seen as an attempt to put more information about implementation details into a proof theory, to make it more deterministic. In fact, this can be seen to be the thrust of much of Voda's recent work.

Such a proof theory has the same value as in set theories (see for instance [Gil86b]); that is, it allows one to give a computably verifiable derivation of a valid sequent. It will be used later to prove important properties of the computational proof theory.

Although we refer to C as one proof theory, it is in fact a family of proof theories C+Π, which are identical save for the rules dealing with the introduction of formulae including predicates defined in the program Π.

Axioms are of one of the following forms, for any terms **a** and **b** and atomic formula **A**:

1. $\{\Gamma, \mathbf{A} \rightarrow \Delta, \mathbf{A}\}$
2. $\{\Gamma, [\mathbf{a}, \mathbf{b}] = [\mathbf{a}, \mathbf{b}]\alpha \rightarrow \Delta\}$, $\{\Gamma, [\mathbf{a}, \mathbf{b}]\alpha = [\mathbf{a}, \mathbf{b}] \rightarrow \Delta\}$ for nonempty α
3. $\{\Gamma \rightarrow \Delta, 0\alpha = 0\}$, $\{\Gamma \rightarrow \Delta, 0 = 0\alpha\}$
4. $\{\Gamma \rightarrow \Delta, [\mathbf{a}, \mathbf{b}]h = \mathbf{a}\}$, $\{\Gamma \rightarrow \Delta, \mathbf{a} = [\mathbf{a}, \mathbf{b}]h\}$
5. $\{\Gamma \rightarrow \Delta, [\mathbf{a}, \mathbf{b}]t = \mathbf{b}\}$, $\{\Gamma \rightarrow \Delta, \mathbf{b} = [\mathbf{a}, \mathbf{b}]t\}$
6. $\{\Gamma \rightarrow \Delta, \mathbf{a} = \mathbf{a}\}$

The rules of inference of C+Π, for any program Π, follow.

1. Pairing
   (a) left:

$$\frac{\{\Gamma, \mathbf{a}h = \mathbf{b}, \mathbf{a}t = \mathbf{c} \rightarrow \Delta, \mathbf{a} = 0\}}{\{\Gamma, \mathbf{a} = [\mathbf{b}, \mathbf{c}] \rightarrow \Delta\}}$$

(b) right:

$$\frac{\{\Gamma, \mathbf{a} = 0 \to \Delta\}\ \{\Gamma \to \Delta, \mathbf{a}h = \mathbf{b}\}\ \{\Gamma \to \Delta, \mathbf{a}t = \mathbf{c}\}}{\{\Gamma \to \Delta, \mathbf{a} = [\mathbf{b}, \mathbf{c}]\}}$$

2. Identity, left:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}\{\mathbf{p} := \mathbf{a}\}\}\ \{\Gamma, \mathbf{A}\{\mathbf{p} := \mathbf{b}\} \to \Delta\}}{\{\Gamma, \mathbf{a} = \mathbf{b} \to \Delta\}}$$

3. Conjunction

(a) left:

$$\frac{\{\Gamma, \mathbf{A}, \mathbf{B} \to \Delta\}}{\{\Gamma, \mathbf{A} \,\&\, \mathbf{B} \to \Delta\}}$$

(b) right:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}\}\ \{\Gamma \to \Delta, \mathbf{B}\}}{\{\Gamma \to \Delta, \mathbf{A} \,\&\, \mathbf{B}\}}$$

4. Disjunction

(a) left:

$$\frac{\{\Gamma, \mathbf{A} \to \Delta\}\ \{\Gamma, \mathbf{B} \to \Delta\}}{\{\Gamma, \mathbf{A} \vee \mathbf{B} \to \Delta\}}$$

(b) right:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}, \mathbf{B}\}}{\{\Gamma \to \Delta, \mathbf{A} \vee \mathbf{B}\}}$$

5. Negation

(a) left:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}\}}{\{\Gamma, \neg\mathbf{A} \to \Delta\}}$$

(b) right:

$$\frac{\{\Gamma, \mathbf{A} \to \Delta\}}{\{\Gamma \to \Delta, \neg\mathbf{A}\}}$$

6. Existential

(a) left:

$$\frac{\{\Gamma, \mathbf{A}([\mathbf{p}, w]) \to \Delta\}}{\{\Gamma, \exists\mathbf{A} \to \Delta\}}$$

for some parameter $\mathbf{p}$ which does not appear in the conclusion

(b) right:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}([\mathbf{a}, w])\}}{\{\Gamma \to \Delta, \exists \mathbf{A}\}}$$

for some term $\mathbf{a}$

7. Universal

(a) left:

$$\frac{\{\Gamma, \mathbf{A}([\mathbf{a}, w]) \to \Delta\}}{\{\Gamma, \forall \mathbf{A} \to \Delta\}}$$

for some term $\mathbf{a}$

(b) right:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}([\mathbf{p}, w])\}}{\{\Gamma \to \Delta, \forall \mathbf{A}\}}$$

for some parameter $\mathbf{p}$ which does not appear in the conclusion

8. Predicate introduction

(a) left:

$$\frac{\{\Gamma, \mathbf{A}(\mathbf{a}) \to \Delta\}}{\{\Gamma, \mathbf{P}(\mathbf{a}) \to \Delta\}}$$

where the definition $\mathbf{P}(w) \leftrightarrow \mathbf{A}$ is in Π

(b) right:

$$\frac{\{\Gamma \to \Delta, \mathbf{A}(\mathbf{a})\}}{\{\Gamma \to \Delta, \mathbf{P}(\mathbf{a})\}}$$

where the definition $\mathbf{P}(w) \leftrightarrow \mathbf{A}$ is in Π

These rules can all be proved to preserve validity; that is, if the premiss of rule application is a valid sequent, then the conclusion is a valid sequent.

We can derive with these rules the intuitive truths that 0 is not equal to any pair:

$$\frac{\{0h = \mathbf{a}, 0t = \mathbf{b} \to 0 = 0\}}{\{0 = [\mathbf{a}, \mathbf{b}] \to\}}$$

That every term is either 0 or a pair:

$$\frac{\{\to \mathbf{a}h = \mathbf{a}h, \mathbf{a} = 0\}\ \{\to \mathbf{a}t = \mathbf{a}t, \mathbf{a} = 0\}\ \{\mathbf{a} = 0 \to \mathbf{a} = 0\}}{\{\to \mathbf{a} = [\mathbf{a}h, \mathbf{a}t], \mathbf{a} = 0\}}$$

And that no term is both 0 and a pair:

$$\frac{\{\mathbf{a} = 0, \mathbf{a}h = \mathbf{b}, \mathbf{a}t = \mathbf{c} \rightarrow \mathbf{a} = 0\}}{\{\mathbf{a} = 0, \mathbf{a} = [\mathbf{b}, \mathbf{c}] \rightarrow\}}$$

Somewhat longer derivations show that the sequents $\{[\mathbf{a}, \mathbf{b}] = [\mathbf{c}, \mathbf{d}] \rightarrow \mathbf{a} = \mathbf{c}\ \&\ \mathbf{b} = \mathbf{d}\}$ and $\{\mathbf{a} = \mathbf{c}\ \&\ \mathbf{b} = \mathbf{d} \rightarrow [\mathbf{a}, \mathbf{b}] = [\mathbf{c}, \mathbf{d}]\}$ are derivable.

Theorem 3.3.1 (Completeness) If a sequent S is valid with respect to Π, then it is derivable in C with respect to Π using the above rules.

Proof. See Appendix A.

Theorem 3.3.2 (Thinning) If S is valid with respect to Π, then so is $S \cup S'$.

Proof. Assume the premise, that is that $S \cap Cl(\mathcal{B})$ is nonempty for all interpretations $\mathcal{B}$ of Π. Then clearly $(S \cup S') \cap Cl(\mathcal{B})$ is also nonempty for all such $\mathcal{B}$, and is therefore also valid with respect to Π. □

Theorem 3.3.3 (Cut) If $S \cup \{+\mathbf{A}\}$ and $S \cup \{-\mathbf{A}\}$ are valid with respect to Π, then so is S.

Proof. Assume S is not valid with respect to some Π. Then there must exist some $\mathcal{B}$ which is an interpretation of Π but not a model of S; that is, such that $S \cap Cl(\mathcal{B}) \not\equiv \emptyset$. By the Closure Completeness Theorem (3.2.8), exactly one of $-\mathbf{A}$ or $+\mathbf{A}$ must be a member of $Cl(\mathcal{B})$; therefore, either $S \cup +\mathbf{A}$ or $S \cup -\mathbf{A}$ must have an empty intersection with $\mathcal{B}$. Contrapositively, if both $S \cup +\mathbf{A}$ and $S \cup -\mathbf{A}$ have non-empty intersections with $\mathcal{B}$, then there is no Π with respect to which S is not valid; that is, S is valid. □

Theorems 3.3.2 (Thinning) and 3.3.3 (Cut) motivate us to add two rules to C to facilitate derivations. These rules are, as shown by the Completeness theorem, not necessary to ensure completeness of C, but have been found to be useful in shortening derivations.

1. Thinning

   (a) left:

$$\frac{\{\Gamma \rightarrow \Delta\}}{\{\Gamma, \mathbf{A} \rightarrow \Delta\}}$$

   (b) right:

$$\frac{\{\Gamma \rightarrow \Delta\}}{\{\Gamma \rightarrow \Delta, \mathbf{A}\}}$$

2. Cut:

$$\frac{\{\Gamma \rightarrow \Delta, \mathbf{A}\}\ \{\Gamma, \mathbf{A} \rightarrow \Delta\}}{\{\Gamma \rightarrow \Delta\}}$$

3.4 Proof Theory R: R$^+$-Maple

This section presents a proof theory, R, which is similar to the language R$^+$-Maple, described in [Vod86b]. The main differences are that R uses parallel disjunction and precomplete negation, whereas R$^+$-Maple uses left-to-right sequential disjunction and negation as failure. Left-to-right disjunction and negation as failure are two computational techniques which were considered for this proof theory, but were rejected for reasons explained in the next chapter.

Again, R is used to refer to a family of proof theories R+Π, where Π is any program.

To assist in the proof of validity of R, we will need to prove some preliminary theorems.

Lemma 3.4.1 (Substitutivity) If $\{\Gamma \to \Delta\}$ is a valid sequent, then so is $\{\Gamma([\mathbf{p}, w]) \to \Delta([\mathbf{p}, w])\}$.

Proof. If $\{\Gamma \to \Delta\}$ is an axiom of C, then $\{\Gamma([\mathbf{p}, w]) \to \Delta([\mathbf{p}, w])\}$ must be of one of the forms $\{\Gamma' \to \Delta', 0\alpha = 0\}$, $\{\Gamma', [\mathbf{a}', \mathbf{b}'] = [\mathbf{a}', \mathbf{b}']\alpha \to \Delta'\}$, $\{\Gamma' \to \Delta', [\mathbf{a}', \mathbf{b}']h = \mathbf{a}'\}$, $\{\Gamma' \to \Delta', [\mathbf{a}', \mathbf{b}']t = \mathbf{b}'\}$, $\{\Gamma' \to \Delta', \mathbf{a}' = \mathbf{a}'\}$, or $\{\Gamma', \mathbf{a}'\alpha = \mathbf{a}' \to \Delta', \mathbf{a}' = 0\}$, with the terms in the indicated identity formulae possibly exchanged. All of these are also axioms of C and therefore valid.

If $\{\Gamma \to \Delta\}$ is not an axiom of C but is valid, then it must have a derivation in C. We can transform the proof tree of $\{\Gamma \to \Delta\}$ into a tree with $\{\Gamma([\mathbf{p}, w]) \to \Delta([\mathbf{p}, w])\}$ at the root by first replacing all occurrences of $\mathbf{p}$ by occurrences of some other parameter (to preserve the validity of $\exists$-left rule applications), and then replacing all occurrences of w in the tree by occurrences of $[\mathbf{p}, w]$. Axioms will be transformed into axioms by this transformation, and valid rule applications into valid rule applications. The resulting tree will therefore be a well-formed proof tree of C, and by consistency of C $\{\Gamma([\mathbf{p}, w]) \to \Delta([\mathbf{p}, w])\}$ will be valid. □

Theorem 3.4.2 (Expansion) If $\{\mathbf{A} \to \mathbf{B}\}$ and $\{\mathbf{B} \to \mathbf{A}\}$ are both valid, then so are $\{\ldots\mathbf{A}\ldots \to \ldots\mathbf{B}\ldots\}$ and $\{\ldots\mathbf{B}\ldots \to \ldots\mathbf{A}\ldots\}$. That is, so are $\{\mathbf{A}' \to \mathbf{B}'\}$ and $\{\mathbf{B}' \to \mathbf{A}'\}$, where $\mathbf{A}'$ is any formula of which $\mathbf{A}$ is a subformula, and $\mathbf{B}'$ is just $\mathbf{A}'$ with the subformula $\mathbf{A}$ replaced by the subformula $\mathbf{B}$.

Proof. By induction on the difference in complexity between $\mathbf{A}$ and $\mathbf{A}'$. Let the complexity of $\mathbf{A}$ be i. Assume that $\{\mathbf{A} \to \mathbf{B}\}$ and $\{\mathbf{B} \to \mathbf{A}\}$ are both valid.

Let $P(j)$ be the proposition that $\{\mathbf{A}' \to \mathbf{B}'\}$ and $\{\mathbf{B}' \to \mathbf{A}'\}$ are valid, where $\mathbf{A}'$ is any formula of complexity j of which $\mathbf{A}$ is a subformula, and $\mathbf{B}'$ is just $\mathbf{A}'$ with an occurrence of $\mathbf{A}$ replaced by an occurrence of $\mathbf{B}$.

I. $j - i = 0$. $\{\mathbf{A}' \to \mathbf{B}'\}$ and $\{\mathbf{B}' \to \mathbf{A}'\}$, are just $\{\mathbf{A} \to \mathbf{B}\}$ and $\{\mathbf{B} \to \mathbf{A}\}$, and so are trivially valid.

II. $j - i > 0$. Assume that $\mathrm{P}(j)$ holds for all $i < j \leq k$, and consider $\mathrm{P}(k+1)$. The complexity of $\mathbf{A}'$ is $k+1$, and $\mathbf{A}$ must be one of several forms.

1. $\mathbf{A}' \equiv \mathbf{C}\ \&\ \ldots\mathbf{A}\ldots$. We have the derivation

$$\dfrac{\dfrac{\{\mathbf{C}, \ldots\mathbf{A}\ldots \to \mathbf{C}\}}{\{\mathbf{C}\ \&\ \ldots\mathbf{A}\ldots \to \mathbf{C}\}} \quad \dfrac{\{\mathbf{C}, \ldots\mathbf{A}\ldots \to \ldots\mathbf{B}\ldots\}}{\{\mathbf{C}\ \&\ \ldots\mathbf{A}\ldots \to \ldots\mathbf{B}\ldots\}}}{\{\mathbf{C}\ \&\ \ldots\mathbf{A}\ldots \to \mathbf{C}\ \&\ \ldots\mathbf{B}\ldots\}}$$

   Clearly $\{\mathbf{C} \to \mathbf{C}\}$ is an axiom, and $\{\ldots\mathbf{A}\ldots \to \ldots\mathbf{B}\ldots, \mathbf{C}\}$ is valid by the induction assumption and Theorem 3.3.2. The derivation of $\{\mathbf{B}' \to \mathbf{A}'\}$ is similar, as are the derivations for the case when $\mathbf{A}' \equiv \ldots\mathbf{A}\ldots\ \&\ \mathbf{C}$.

2. $\mathbf{A}' \equiv \mathbf{C} \vee \ldots\mathbf{A}\ldots$, $\mathbf{A}' \equiv \ldots\mathbf{A}\ldots \vee \mathbf{C}$. Similar to the cases in (1).

3. $\mathbf{A}' \equiv \neg\ldots\mathbf{A}\ldots$. In this case $\{\mathbf{A}' \to \mathbf{B}'\}$ is $\{\neg\ldots\mathbf{A}\ldots \to \neg\ldots\mathbf{B}\ldots\}$. We have the derivation

$$\dfrac{\dfrac{\{\ldots\mathbf{B}\ldots \to \ldots\mathbf{A}\ldots\}}{\{\to \neg\ldots\mathbf{B}\ldots, \ldots\mathbf{A}\ldots\}}}{\{\neg\ldots\mathbf{A}\ldots \to \neg\ldots\mathbf{B}\ldots\}}$$

   and since $\{\ldots\mathbf{B}\ldots \to \ldots\mathbf{A}\ldots\}$ is valid by the induction assumption, so is $\{\mathbf{A}' \to \mathbf{B}'\}$. The derivation of $\{\mathbf{B}' \to \mathbf{A}'\}$ is similar.

4. $\mathbf{A}' \equiv \exists\ldots\mathbf{A}\ldots$. We have the derivation

$$\dfrac{\dfrac{\{\ldots\mathbf{A}\ldots([\mathbf{p}, w]) \to \ldots\mathbf{B}\ldots([\mathbf{p}, w])\}}{\{\ldots\mathbf{A}\ldots([\mathbf{p}, w]) \to \exists\ldots\mathbf{B}\ldots\}}}{\{\exists\ldots\mathbf{A}\ldots \to \exists\ldots\mathbf{B}\ldots\}}$$

   and by Lemma 3.4.1 and the induction assumption,
   $\{\ldots\mathbf{A}([\mathbf{p}, w])\ldots \to \ldots\mathbf{B}([\mathbf{p}, w])\ldots\}$ is valid; therefore so is $\{\mathbf{A}' \to \mathbf{B}'\}$. The derivation of $\{\mathbf{B}' \to \mathbf{A}'\}$ is similar.

5. $\mathbf{A}' \equiv \forall\ldots\mathbf{A}\ldots$. Similar to the cases in (4).

III. Since $\mathrm{P}(i)$ is true, and if $\mathrm{P}(k)$ is true then $\mathrm{P}(k+1)$ is true, $\mathrm{P}(j)$ is true for all $j \geq i$; therefore, if $\{\mathbf{A} \to \mathbf{B}\}$ and $\{\mathbf{B} \to \mathbf{A}\}$ are both valid, so are $\{\ldots\mathbf{A}\ldots \to \ldots\mathbf{B}\ldots\}$ and $\{\ldots\mathbf{B}\ldots \to \ldots\mathbf{A}\ldots\}$. □

Corollary 3.4.3 (Rule Validity) If $\{\mathbf{A} \rightarrow \mathbf{B}\}$ and $\{\mathbf{B} \rightarrow \mathbf{A}\}$ are both valid sequents, then the rule

$$\frac{S \cup \{\pm \ldots \mathbf{A} \ldots\}}{S \cup \{\pm \ldots \mathbf{B} \ldots\}}$$

preserves validity.

Proof. Assume that $\{\mathbf{A} \rightarrow \mathbf{B}\}$ and $\{\mathbf{B} \rightarrow \mathbf{A}\}$ are valid. By Theorem 3.4.2, the sequent $\{\mp \ldots \mathbf{A} \ldots, \pm \ldots \mathbf{B} \ldots\}$ is valid; by applying the thinning rule of C, we can show that $S \cup \{\mp \ldots \mathbf{A} \ldots, \pm \ldots \mathbf{B} \ldots\}$ is also valid. If $S \cup \{\pm \ldots \mathbf{A} \ldots\}$ is valid, then by an application of the cut rule of C, we can derive the sequent $S \cup \{\pm \ldots \mathbf{B} \ldots\}$; therefore it is valid as well, and the rule preserves validity. □

What the above corollary shows is that if we want to prove that each of the rules of some proof theory preserve validity when they are in the stated form, it suffices to show that the two related sequents have derivations in proof theory C. Since all of the rule schemes of the proof theory R are of the stated form, we can employ this corollary to great advantage in proving the validity of R.

Definition 3.4.4 A *constraint list* is a formula either of the form $w\alpha = w\alpha$ or of the form $\neg w\alpha = w\beta$ & $\mathbf{N}$, where β is to the right of α and $\mathbf{N}$ is a constraint list.

Definition 3.4.5 An *environment characterization* is a formula either of the form **fail** or of the form $w = \mathbf{a}$ & $\mathbf{N}$, where $\mathbf{a}$ is an environment and $\mathbf{N}$ is a constraint list.

In the sequel the symbols $\mathbf{E}$ and $\mathbf{F}$ stand for environment characterizations. Environment characterizations are so called because they completely characterize the values of all the parts of the environment variable w. In the original R^+-Maple language, the role of environment characterizations was played by identity formulae of the form $w = \mathbf{a}$. This was sufficient to describe the bindings of the environment parts because negation as failure was used to compute negated formulae; it was not necessary, therefore, to express negative information. It will turn out (see Appendix B) that every conjunction of formulae of the form $w = \mathbf{a}$ or $\neg w = \mathbf{a}$ is equivalent to a disjunction of environment characterizations, although this disjunction may be infinitely long when expanded.

We will use the symbol **fail** in the same was as in R^+-Maple, to denote the formula $w = [w, 0]$ (which is not true in any base).

Definition 3.4.6 The *De Morgan negation* of a part of an environment characterization $\mathbf{E}$, written $neg(\mathbf{E})$, is a metatheoretic transformation of the formula $\mathbf{E}$, defined as follows:

1. $neg(\mathbf{fail}) \equiv w = w$
2. $neg(w = \mathbf{a}\ \&\ \mathbf{N}) \equiv \neg w = \mathbf{a} \vee neg(\mathbf{N})$
3. $neg(\neg w\alpha = w\beta\ \&\ \mathbf{N}) \equiv w\alpha = w\beta \vee neg(\mathbf{N})$
4. $neg(w\alpha = w\alpha) \equiv \mathbf{fail}$

Theorem 3.4.7 The De Morgan negation of any environment characterization is equivalent to $\neg\mathbf{E}$; that is, the sequent $\{neg(\mathbf{E}) \rightarrow \neg\mathbf{E}\}$ is valid.

Proof. By induction, giving a derivation in C of the above sequent for each clause of the definition of $neg(\mathbf{E})$. □

Definition 3.4.8 The *quantifier dischargement* of a part of an environment characterization $\mathbf{E}$, written $dis(\mathbf{E})$, is a metatheoretic transformation of the formula $\mathbf{E}$, defined as follows:

1. $dis(\mathbf{fail}) \equiv \mathbf{fail}$
2. $dis(w = [\mathbf{a}, \mathbf{b}(wt)]\ \&\ \mathbf{N}) \equiv w = \mathbf{b}\ \&\ dis(\mathbf{N})$
3. $dis(\neg wt\alpha = wt\beta\ \&\ \mathbf{N}) \equiv \neg w\alpha = w\beta\ \&\ dis(\mathbf{N})$
4. $dis(\neg wh\alpha = w\beta\ \&\ \mathbf{N}) \equiv dis(\mathbf{N})$
5. $dis(wt\alpha = wt\alpha) \equiv w\alpha = w\alpha$

Theorem 3.4.9 The quantifier dischargement of any environment characterization is equivalent to $\exists\mathbf{E}$; that is, the sequent $\{dis(\mathbf{E}) \rightarrow \exists\mathbf{E}\}$ is valid.

Proof. By induction, giving a derivation in C of the above sequent for each clause of the definition of $dis(\mathbf{E})$. □

We now move to the formal definition of the proof theory R+Π. It is similar in structure to R$^+$-Maple, but it uses formula markers in the style of [Vod85]. Two markers, $up(\mathbf{A})$ and $down(\mathbf{A})$, are introduced to mark computations descending deeper into the formula tree, and computations returning upward with partial solutions in the form of environment characterizations.

These markers can easily be introduced by extending the environment theory semantics to include them, with the entailment rules $\{\mathbf{A}\} \vdash\!\!\!\rightarrow up(\mathbf{A})$ and $\{\mathbf{A}\} \vdash\!\!\!\rightarrow down(\mathbf{A})$, and the appropriate additions to proof theory C. Alternatively, we can consider them as metatheoretic abbreviations for distingushed formulae equivalent to the marked formula $\mathbf{A}$; for instance, $down(\mathbf{A}) \equiv (\mathrm{p}_1 = \mathrm{p}_1)\ \&\ \mathbf{A}$, $up(\mathbf{A}) \equiv (\mathrm{p}_2 = \mathrm{p}_2)\ \&\ \mathbf{A}$. We have the following theorem:

Theorem 3.4.10 If **p** and **q** are parameters, and $\{\mathbf{A} \rightarrow \mathbf{B}\}$ and $\{\mathbf{B} \rightarrow \mathbf{A}\}$ are both valid sequents, then the rule

$$\frac{S \cup \{\pm \ldots \mathbf{p} = \mathbf{p}\ \&\ \mathbf{A} \ldots\}}{S \cup \{\pm \ldots \mathbf{q} = \mathbf{q}\ \&\ \mathbf{B} \ldots\}}$$

preserves validity.

Proof. Similar to that of Theorem 3.4.2.

This theorem effectively states that we can ignore the markers for the purposes of proving rule validity; that is, that they are truly markers which will have significance only to the flow of control of the computation.

While the proof is considered to proceed from axioms to conclusions, the course of the computation is considered to proceed from the bottom of the derivation to the top. We can consider a proof of a sequent of the form $\{\rightarrow \exists down(w = [wh, wt]\ \&\ \mathbf{A}(wh))\}$ to be a *query* which asks whether there are any bindings for the variable w which will result in the formula **A** being true.

The axioms of R+Π are the sequents of the form

1. $\{\rightarrow \exists up(\mathbf{E})\}$, or of the form
2. $\{\rightarrow \exists up(\mathbf{E} \vee \mathbf{B})\}$.

The environment characterization **E** of the axiom is the "solution" to the query (the sequent at the bottom of the proof), in the sense that it can be proven from the computed derivation of $\{\rightarrow \exists down(w = [wh, wt]\ \&\ \mathbf{A}(wh))\}$ that a derivation for $\{\mathbf{E} \rightarrow \mathbf{A}\}$ exists. Further solutions can be obtained from an axiom of the second form by computation on the backtrack formula **B**.

The rules of deduction for R+Π follow. Some rules are accompanied by proofs of validity preservation employing Corollary 3.4.3.

1. Downward:

   (a) Conjunction

$$\frac{\{\rightarrow \ldots down(\mathbf{E}\ \&\ \mathbf{A})\ \&\ \mathbf{B} \ldots\}}{\{\rightarrow \ldots down(\mathbf{E}\ \&\ (\mathbf{A}\ \&\ \mathbf{B})) \ldots\}}$$

   Validity preservation:

$$\dfrac{\dfrac{\dfrac{\{\mathbf{E}, \mathbf{A}, \mathbf{B} \rightarrow \mathbf{E}\} \qquad \dfrac{\{\mathbf{E}, \mathbf{A}, \mathbf{B} \rightarrow \mathbf{A}\}\ \{\mathbf{E}, \mathbf{A}, \mathbf{B} \rightarrow \mathbf{B}\}}{\{\mathbf{E}, \mathbf{A}, \mathbf{B} \rightarrow \mathbf{A}\ \&\ \mathbf{B}\}}}{\{\mathbf{E}, \mathbf{A}, \mathbf{B} \rightarrow \mathbf{E}\ \&\ (\mathbf{A}\ \&\ \mathbf{B})\}}}{\{\mathbf{E}\ \&\ \mathbf{A}, \mathbf{B} \rightarrow \mathbf{E}\ \&\ (\mathbf{A}\ \&\ \mathbf{B})\}}}{\{(\mathbf{E}\ \&\ \mathbf{A})\ \&\ \mathbf{B} \rightarrow \mathbf{E}\ \&\ (\mathbf{A}\ \&\ \mathbf{B})\}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathbf{E},\mathbf{A},\mathbf{B}\rightarrow\mathbf{E}\}\ \{\mathbf{E},\mathbf{A},\mathbf{B}\rightarrow\mathbf{A}\}}{\{\mathbf{E},\mathbf{A},\mathbf{B}\rightarrow\mathbf{E}\ \&\ \mathbf{A}\}}\qquad\{\mathbf{E},\mathbf{A},\mathbf{B}\rightarrow\mathbf{B}\}}{\{\mathbf{E},\mathbf{A},\mathbf{B}\rightarrow(\mathbf{E}\ \&\ \mathbf{A})\ \&\ \mathbf{B}\}}}{\{\mathbf{E},\mathbf{A}\ \&\ \mathbf{B}\rightarrow(\mathbf{E}\ \&\ \mathbf{A})\ \&\ \mathbf{B}\}}}{\{\mathbf{E}\ \&\ (\mathbf{A}\ \&\ \mathbf{B})\rightarrow(\mathbf{E}\ \&\ \mathbf{A})\ \&\ \mathbf{B}\}}$$

(b) Disjunction

$$\dfrac{\{\rightarrow\ldots down(\mathbf{E}\ \&\ \mathbf{A})\vee down(\mathbf{E}\ \&\ \mathbf{B})\ldots\}}{\{\rightarrow\ldots down(\mathbf{E}\ \&\ (\mathbf{A}\vee\mathbf{B}))\ldots\}}$$

(c) Existential

$$\dfrac{\{\rightarrow\ldots\exists down((w=[wh,\mathrm{a}(wt)]\ \&\ \mathbf{N}(wt))\ \&\ \mathbf{A})\ldots\}}{\{\rightarrow\ldots down(\mathbf{E}\ \&\ \exists\mathbf{A})\ldots\}}$$

where $\mathbf{E}$ is of the form $w=\mathrm{a}\ \&\ \mathbf{N}$

Validity preservation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{w=\mathbf{a},\mathbf{N}\rightarrow w=\mathbf{a}\}\ \{w=\mathbf{a},\mathbf{N}\rightarrow\mathbf{N}\}}{\{w=\mathbf{a},\mathbf{N}\rightarrow w=\mathbf{a}\ \&\ \mathbf{N}\}}}{\{w=\mathbf{a}\ \&\ \mathbf{N}\rightarrow w=\mathbf{a}\ \&\ \mathbf{N}\}}}{\{w=\mathbf{a}\ \&\ \mathbf{N},\mathbf{A}(\mathrm{b})\rightarrow w=\mathbf{a}\ \&\ \mathbf{N}\}}\qquad\{(w=\mathbf{a}\ \&\ \mathbf{N}),\mathbf{A}(\mathrm{b})\rightarrow\mathbf{A}(\mathrm{b})\}}{\{w=\mathbf{a}\ \&\ \mathbf{N},\mathbf{A}(\mathrm{b})\rightarrow(w=\mathbf{a}\ \&\ \mathbf{N})\ \&\ \mathbf{A}(\mathrm{b})\}}}{\text{some pairing, identity, and cut rule applications}}}{\{w=\mathbf{a}\ \&\ \mathbf{N},\mathbf{A}(\mathrm{b})\rightarrow((!b=[\mathrm{b}h,\mathrm{a}(\mathrm{b}t)]\ \&\ \mathbf{N}(\mathrm{b}t))\ \&\ \mathbf{A}(\mathrm{b}))\}}}{\{w=\mathbf{a}\ \&\ \mathbf{N},\mathbf{A}(\mathrm{b})\rightarrow\exists((w=[wh,\mathrm{a}(wt)]\ \&\ \mathbf{N}(wt))\ \&\ \mathbf{A})\}}}{\{w=\mathbf{a}\ \&\ \mathbf{N},\exists\mathbf{A}\rightarrow\exists((w=[wh,\mathrm{a}(wt)]\ \&\ \mathbf{N}(wt))\ \&\ \mathbf{A})\}}}{\{(w=\mathbf{a}\ \&\ \mathbf{N})\ \&\ \exists\mathbf{A}\rightarrow\exists((w=[wh,\mathrm{a}(wt)]\ \&\ \mathbf{N}(wt))\ \&\ \mathbf{A})\}}$$

where $\mathbf{b}\equiv[\mathbf{p},w]$

(The proof of the inverse is similar.)

(d) Universal

$$\dfrac{\{\rightarrow\ldots\mathbf{E}\ \&\ \forall\neg down((w=[wh,\mathrm{a}(wt)]\ \&\ \mathbf{N}(wt))\ \&\ \neg\mathbf{A})\ldots\}}{\{\rightarrow\ldots down(\mathbf{E}\ \&\ \forall\mathbf{A})\ldots\}}$$

where $\mathbf{E}$ is of the form $w=\mathrm{a}\ \&\ \mathbf{N}$

(e) Predicate call

$$\frac{\{\rightarrow \ldots \exists down((w = [\mathbf{b}(wt), \mathbf{a}(wt)] \;\&\; \mathbf{N}(wt)) \;\&\; \mathbf{A}(wh)) \ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \mathbf{P}(\mathbf{b})) \ldots\}}$$

where $\mathbf{P}(w) \leftrightarrow \mathbf{A}$ is in Π and $\mathbf{E} \equiv (w = \mathbf{a} \;\&\; \mathbf{N})$

2. Environment Solution:

$$\frac{\{\rightarrow \ldots up(\mathbf{B}) \ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \mathbf{A}) \ldots\}}$$

where $\mathbf{A}$ is of the form $\mathbf{a} = \mathbf{b}$ or of the form $\neg\mathbf{a} = \mathbf{b}$, and $\mathbf{B}$ is the result of the Environment Solution Algorithm (see appendix B) applied to the formula $\mathbf{E} \;\&\; \mathbf{A}$, and there are no subformulae of the form $up(\mathbf{C})$ in the conclusion.

3. Upward normalization:

$$\frac{\{\rightarrow \ldots up(\mathbf{B}) \vee \mathbf{A} \ldots\}}{\{\rightarrow \ldots \mathbf{A} \vee up(\mathbf{B}) \ldots\}}$$

4. Failure:

(a) Conjunction

$$\frac{\{\rightarrow \ldots up(\mathbf{fail}) \ldots\}}{\{\rightarrow \ldots up(\mathbf{fail}) \;\&\; \mathbf{A} \ldots\}}$$

(b) Disjunction

$$\frac{\{\rightarrow \ldots down(\mathbf{A}) \ldots\}}{\{\rightarrow \ldots up(\mathbf{fail}) \vee \mathbf{A} \ldots\}}$$

Validity preservation:

$$\frac{\{\mathbf{A} \rightarrow \mathbf{fail}, \mathbf{A}\}}{\{\mathbf{A} \rightarrow \mathbf{fail} \vee \mathbf{A}\}}$$

$$\frac{\dfrac{\{wh = w, wt = 0 \rightarrow \mathbf{A}, w = 0\}}{\{w = [w, 0] \rightarrow \mathbf{A}\}} \quad \{\mathbf{A} \rightarrow \mathbf{A}\}}{\{w = [w, 0] \vee \mathbf{A} \rightarrow \mathbf{A}\}}$$

(c) Existential

$$\frac{\{\rightarrow \ldots up(\mathbf{fail}) \ldots\}}{\{\rightarrow \ldots \exists up(\mathbf{fail}) \ldots\}}$$

(d) Universal

$$\frac{\{\rightarrow \ldots up(\mathbf{E}) \ldots\}}{\{\rightarrow \ldots \mathbf{E} \;\&\; \forall \neg up(\mathbf{fail}) \ldots\}}$$

5. Success, no backtrack:

(a) Conjunction

$$\frac{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \mathbf{B}) \ldots\}}{\{\rightarrow \ldots up(\mathbf{E}) \;\&\; \mathbf{B} \ldots\}}$$

(b) Disjunction

$$\frac{\{\rightarrow \ldots up(\mathbf{E} \vee \mathbf{B}) \ldots\}}{\{\rightarrow \ldots up(\mathbf{E}) \vee \mathbf{B} \ldots\}}$$

(c) Existential

$$\frac{\{\rightarrow \ldots up(\mathbf{F}) \ldots\}}{\{\rightarrow \ldots \exists up(\mathbf{E}) \ldots\}}$$

where $\mathbf{F}$ is the quantifier dischargement of $\mathbf{E}$

(d) Universal

$$\frac{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \mathbf{G}) \ldots\}}{\{\rightarrow \ldots \mathbf{E} \;\&\; \forall \neg up(\mathbf{F}) \ldots\}}$$

where $\mathbf{G}$ is the De Morgan negation of the quantifier dischargement of $\mathbf{F}$

6. Success with backtrack:

(a) Conjunction

$$\frac{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \mathbf{A}) \vee (\mathbf{C} \;\&\; \mathbf{A}) \ldots\}}{\{\rightarrow \ldots up(\mathbf{E} \vee \mathbf{C}) \;\&\; \mathbf{A} \ldots\}}$$

(b) Disjunction

$$\frac{\{\rightarrow \ldots up(\mathbf{E} \vee (\mathbf{C} \vee \mathbf{A})) \ldots\}}{\{\rightarrow \ldots up(\mathbf{E} \vee \mathbf{C}) \vee \mathbf{A} \ldots\}}$$

(c) Existential

$$\frac{\{\rightarrow \ldots up(\mathbf{F} \vee \exists \mathbf{C}) \ldots\}}{\{\rightarrow \ldots \exists up(\mathbf{E} \vee \mathbf{C}) \ldots\}}$$

where $\mathbf{F}$ is the quantifier dischargement of $\mathbf{E}$

(d) Universal

$$\frac{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \mathbf{G}) \;\&\; \forall \neg \mathbf{C} \ldots\}}{\{\rightarrow \ldots \mathbf{E} \;\&\; \forall \neg up(\mathbf{F} \vee \mathbf{C}) \ldots\}}$$

where $\mathbf{G}$ is the De Morgan negation of the quantifier dischargement of $\mathbf{F}$

Validity preservation:

$$\dfrac{\dfrac{\{\mathbf{F}([\mathbf{p},w]) \rightarrow \mathbf{F}([\mathbf{p},w])\}}{\{\mathbf{F}([\mathbf{p},w]) \rightarrow \exists\mathbf{F}\}}}{\{\forall\neg\mathbf{C}, \mathbf{F}([\mathbf{p},w]) \rightarrow \exists\mathbf{F}\}} \quad \dfrac{\dfrac{\{\mathbf{C}([\mathbf{p},w]) \rightarrow \mathbf{C}([\mathbf{p},w])\}}{\{\neg\mathbf{C}([\mathbf{p},w]), \mathbf{C}([\mathbf{p},w]) \rightarrow\}}}{\{\forall\neg\mathbf{C}, \mathbf{C}([\mathbf{p},w]) \rightarrow\}}}{\{\forall\neg\mathbf{C}, \mathbf{C}([\mathbf{p},w]) \rightarrow \exists\mathbf{F}\}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\forall\neg\mathbf{C}, (\mathbf{F}([\mathbf{p},w]) \vee \mathbf{C}([\mathbf{p},w])) \rightarrow \exists\mathbf{F}\}}{\{\forall\neg\mathbf{C} \rightarrow \neg(\mathbf{F}([\mathbf{p},w]) \vee \mathbf{C}([\mathbf{p},w])), \exists\mathbf{F}\}}}{\{\forall\neg\mathbf{C} \rightarrow \forall\neg(\mathbf{F} \vee \mathbf{C}), \exists\mathbf{F}\}}}{\{\mathbf{E}, \forall\neg\mathbf{C} \rightarrow \forall\neg(\mathbf{F} \vee \mathbf{C}), \exists\mathbf{F}\}}}{\{\mathbf{E}, \forall\neg\mathbf{C} \rightarrow \mathbf{E} \,\&\, \forall\neg(\mathbf{F} \vee \mathbf{C}), \exists\mathbf{F}\}}}{\{\mathbf{E}, \neg\exists\mathbf{F}, \forall\neg\mathbf{C} \rightarrow \mathbf{E} \,\&\, \forall\neg(\mathbf{F} \vee \mathbf{C})\}}}{\{\mathbf{E} \,\&\, \neg\exists\mathbf{F}, \forall\neg\mathbf{C} \rightarrow \mathbf{E} \,\&\, \forall\neg(\mathbf{F} \vee \mathbf{C})\}}}{\{(\mathbf{E} \,\&\, \neg\exists\mathbf{F}) \,\&\, \forall\neg\mathbf{C} \rightarrow \mathbf{E} \,\&\, \forall\neg(\mathbf{F} \vee \mathbf{C})\}}$$

(The proof of the inverse is similar.)

7. Negation:

   (a) Conjunction

$$\dfrac{\{\rightarrow \ldots down(\mathbf{E} \,\&\, (\neg\mathbf{A} \vee \neg\mathbf{B}))\ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \neg(\mathbf{A} \,\&\, \mathbf{B}))\ldots\}}$$

   (b) Disjunction

$$\dfrac{\{\rightarrow \ldots down(\mathbf{E} \,\&\, (\neg\mathbf{A} \,\&\, \neg\mathbf{B}))\ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \neg(\mathbf{A} \vee \mathbf{B}))\ldots\}}$$

   (c) Double negation

$$\dfrac{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \mathbf{A})\ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \neg\neg\mathbf{A})\ldots\}}$$

   (d) Existential

$$\dfrac{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \forall\neg\mathbf{A})\ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \neg\exists\mathbf{A})\ldots\}}$$

   (e) Universal

$$\dfrac{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \exists\neg\mathbf{A})\ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \neg\forall\mathbf{A})\ldots\}}$$

(f) Predicate call

$$\frac{\{\rightarrow \ldots \mathbf{E} \;\&\; \forall \neg down((w = [\mathbf{b}(wt), \mathbf{a}(wt)] \;\&\; \mathbf{N}(wt)) \;\&\; \mathbf{A}(wh)) \ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \;\&\; \neg \mathbf{P}(\mathbf{b})) \ldots\}}$$

where $\mathbf{P}(w) \leftrightarrow \mathbf{A}$ is in Π and $\mathbf{E} \equiv (w = \mathbf{a} \;\&\; \mathbf{N})$

A brief description of the computation may be in order. A stream of computation "descends" into the formula tree where a *down* marker appears. If the formula to be processed is a conjunction, the computation descends into the left-hand conjunct; if it is a disjunction, the computation "splits" into two separate streams, each marked; if it is a quantified formula, the computation moves inside the quantifier; if it is a predicate call, the appropriate predicate body is substituted; and if it is a negated formula, the negation is pushed down farther by De Morgan's laws.

When a leaf node in the formula tree (a subformula of the form $\mathbf{a} = \mathbf{b}$ or $\neg \mathbf{a} = \mathbf{b}$) is reached, the environment solution algorithm (a generalization of unification) is applied, and the partial solution obtained begins to "ascend" the formula tree again, moving back through any quantifiers it left behind. For simplicity's sake, only one stream of computation is allowed to be ascending at a time, hence the restriction on the environment solution rule. When an stream ascends to the level of a disjunction, the formula is normalized by making and ascending stream the left-hand one. If at any time in its ascent the formula meets a conjunction (whose computation was previously suspended), it descends into the conjunct; otherwise, it ascends all the way to the top level, effectively reporting a solution to the original query.

The proof theory R is not fully deterministic, because in any computation there can be several subformulae with *down* markers, where computation can take place. However, at each *down* marker, there is only one rule which can possibly apply to that marked subformula. The next chapter describes a fully deterministic proof theory which is equivalent to R.

R is *precomplete* with respect to the given semantics in the following informal sense. For any program Π and most sentences $\mathbf{A}$, if the sequent $\{\rightarrow \exists \mathbf{A}(wh)\}$ is derivable in C+Π, then the equivalent sequent $\{\rightarrow \exists down(w = w \;\&\; \mathbf{A}(wh))\}$ is derivable in R+Π. However, sentences $\mathbf{A}$ of the form $\mathbf{P}(\mathbf{a}) \vee \neg \mathbf{P}(\mathbf{a})$, where the definition of $\mathbf{P}$ causes computation of $\mathbf{P}(\mathbf{a})$ to diverge, do not have this property, although they clearly receive the + sign in all bases.

This behaviour of R suggests that it could be equivalent to a proof theory having some restricted form of excluded middle. The excluded middle of intuitionistic theories seems too weak; there are some theorems of R which would not be theorems of an intuitionistic system, such as

$$\{\rightarrow \exists down(w = [wh, wt] \;\&\; \forall(wh = 0 \vee \neg wh = 0))$$

However, we cannot have full classical excluded middle, due to the unprovability in R of the sentences noted above.

The search for an equivalent classical-style proof theory could be restated as the search for a reductionist semantics with respect to which R is a consistent and complete proof theory. By "reductionist" we here mean a semantics which assigns truth value to individual atomic sentences, and to non-atomic sentences based solely on the truth values of their constituent subformulae. As we shall see in the next chapter, R, rather than variants of R employing left-to-right disjunction or negation as failure, seems more likely to yield a solution to this search.

Chapter 4

Alternative Formulations

In this chapter, we will explore some alternative ways in which the evironment theory of the last chapter could be formulated and presented. These include formulations with a sequential evaluation of disjunction; those with negation as failure rather than precomplete negation; and those designed with greater efficiency and ease of implementation in mind.

4.1 Simulated Parallelism

The computational rules given in the last chapter assume a parallel algorithm. Computation goes on at the $down(\ldots)$ nodes in the formula tree, and when it reaches a disjunction, the computation forks; that is, where there was one $down(\ldots)$ subformula there are now two, and computation proceeds at each node.

If we are trying to develop a sequential implementation, we have two main choices. We could alter the rules for disjunction so that they are truly sequential, but as discussed in the next section, this approach has its disadvantages. If we still wish to use the parallel algorithm, however, we could simulate parallel computation by the use of the device known in recursive function theory as dovetailing.

An interpreter (universal function) simulates the evaluation of a program (the Gödel number of a partial recursive function) by processing each individual instruction in the program. Similarly, a dovetailing interpreter simulates the evaluation of two or more programs by processing an instruction from each program, then the next instruction from each program, and so on.

The dovetailing in an environment-theory computational proof theory could take place on several levels. The highest would be on the level of the operating system; this is basically what a timesharing operating system does all the time. An intermediate level would be to simulate parallelism within the interpreter, but to retain the rules and

data structures of a truly parallel computation. The lowest level would be to express the parallelism in the computational rules.

The two higher levels are not interesting from a logical point of view. However, a study of the lowest level provides some useful insights into the issues behind implementations of logic programming.

The purpose of the computational "markers" [Vod85] is to incorporate some notion of program control into the formulae being computed, and into the computational rules. To incorporate the flow of control in a dovetailing implementation, we could augment the markers with an integer indicating the current stage of computation of the subformula. Consider the proof theory RD1, defined as being identical to R except for the following modifications.

1. The marker in the axioms, instead of being of the form up, is of the form up^n, for any n. (We could call this n the *marker counter*; the markers with counters could be defined with parameters in the same way as the original markers, since we have a denumerable number of parameters.)

2. Similarly, the marker in the conclusion of each rule is of the form $down^n$ or up^n, for any n.

3. When the marker counter in the conclusion of a rule is n, the marker in the premiss of each rule is of the form $down^{n+1}$ or up^{n+1}.

4. Each rule has the additional condition that the indicated conclusion subformula $down^n(\mathbf{A})$ is the leftmost subformula with the same marker counter as the rightmost marker.

We claim that proof theory RD1 is complete in the same sense that R is; that is, that RD1+Π can prove all sequents of the form $\{\rightarrow \exists down(w = [wh, wt]\ \&\ \mathbf{A})\}$ proven by R+Π. Each branch of the computation is executed, one rule application at a time, with the rule applications going from leftmost to rightmost marker and then resuming at the leftmost marker. Of course, in an implementation the marker counters would not be necessary; each marker would have an associated data structure which pointed to the next marker to be evaluated.

Actually, it is not necessary to execute the branches in parallel at the level of individual rule applications. It suffices to stop an infinite downward sequence of evaluation resulting from a recursive predicate definition. Consider the proof theory RD2, defined as being identical to R except for the following modifications.

1. The marker in the axioms, instead of being of the form up, is of the form up^n, for any n.

2. Similarly, the marker in the conclusion of each rule is of the form $down^n$ or up^n, for any n.

3. When the marker counter in the conclusion of a rule is n, the marker in the premiss of each rule is of the form $down^n$ or up^n, *except* in the rules for predicate evaluation, in which the marker in the premiss of each rule is of the form $down^{n+1}$.

4. Each rule has the additional condition that the indicated conclusion subformula $down^n(\mathbf{A})$ is the leftmost subformula with the same marker counter as the rightmost marker.

Again, we claim that RD2 is complete in the same sense that R is. It may be possible to increase the granularity of the computation even more, by incorporating into the marker information about (for instance) what predicates have been called since the last pause in computation on the branch.

4.2 Sequential Disjunction

Many implementations of logic programming languages use a left-to-right sequential computation of disjunction; that is, if the equivalent of the formula $\mathbf{A} \vee \mathbf{B}$ is about to be decided, the computation proceeds by first deciding $\mathbf{A}$, and if all solutions from $\mathbf{A}$ lead to failure, by then deciding $\mathbf{B}$. The main reason for this implementation approach is to ensure that the exponential explosion of space required by parallel disjunction does not occur. Other reasons include the inefficiency of implementing disjunctive parallelism on the operating system level, and the difficulty of implementing a sequential simulation of parallelism by dovetailing.

Altering our proof theory R so that it uses sequential disjunction is fairly simple. The rule for downward movement into a disjunction would become:

$$\frac{\{\rightarrow \ldots down(\mathbf{E} \,\&\, \mathbf{A}) \vee \mathbf{B} \ldots\}}{\{\rightarrow \ldots down(\mathbf{E} \,\&\, (\mathbf{A} \vee \mathbf{B})) \ldots\}}$$

and the rule for failure of one branch of a disjunction would become:

$$\frac{\{\rightarrow \ldots down(\mathbf{A}) \ldots\}}{\{\rightarrow \ldots up(\mathbf{fail}) \vee \mathbf{A} \ldots\}}$$

These rules would essentially suspend computation of the right-hand disjunct until all solutions from the left-hand disjunct had led to failure.

But here we run into a semantical question: what is the real meaning of this sequential disjunction? There are several approaches we can take to an answer.

4.2.1 Unchanged Semantics

One approach is to simply retain the old semantics for environment theory. Sequential disjunction can be proven to be sound with respect to the standard model theories; the proofs of soundness of the various resolution strategies which employ it basically do this [Llo84].

Sequential disjunction is not complete with respect to the standard model theories, however. Consider the top-level goal $w = \mathbf{a}$ & $(\mathbf{A} \vee \mathbf{B})$. If $\mathbf{A}$ is a predicate with a definition that goes into infinite recursion, $\mathbf{B}$ will never be computed even if we can see intuitively that $w = \mathbf{a}$ & $\mathbf{B}$ can be proven.

We will not get all possible solutions from a sequential disjunction, and in fact we may not get any, even when solutions exist. The same can be said for any formula which contains a disjunction or a call to a predicate which contains a disjunction. This is a significant divergence from the accepted semantics of disjunction, but it can be argued that this lack of completeness is unimportant. As long as we are always proving correct things, and the set of things we can prove is sufficiently large for our purposes, why do we need completeness?

On the other hand, the above argument could be used to question the need for a semantics in the first place; if we need only validity but not completeness, then surely our definition of truth can become arbitrarily trivial, mimicking the computation and declaring everything computed to be true as true. But if the standard semantics describes accurately our intuitive notion of truth and meaning, why then must our computer systems implement a significantly reduced notion of truth and meaning? What, in fact, is the notion of truth and meaning implemented by sequential disjunction?

Most logic programming theorists have not dealt with this question. Its answer requires modifications to the semantics of the programming system.

4.2.2 Modified Semantics for Disjunction

A first stab at the answer is provided by a modification to the semantics for disjunction. We want the truth value of a disjunction to remain the same for the case when the left-hand disjunct is defined, but to be undefined (in the three-valued logic analogue, to receive the ? sign) when the left-hand disjunct is undefined, regardless of the value of the right-hand disjunct.

Let us call this model theory SD, for "sequential disjunction". The entailment rules for the other connectives remain the same. The truth table for disjunction becomes the following.

$\vee$	$+$	$-$	?
$+$	$+$	$+$	$+$
$-$	$+$	$-$	?
?	?	?	?

(Compare with that in the section on Formal Semantics, above.) The rules for entailment involving disjunction become:
$\{+\mathbf{A}\} \vdash\!\!\!\rightarrow + (\mathbf{A} \vee \mathbf{B})$
$\{-\mathbf{A}, \pm\mathbf{B}\} \vdash\!\!\!\rightarrow \pm (\mathbf{A} \vee \mathbf{B})$
Proof theory C can be modified in a similar manner in order to retain its completeness. But have we acheived completeness for proof theory R? We have for the propositional case, but lose it when we introduce quantifiers.

For example, the sequent $\{\rightarrow \exists((\neg wh = 0 \;\&\; \mathbf{P}(w)) \vee (wh = 0))\}$ is valid in SD due to the validity of the sequent $\{\rightarrow (\neg 0 = 0 \;\&\; \mathbf{P}([0, w])) \vee (0 = 0)\}$, but if we compute with it by the rules for existential quantifier, and the predicate call $\mathbf{P}(w)$ goes into infinite recursion, it will never find the answer – exactly the situation in which we wanted the truth value to be indeterminate.

The reason for this discrepancy becomes clearer when we note that the existential quantifier is in some sense an infinite disjunction. $\exists\mathbf{A}$ is essentially stating the same thing as $\mathbf{A}([\mathbf{a}_1, w]) \vee \mathbf{A}([\mathbf{a}_2, w]) \vee \mathbf{A}([\mathbf{a}_3, w]) \vee \ldots$, where $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \ldots$ is the enumeration of all the terms and where the disjunction is computed *in parallel.* We have succeeded in modelling the sequential nature of the explicit, propositional disjunction, but not the sequential algorithm we must employ when computing the implicit disjunction of the quantifiers. If we still want to achieve completeness, it seems we must modify the semantics of the quantifiers as well.

4.2.3 Modified Semantics for Disjunction and Quantifiers

Unfortunately, when we try to develop a consistent truth definition for the existential quantifier, we run into problems almost immediately. Recall that no formula which contains a disjunction is guaranteed to yield all solutions under sequential disjunction. We can define the truth value of $\exists(\mathbf{A} \vee \mathbf{B})$ as being the same as the value of $\exists\mathbf{A} \vee \exists\mathbf{B}$, and the value of $\exists\mathbf{A}$, where $\mathbf{A}$ is an atomic formula, as before. But $\exists(\mathbf{A} \;\&\; \mathbf{B})$ cannot be defined as easily. Such a sentence will be transformed into some disjunction $\exists(\mathbf{C} \vee \mathbf{D})$ by the computation algorithm, and we must know what disjunction it is to define its truth value.

In other words, since we cannot use proof-theoretic constructs in the semantics (*e.g.,* by referring to the computability of a formula), it seems we must mimic the computation in the definition of the existential quantifier.

The same argument applies for the universal quantifier, since its definition in the

Gilmore semantics depends on an infinite conjunction which is transformed to a disjunction in the computation rules. We can make the following conjecture in fairly informal language:

Conjecture 4.2.1 If M is a semantics of a computational logic which contains the standard connectives and quantifiers, and P is a deterministic proof theory of that computational logic which uses sequential disjunction, and P is complete with respect to M, then the definition of truth in M mimics the computations in P. In other words, M is a trivial semantic characterization of the proof theory P.

We cannot, at this point, prove the above conjecture (partly because its language is not sufficiently rigourous). We feel that another promising area for further research will be to express this conjecture more rigorously and prove it. We must make decisions on the design of our logic programming systems based on the proof of this conjecture: do we wish them to be complete only with respect to a trivial semantics, or should we have the goal of making them as complete as possible with respect to the fairly intuitive and well-accepted truth-functional semantics of mathematical logic?

We make the second choice, and use parallel disjunction rather than sequential. The fact that parallel disjunction can be sequentially simulated using dovetailing (simulated parallelism) is another factor in favour of making this choice.

4.3 Parallel Conjunction

If the use of left-to-right sequential disjunction creates such a wide gap between proof theory and any non-trivial semantics, why then does the proof theory R use sequential conjunction? The answer lies in what we want to use the proof theory for.

In logic programming, we are interested mainly in the use of the proof theory for generating sets of variable bindings which make a query true. We are not necessarily interested in using the proof theory to refute things.

When sequential disjunction is used, the system may not be able to find solutions to a query of the form $\mathbf{A} \vee \mathbf{B}$ when solutions to **B** exist. When sequential conjunction is used, on the other hand, the processing of a query of the form $\mathbf{A} \,\&\, \mathbf{B}$ must always result in solutions if they exist, because both **A** and **B** must be true in the same base. The system will be unable to prove that a query of the form $\mathbf{A} \,\&\, \mathbf{B}$ is unsatisfiable in general, although many unsatisfiable queries will be able to be so proven.

Although a parallel conjunction would be desirable, we conclude that it is not necessary. The good reasons for using sequential disjunction – lessening of the exponential explosion of space usage and simplicity of algorithm – also hold in the case of sequential conjunction.

4.4 Negation as Failure

The standard method of computing negation is the general paradigm known as negation as failure [Cla78]. No negation as failure is complete with respect to classical semantics, but certain types have been proved to be sound [Llo84]; indeed, a great deal of recent research has gone into such soundness proofs.

In Voda's paper describing environments [Vod86b], the language described uses a version of negation as failure. When a computation moves into a negation, it does so essentially by the following rule:

$$\frac{\{\rightarrow \ldots w = \mathbf{a} \;\&\; \neg down(w = \mathbf{a} \;\&\; \mathbf{A}) \ldots\}}{\{\rightarrow \ldots down(w = \mathbf{a} \;\&\; \neg \mathbf{A}) \ldots\}}$$

But if the sentence is then transformed by the computation into an equivalent one of the form $w = \mathbf{a} \;\&\; \neg w = \mathbf{b}$, where $\mathbf{b}$ is different from $\mathbf{a}$, the computation blocks. Conditions on the environment are expressed by a single environment equation of the form $w = \mathbf{a}$, and therefore cannot express the conjunctive condition.

The analogue in Prolog is the situation when a computation moves into a negated formula without all terms in the formula ground, *i.e.*, assigned a value. If any bindings take place within the evaluation of the formula, the bindings cannot be passed through the negation, or the computation will become unsound, so the computation must block, reporting that the goal formula can be neither proven nor refuted.

The method of preference for avoiding this problem is delayed negation; see for instance [CG83]. The computation of a negation is delayed until such time as all the terms in the negated formula have become ground due to results from other conjunctive subgoals. Once all terms are ground, the computation of negation as failure can be proven to be both sound and complete.

The obvious difficulty with this approach is that if the negated formula is not part of a conjunction, there are no other conjunctive subgoals to produce values, and thus there is no way to ground the terms in the negated formula. Lloyd [Llo84] ignores this case. However, there is one important situation where it arises consistently; namely, when the universal quantifier is used or simulated.

Consider the standard first-order logic formula $\mathbf{P}(x) \;\&\; \forall y \mathbf{Q}(x, y)$. Since Prolog has no explicit quantifiers, but only implicit universal quantifiers at the beginnings of predicate definitions, we must express such a formula in a different way if we are to compute it. First we note that the formula is equivalent to $\mathbf{P}(x) \;\&\; \neg\exists y \neg \mathbf{Q}(x, y)$, and then note that the only way to obtain an existential quantifier in that position is to define a predicate containing the formula being quantified. Thus, the formula must be computed in Prolog by defining the predicate $\mathbf{R}$ by the clause
$\mathbf{R}(x) :- \neg \mathbf{Q}(x, y)$.

and deciding the goal
$\mathbf{P}(x), \neg\mathbf{R}(x)$.

Clearly, the body of **R** is exactly the type of formula which is not decidable by negation as failure. Since this situation arises every time a universal quantifier is encountered, we must conclude that it is not feasible to have full universal quantification in a system with negation as failure.

In addition to the conflict with universal quantification, there are difficulties in defining the semantics of negation as failure. Here we run into many of the same problems as we did with sequential disjunction. Let us assume that we want our negation as failure rule to be complete with respect to the semantics. We want the formula $w = \mathbf{a}$ & $\neg$**fail** to take the same sign as the formula $w = \mathbf{a}$, and the formula $w = \mathbf{a}$ & $\neg w = \mathbf{a}$ to take the sign $-$ (the cases decided by negation as failure). However, we want the formula $w = \mathbf{a}$ & $\neg w = \mathbf{b}$ to take the sign ? (*i.e.*, to be of undefined truth value) when $w = \mathbf{a} \not\rightarrow w = \mathbf{b}$, since this is the case negation as failure cannot decide.

Again, the truth value of the formula in question depends on the surrounding formulae. But what, then, is the truth value of a formula of the form **A** & $\neg$**B**? It depends on the order of the solutions which are computed from the individual formulae **A** and **B**; that is, it depends on things which we cannot deduce without doing the computation. With either true parallel disjunction, simulated parallel disjunction, or left-to-right sequential disjunction, the order of solutions obtained cannot be predicted. We expect a similar result to that for sequential disjunction will obtain; namely, that negation as failure is a complete procedure only with respect to a trivial semantics.

Chapter 5

Conclusions and Future Work

We have shown that the semantics of Voda's Theory of Pairs can be expressed formally by a system similar to Gilmore's natural deduction first-order set theory. Using Voda's technique of describing programming languages as proof theories, we have given a language which can prove a large subset of possible logic programming queries. While not complete with respect to the given semantics, this language is precomplete in some well-defined sense, mainly due to its improved treatment of negation. To show the soundness of the language with respect to the semantics, we have used the complete natural deduction proof theory corresponding to the semantics as an effective analytic tool.

Future work arising from this study covers a fairly wide area. Theoretically, the goal of studying logic programming formally is to develop improved logic programming languages; it would be desirable to implement the language described here. An attempt to implement the language may expose inconsistencies in the definition, and may lead to an improved language which corresponds better to actual computation. This line of research follows along the same path as previous work by Voda: the attempt to describe, in standard logic, the data and control structures used in the interpreter of a programming language.

We are not entirely satisfied with our definition of environment characterization and the environment solution algorithm. We feel that a less complex notation may be obtainable, which expresses the negative information of the characterization as compactly as the environment expresses positive information. It would also be nice if the form of environment characterization so obtained had the property that the environment solution algorithm on it terminated on all failures as well as on all successes. The key requirement for an environment characterization, however, is that it can pass outside a quantifier easily, discharging the quantified variable with a minimum of computational effort.

The semantic characterizability of left-to-right sequential disjunction and negation as failure is also an interesting area. If we cannot give a semantics with respect to which a language using these techniques is complete, other than a trivial one, any assertion that such a language corresponds to formal logic is considerably weakened. An analysis of the issues here would seem to demand a formal definition of a semantics, and also definitions of the acceptability and triviality of a semantics. It is not clear at this stage what these definitions would look like.

Finally, we have a strong but, for now, intuitive feeling that there is a connection between logic programming with precomplete negation and such set theories as Gilmore's, with its form of excluded middle. To be able to prove the equivalence of a precomplete logic programming language with some formulation of predicate calculus with restricted excluded middle may be a result of some theoretical importance.

Bibliography

[AvE82] Krzysztof R. Apt and Maarten H. van Emden. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery*, 29(3):841–862, July 1982.

[Bet66] Evert W. Beth. *Foundations of Mathematics.* Harper and Row, New York, 1966.

[CG83] K. L. Clark and S. Gregory. *PARLOG: A Parallel Logic Programming Language.* Technical Report DOC 83/5, Department of Computing, Imperial College, London, 1983.

[Cla78] Keith L. Clark. *Negation as Failure*, pages 293–322. Plenum Press, New York, 1978.

[Fit85] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 4:295–312, 1985.

[Gen69] Gerhardt Gentzen. *The Collected Papers of Gerhard Gentzen.* North-Holland, Amsterdam, 1969.

[Gil86a] Paul C. Gilmore. *Computer Science 504 Course Notes.* University of B. C. Department of Computer Science, Vancouver, B. C., 1986.

[Gil86b] Paul C. Gilmore. Natural deduction based set theories: a new resolution of the old paradoxes. *Journal of Symbolic Logic*, 51(2):393–411, June 1986.

[HA38] David Hilbert and W. Ackermann. *Grundzüge der Theoretischen Logik.* Springer, Berlin, 1938.

[Kow74] Robert Kowalski. Predicate logic as programming language. In *Information Processing 74 - Proceedings of the IFIP Conference*, North-Holland, 1974.

[Kri75] Saul Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[Llo84] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[Men64] Elliott Mendelson. *Introduction to Mathematical Logic.* D. van Nostrand, Princeton, 1964.

[Nai84] Lee Naish. *MU-Prolog 3.1db Reference Manual.* University of Melbourne, 1984.

[Sco75] Dana Scott. *Combinators and Classes*, pages 1–26. Volume 37 of *Lecture Notes in Computer Science*, Springer-Verlag, 1975.

[Smu68] Raymond M. Smullyan. *First-Order Logic.* Springer-Verlag, Berlin, 1968.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–742, October 1976.

[Vod84] Paul J. Voda. *Theory of Pairs, Part I: Provably Recursive Functions.* Technical Report 84-25, Department of Computer Science, University of British Columbia, Vancouver, December 1984.

[Vod85] Paul J. Voda. A view of programming languages as symbiosis of meaning and computations. *New Generation Computing*, 3:71–100, 1985.

[Vod86a] Paul J. Voda. Choices in, and limitations of, logic programming. In *Proceedings of the Third International Logic Programming Conference*, 1986.

[Vod86b] Paul J. Voda. Computation of full logic programs using one-variable environments. *New Generation Computing*, 4(2), 1986.

[Vod86c] Paul J. Voda. *Precomplete Negation and Universal Quantification.* Technical Report 86-9, Department of Computer Science, University of British Columbia, Vancouver, April 1986.

Appendix A

Completeness Proof for C

In the body of this appendix, we will assume that the proof theory referred to is always the proof theory C described above, augmented by the rules concerning the program Π. That is, "derivability" will mean "derivability in C+Π", etc. We will also assume that the signed sentences of the theory can be enumerated with the enumeration $\sigma_1, \sigma_2, \sigma_3, \ldots$

Theorem A..1 (Subset) A sequent is derivable if and only if some finite subset of it is derivable.

Proof. ($\leftarrow$) If a finite subset of a sequent is derivable, we can construct a derivation of the whole sequent by simply augmenting every sequent in the proof by the remainder of the sequent. Clearly this derivation will be correct.

($\rightarrow$) We can construct one finite derivable subset from the derivation of any sequent. For an axiom, the subset is the one or two formulae required to be in it by the definition of axiom. Assume that for all sequents derivable with n steps, we can build such a subset.

Consider any sequent derivable with $n + 1$ steps. Let the premisses be of the form $Seq \cup S_1, \ldots Seq \cup S_n$, and let the conclusion be of the form $Seq \cup \{\mathbf{A}\}$. If $T_1, \ldots T_n$ are the constructed finite derivable subsets of the premisses, then clearly $(T_1 \cup \ldots \cup T_n \cup \{\mathbf{A}\}) - (S_1 \cup \ldots \cup S_n)$ is also derivable.

A finite derivable subset of any derivable sequent can thus be built up by induction. □

Definition A..2 The *parameter set* of a sequent is the set of all parameters in all the terms appearing in that sequent.

Intuitively, we will prove the completeness of C by proving that any non-derivable sequent must not be valid. We will do this by extending the sequent by the addition of

formulae, forming a chain of non-derivable sequents whose union will be the foundation for a base in which the sequent is not true. This chain will be one path through the *Ext* tree, defined as follows.

Definition A..3 The sets $Ext_n(Seq)$ and $Chk_n(Seq)$, and the formula $AtBat_n(Seq)$, for any sequent Seq with a finite parameter set, form an infinite trinary tree in which every node n has nodes $3n-1$, $3n$, and $3n+1$ as children. These constructs can be thought of as the "extensions" of Seq, the "check sets" corresponding to each extension, and the "formula at bat" in each extension. Their definition is mutually recursive.

1. $Ext_1(Seq)$ is Seq.
2. $Chk_1(Seq)$ is the empty set, $\{\}$.
3. If there are members of $Ext_n(Seq)$ which are not members of $Chk_n(Seq)$, $AtBat_n(Seq)$ is the earliest such signed sentence in the enumeration sequence $\sigma_1, \sigma_2, \ldots$ Otherwise, $AtBat_n(Seq)$ is undefined.
4. For $n \geq 1$, if $AtBat_n(Seq)$ is defined, then for $3n-1 \leq m \leq 3n+1$, $Ext_m(Seq)$ is $Ext_n(Seq) \cup \Delta(Ext_m(Seq))$. $\Delta(Ext_m(Seq))$ is defined by the following table.

$AtBat_n(Seq)$	$\Delta(Ext_{3n-1}(Seq))$	$\Delta(Ext_{3n}(Seq))$	$\Delta(Ext_{3n+1}(Seq))$
$+\mathbf{a} = [\mathbf{b}, \mathbf{c}]$ $+[\mathbf{b}, \mathbf{c}] = \mathbf{a}$	$\{+\mathbf{a}h = \mathbf{b}\}$	$\{+\mathbf{a}t = \mathbf{c}\}$	$\{-\mathbf{a} = 0\}$
$-\mathbf{a} = [\mathbf{b}, \mathbf{c}]$ $-[\mathbf{b}, \mathbf{c}] = \mathbf{a}$	$\{-\mathbf{a}h = \mathbf{b}, -\mathbf{a}t = \mathbf{c}, +\mathbf{a} = 0\}$		
$\pm\mathbf{a} = \mathbf{b}$ (not pairs)	$\{\}$		
$+\mathbf{A}\ \&\ \mathbf{B}$	$\{+\mathbf{A}\}$	$\{+\mathbf{B}\}$	
$-\mathbf{A}\ \&\ \mathbf{B}$	$\{-\mathbf{A}, -\mathbf{B}\}$		
$+\mathbf{A} \vee \mathbf{B}$	$\{+\mathbf{A}, +\mathbf{B}\}$		
$-\mathbf{A} \vee \mathbf{B}$	$\{-\mathbf{A}\}$	$\{-\mathbf{B}\}$	
$\pm\neg\mathbf{A}$	$\{\mp\mathbf{A}\}$		
$+\exists\mathbf{A}$	$\{+\mathbf{A}([\mathbf{a}_1^n, w]), +\mathbf{A}([\mathbf{a}_2^n, w]), \ldots\}^{(a)}$		
$-\exists\mathbf{A}$	$\{-\mathbf{A}([\mathbf{p}^n, w])\}^{(b)}$		
$+\forall\mathbf{A}$	$\{+\mathbf{A}([\mathbf{p}^n, w])\}^{(b)}$		
$-\forall\mathbf{A}$	$\{-\mathbf{A}([\mathbf{a}_1^n, w]), -\mathbf{A}([\mathbf{a}_2^n, w]), \ldots\}^{(a)}$		
$\pm\mathbf{P}(\mathbf{a})$	$\{\pm\mathbf{A}^{\mathbf{P}}(\mathbf{a})\}^{(c)}$		

Notes:

(a) $\mathbf{a}_1^n, \mathbf{a}_2^n, \ldots$ is the enumeration of all the terms generated from the parameter set of $Ext_n(Seq)$.

(b) $\mathbf{p}^n$ is a parameter not appearing in $Ext_n(Seq)$.

(c) $\mathbf{A^P}$ is the definition of predicate $\mathbf{P}$ in the program Π; *i.e.*, $\mathbf{P}(w) \leftrightarrow \mathbf{A^P}$ is in Π.

5. If $AtBat_n(Seq)$ is defined and of the form $+\forall\mathbf{A}$ or $-\exists\mathbf{A}$, then $Chk_{3n-1}(Seq) \ldots Chk_{3n+1}(Seq)$ are $(Chk_n(Seq) \cup \{AtBat_n(Seq)\}) - MU$, where MU is the set of all signed sentences in $Ext_n(Seq)$ of the form $-\forall\mathbf{B}$ or $+\exists\mathbf{A}$.
6. If $AtBat_n(Seq)$ is defined and not of one of the above forms, then $Chk_{3n-1}(Seq) \ldots Chk_{3n+1}(Seq)$ are just $Chk_n(Seq) \cup \{AtBat_n(Seq)\}$.
7. If $AtBat_n(Seq)$ is undefined, then $Ext_{3n-1}(Seq) \ldots Ext_{3n+1}(Seq)$ are identical to $Ext_n(Seq)$, and $Chk_{3n-1}(Seq) \ldots Chk_{3n+1}(Seq)$ are identical to $Chk_n(Seq)$.

Theorem A..4 (Tree Derivability) If, for some sequent Seq with a finite parameter set, $Ext_{3n-1}(Seq)$, $Ext_{3n}(Seq)$, and $Ext_{3n+1}(Seq)$ are derivable, then so is $Ext_n(Seq)$.

Proof. The proof is by case analysis on the cases given in the table in definition A..3. Consider case (4.4), for example. If $Ext_{2n}(Seq)$ is derivable, then a proof for $Ext_n(Seq)$ can be constructed by simply adding a step onto the proof of $Ext_{2n}(Seq)$. The other cases are very similar, except for case (4.6), which we will consider here.

In case (4.6), if $Ext_{2n}(Seq)$ is derivable, then by Theorem 1 some finite subset of it is derivable. Call this finite subset S. If S is also a subset of $Ext_n(Seq)$, then by Theorem 1 that is derivable as well. Otherwise, S must be of the form $S_1 \cup S_2$, where S_1 is a subset of $Ext_n(Seq)$, and all the formulae in S_2 are of the form $-[\mathrm{t}/\mathrm{x}]\mathbf{A}$. A proof for $S_1 \cup \{-(\mathrm{x})\mathbf{A}\}$ can be constructed by adding one step for every sentence in S_2, replacing each by $-(\mathrm{x})\mathbf{A}$. But $S_1 \cup \{-(\mathrm{x})\mathbf{A}\}$ is clearly a subset of $Ext_n(Seq)$, which is therefore derivable. □

Corollary A..5 If $Ext_n(Seq)$ is not derivable, then either $Ext_{2n}(Seq)$ or $Ext_{2n+1}(Seq)$ is not derivable.

Proof. This is just the contrapositive to Thm. A..4. □

Corollary A..6 If Seq is not derivable, then there is an infinite sequence of integers $\pi_1, \pi_2, \ldots$ such that

1. π_1 is 1,
2. For all $i > 1$, π_i is either $3\pi_i - 1$, $3\pi_i$, or $2\pi_i + 1$, and
3. For no $i \geq 1$ is $Ext_{\pi_i}(Seq)$ derivable.

Proof. The proof is by induction, using Cor. A..5 in the induction step. □

Definition A..7 If $\pi_1, \pi_2, \ldots$ is the sequence mentioned in Cor. A..6 for a sequent Seq, then $NDExt_i(Seq)$ is $Ext_{\pi_i}(Seq)$, $NDChk_i(Seq)$ is $Chk_{\pi_i}(Seq)$, and $NDAtBat_i(Seq)$ is $AtBat_{\pi_i}(Seq)$.

Theorem A..8 (Batting Order) Consider any underivable sequent Seq with a finite parameter set. If the signed sentence σ_i is a member of $NDExt_j(Seq)$ but not of $NDChk_j(Seq)$, there is a $k \geq j$ such that $NDAtBat_k(Seq)$ is σ_i.

Proof. Let S be the set of signed sentences of the form $-\forall\mathbf{A}$ or $+\exists\mathbf{A}$ which are before σ_i in the enumeration, and let T be the set of signed sentences of all other forms before σ_i in the enumeration. Then let m_n, for $n \geq 1$, be $|S - NDChk_n(Seq)| + i * |T - NDChk_n(Seq)|$. For all k such that $NDAtBat_k(Seq)$ is before σ_i, we can show that $m_k \geq m_{k+1}$. This is because if such an $NDAtBat_k(Seq)$ is of the form $-(\mathrm{x})\mathbf{A}$, m_{k+1} will be $m_k - 1$, and if it is of any other form, m_{k+1} will be at least $m_k - i$ and at most $m_k - 1$.

Let σ_i appear in $NDExt_j(Seq)$, but not in $NDChk_j(Seq)$. It will thus appear in $NDExt_k(Seq)$ for all $k \geq j$. Now assume that there is no $k \geq j$ such that σ_i is $NDAtBat_k(Seq)$. Thus, for all $k \geq j$, there is some signed sentence before σ_i in the enumeration which appears in $NDExt_k(Seq)$ but not in $NDChk_k(Seq)$. We can conclude, since m_n is decreasing for all $k \geq j$, that there is some k at which m_k is zero. But the only way that could happen would be if all signed sentences before σ_i in the enumeration were in $NDChk_k(Seq)$, in which case $NDAtBat_k(Seq)$ would have to be σ_i, contradicting our assumption. Therefore there must be some $k \geq j$ such that $NDAtBat_k(Seq)$ is σ_i. □

Corollary A..9 If the signed sentence σ_i appears in any $NDExt_j(Seq)$, then there is a k such that $NDAtBat_k(Seq)$ is σ_i.

Proof. For $j > 1$, every member of $NDChk_j(Seq)$ appears in $NDExt_{j-1}(Seq)$. If σ_i appears in any $NDExt_j(Seq)$, there must be a smallest such j; therefore for that smallest j, σ_i is a member of $NDExt_j(Seq)$ but not of $NDChk_j(Seq)$, and the result of Theorem 3 holds. □

With the informal notions about sets that we have been using, we can assert that the "infinite union" of the sequents $NDExt_i(Seq)$ exists. That is, there is a set (let us call it $Ext^*(Seq)$) such that a signed sentence is in $Ext^*(Seq)$ if and only if it is in some $NDExt_i(Seq)$, for some finite $i \geq 1$.

Theorem A..10 (Extension non-derivability) For any underivable sequent Seq with a finite parameter set, $Ext^*(Seq)$ is not derivable.

Proof. Assume $Ext^*(Seq)$ is derivable. Then some finite subset of it is derivable. By our definition of $Ext^*(Seq)$, each member of this subset appears in some $NDExt_i(Seq)$.

But by our definitions, every $NDExt_i(Seq)$ is a subset of $NDExt_{i+1}(Seq)$, so there must be some $NDExt_j(Seq)$ in which every member of this subset appears. This would make $NDExt_j(Seq)$ derivable as well, contradicting our assumptions about the formation of $Ext^*(Seq)$. Therefore, $Ext^*(Seq)$ cannot be derivable. □

Corollary A..11 Whenever a signed sentence $\pm\mathbf{A}$ appears in $Ext^*(Seq)$, the signed sentence $\mp\mathbf{A}$ does not also appear.

Proof. If it did, $Ext^*(Seq)$ would be an axiom, and therefore derivable. □

Corollary A..12 No signed sentence of the form $+\mathbf{a} = \mathbf{a}$ appears in $Ext^*(Seq)$.

Proof. If it did, $Ext^*(Seq)$ would be an axiom, and therefore derivable. □

Corollary A..13 Whenever the signed sentences $-\mathbf{a} = \mathbf{b}$ and $+\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ appear in $Ext^*(Seq)$, the signed sentence $-\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ does not also appear.

Proof. If it did, the finite subset of $Ext^*(Seq)$, $\{+\mathbf{A}\{\mathbf{p} := \mathbf{a}\}, -\mathbf{A}\{\mathbf{p} := \mathbf{b}\}, -\mathbf{a} = \mathbf{b}\}$ could be derived from the two sequents $\{+\mathbf{A}\{\mathbf{p} := \mathbf{a}\}, -\mathbf{A}\{\mathbf{p} := \mathbf{a}\}\}$ and $\{+\mathbf{A}\{\mathbf{p} := \mathbf{b}\}, -\mathbf{A}\{\mathbf{p} := \mathbf{b}\}\}$, so $Ext^*(Seq)$ would also be derivable. □

Definition A..14 $\Phi(Seq)$, which we will prove to be the falsifying interpretation of Seq, is defined as follows. Let us assume, without loss of generality, that the parameter $\mathbf{p}_0$ is a member of the parameter set of Seq.

1. If $\pm\mathbf{A}$ is in $Ext^*(Seq)$ for $\mathbf{A}$ of the form $\mathbf{a} = \mathbf{b}$ or $\mathbf{P}(\mathbf{a})$, then $\mp\mathbf{A}$ is in $\Phi(Seq)$.
2. For all terms $\mathbf{a}$, $+\mathbf{a} = \mathbf{a}$ is in $\Phi(Seq)$.
3. For all terms $\mathbf{a}$ and $\mathbf{b}$, the following signed sentences are in $\Phi(Seq)$:
   (a) $+0 = 0\alpha$, $0\alpha = 0$
   (b) $-[\mathbf{a}, \mathbf{b}] = [\mathbf{a}, \mathbf{b}]\alpha$, $-[\mathbf{a}, \mathbf{b}]\alpha = [\mathbf{a}, \mathbf{b}]$ for all nonempty α
   (c) $+[\mathbf{a}, \mathbf{b}]h = \mathbf{a}$, $+\mathbf{a} = [\mathbf{a}, \mathbf{b}]h$
   (d) $+[\mathbf{a}, \mathbf{b}]t = \mathbf{b}$, $+\mathbf{b} = [\mathbf{a}, \mathbf{b}]t$
4. If the parameter set of $Ext^*(Seq)$ does not contain all the parameters, then, for all parameters $\mathbf{p}_i$ not in that set, the signed sentence $+\mathbf{p}_i = \mathbf{p}_0$ is in $\Phi(Seq)$.
5. If both $\pm\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ and $+\mathbf{a} = \mathbf{b}$ are in $\Phi(Seq)$, then $\pm\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ is in $\Phi(Seq)$.

6. If both $\pm\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ and $\mp\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ are in $\Phi(Seq)$, then $-\mathbf{a} = \mathbf{b}$ is in $\Phi(Seq)$.
7. For all other atomic sentences $\mathbf{A}$, $+\mathbf{A}$ is in $\Phi(Seq)$.

Theorem A..15 (Φ Interpretation) For any underivable sequent Seq with a finite parameter set, $\Phi(Seq)$ is an interpretation of Π.

Proof. That all atomic sentences are represented in one sign or another is obvious from clause 6 of the definition of $\Phi(Seq)$. No sentence in the set by virtue of clause 1 is represented with both $+$ and $-$ signs, because by Cor. A..11 no sentence is in $Ext^*(Seq)$ with both $+$ and $-$ signs.

The signed sentence $+\mathbf{a} = \mathbf{b}$ can enter $\Phi(Seq)$ only if $\mathbf{a}$ and $\mathbf{b}$ are identical, or if $-\mathbf{a} = \mathbf{b}$ is in $Ext^*(Seq)$ (in which case by Cor. A..13 not both of $\pm\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ and $\mp\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ can be in $\Phi(Seq)$), or if $\mathbf{a}$ is not in the parameter set of $Ext^*(Seq)$, or if there is no $\mathbf{A}$ such that both $\pm\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ and $\mp\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ are in $\Phi(Seq)$. In all cases where both $\pm\mathbf{A}\{\mathbf{p} := \mathbf{a}\}$ and $\mp\mathbf{A}\{\mathbf{p} := \mathbf{b}\}$ are in $Ext^*(Seq)$, $-\mathbf{a} = \mathbf{b}$ is in $\Phi(Seq)$. Clearly, the conditions for interpretations of Π on identity sentences are met, and no sentence can appear with both a $+$ and $-$ sign in $\Phi(Seq)$, so $\Phi(Seq)$ is an interpretation of Π. □

Theorem A..16 (Extension Completeness) For any underivable sequent Seq whose parameter set is finite, $Ext^*(Seq)$ is not true in $\Phi(Seq)$.

Proof. This is equivalent to saying that no signed sentence in $Ext^*(Seq)$ appears in the closure of $\Phi(Seq)$. The proof is by induction on the complexity of individual signed sentences in $Ext^*(Seq)$.

By the definition of $\Phi(Seq)$, if $\pm\mathbf{A}$ is in $Ext^*(Seq)$ for $\mathbf{A}$ of the form $\mathbf{a} = \mathbf{b}$ or $\mathbf{P}(\mathbf{a})$, then $\mp\mathbf{A}$ is in $\Phi(Seq)$; so clearly, all signed sentences in $Ext^*(Seq)$ of complexity 0 are not in the closure of $\Phi(Seq)$.

Assume that all sentences of complexity k in $Ext^*(Seq)$ are not in the closure. If a signed sentence of the form $\pm\neg\mathbf{A}$ and of complexity $k+1$ is in $Ext^*(Seq)$, then it must be in some $NDExt_i(Seq)$; therefore by Cor. A..9 there must be some j such that the formula is $NDAtBat_j(Seq)$; therefore $\mp\mathbf{A}$ is in $NDExt_{j+1}(Seq)$ and thus in $Ext^*(Seq)$; therefore by our induction assumption $\pm\mathbf{A}$ must be in the closure of $\Phi(Seq)$; therefore by the semantic successor rules $\mp\neg\mathbf{A}$ must also be in the closure of $\Phi(Seq)$; therefore $\pm\neg\mathbf{A}$ cannot be in the closure of $\Phi(Seq)$.

The other cases for signed sentences σ are similar, except for the case where σ is of the form $+\exists\mathbf{A}$ (or $-\forall\mathbf{A}$), which will be covered here. Every term $\mathbf{a}$ which appears in $Ext^*(Seq)$ is formed from a finite number of parameters. Therefore, there is some earliest $NDExt_k(Seq)$ whose parameter set contains all the parameters from which $\mathbf{a}$ is formed. Further, either $k = 1$ or $NDAtBat_{k-1}(Seq)$ is of the form $-\exists\mathbf{A}$ (or $+\forall\mathbf{A}$).

So if σ is in $Ext^*(Seq)$, it must be $NDAtBat_m(Seq)$ for some $m \geq k$, because even if σ is an element of $NDExt_k(Seq)$, it will not be an element of $NDChk_k(Seq)$, and so by Theorem A..8, must come "up to bat" again.

We therefore must have $\pm\mathbf{A}([\mathbf{a}, w])$ in $Ext^*(Seq)$ for all $\mathbf{a}$ formed from parameters in the parameter set of $Ext^*(Seq)$. By our induction assumption, we must have $\mp\mathbf{A}([\mathbf{a}, w])$ in the closure of $\Phi(Seq)$ for all such $\mathbf{a}$. Because all parameters which are not members of the parameter set of $Ext^*(Seq)$ are made equivalent to one which is, $\mp\mathbf{A}([\mathbf{a}, w])$ must also be in the closure of $\Phi(Seq)$ for all terms formed from those parameters as well; in other words, for all terms $\mathbf{a}$. By the semantic successor rules, we must therefore have $-\exists\mathbf{A}$ $(+\forall\mathbf{A})$ in the closure of $\Phi(Seq)$, and therefore σ is not in the closure.

By induction, we can therefore say that $Ext^*(Seq)$ is not true in $\Phi(Seq)$. □

Corollary A..17 No underivable sequent Seq whose parameter set is finite is true in $\Phi(Seq)$.

Proof. Since $Ext^*(Seq)$ has no intersection with the closure of $\Phi(Seq)$, no subset of it can have an intersection with the closure either. Seq, however, is a subset of $Ext^*(Seq)$, and therefore cannot be valid with respect to $\Phi(Seq)$. □

Corollary A..18 (Completeness) Every valid finite sequent is derivable.

Proof. Consider any underivable finite sequent Seq. The parameter set of Seq must be finite; therefore, there is a base, $\Phi(Seq)$, in which it is not true; therefore it is not valid. Conversely, therefore, every valid sequent is derivable. □

Cor. A..18 is the main result of this appendix. It tells us that the proof theory C we have developed for environment theory is complete; that is, that every finite sequent which is valid, and therefore desirable to be derived, is in fact derivable.

Appendix B

Environment Solution Algorithm

Definition B..19 A pointer $\mathbf{x}$ in an environment $\mathbf{a}$ is *dereferenced* if $\mathbf{a/x} \equiv \mathbf{x}$. An environment $\mathbf{a}$ is dereferenced if all pointers it contains are dereferenced; that is, if all pointers either are self-pointers or point to self-pointers.

Lemma B..20 There is a terminating algorithm which converts every environment $\mathbf{a}$ to an equivalent dereferenced environment $\mathbf{a}'$.

Proof. There are a finite number of undereferenced pointers $\mathbf{x}$ in $\mathbf{a}$ (pointers such that $\mathbf{a/x} \not\equiv \mathbf{x}$). Starting at the rightmost such pointer, at part $\mathbf{y}$ of $\mathbf{a}$, set $\mathbf{a/y}$ to $\mathbf{a/(a/y)}$. The altered environment will be equivalent to the original, and all pointers in $\mathbf{a/y}$ will now point to self-pointers; therefore the number of undereferenced pointers will have been lowered. Repeat with the rightmost undereferenced pointer until the number of such pointers reaches 0. □

Definition B..21 An *equation system* is a formula of the form $\mathbf{D}_1 \vee \mathbf{D}_2 \vee \ldots \vee \mathbf{D}_n$, where each disjunct $\mathbf{D}_i$ is of the form $w = \mathbf{a}_i \;\&\; \mathbf{C}_{i,1} \;\&\; \mathbf{C}_{i,2} \;\&\; \ldots \;\&\; \mathbf{C}_{i,m_i}$ and each conjunct $\mathbf{C}_{i,j}$ is of the form $\mathbf{b}_{i,j} = \mathbf{c}_{i,j}$ (an *equation* conjunct) or $\mathbf{b}_{i,j} \neq \mathbf{c}_{i,j}$ (an *inequality* conjunct).

Lemma B..22 If, in an equation system, for all equation conjuncts $\mathbf{C}_{i,j}$ of the disjunct $\mathbf{D}_i$,

1. $\mathbf{b}_{i,j}$ is a pointer $\mathbf{x}_{i,j}$,
2. $\mathbf{c}_{i,j}$ is a pair $[\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$, and
3. each of the pointers $\mathbf{x}_{i,j}$ appears in some $\mathbf{c}_{i,k}$, where $\mathbf{C}_{i,k}$ is an equation conjunct,

then the disjunct $\mathbf{D}_i$ is equivalent to **fail**; that is, the sequent $\{-\mathbf{D}_i\}$ is valid.

Proof. Assume the stated conditions. Let us say that a pointer $\mathbf{b}_{i,j}$ *refers* to another pointer $\mathbf{b}_{i,k}$ if $\mathbf{b}_{i,k}$ is a part of $\mathbf{c}_{i,j}$. Because each $\mathbf{b}_{i,k}$ is referred to by some $\mathbf{b}_{i,j}$, we can easily construct a list of pointers $\mathbf{y}_1, \mathbf{y}_2, \ldots \mathbf{y}_{m_i+1}$ such that $\mathbf{y}_{i+1}$ refers to y_i for all $1 \leq i \leq m_i$. Clearly there must be some $\mathbf{b}_{i,j}$ which appears twice in the list. This implies that $\mathbf{b}_{i,j}$ is a proper part of itself; that is, that $+\mathbf{b}_{i,j}\alpha = \mathbf{b}_{i,j}$, for some nonempty α, is in any base whose closure contains $+\mathbf{D}_i$. Since this is not allowed by the semantics of pair identity, $-\mathbf{D}_i$ must be in the closure of all bases, and $\mathbf{D}_i$ must be equivalent to **fail**. □

Definition B..23 The *depth* of 0 and of all pointers is 0; the depth of a pair is the greater of the depths of its head and tail. The *depth measure* of an equation $\mathbf{a} = \mathbf{b}$ or inequality $\mathbf{a} \neq \mathbf{b}$ is ω^d, where ω is the first transfinite limit ordinal and d is the greater of the depths of **a** and **b**.

Environment Solution Algorithm. The algorithm considers the disjuncts as a linear list. The addition of a disjunct to the end of the list is done in the natural way, by increasing the value of m_i. The conjuncts within each top-level disjunct are also treated as a list, and can be added to or eliminated individually.

Throughout, the order of terms in an equality or inequality is disregarded; any reference to (say) an equation of the form $\mathbf{a} = \mathbf{b}$ can be considered as referring also to equations of the form $\mathbf{b} = \mathbf{a}$.

1. Repeat, for all disjuncts $\mathbf{D}_i$ not of the form **fail**, until either one disjunct $\mathbf{D}_i$ is such that $\mathbf{C}_{i,j} \equiv \mathbf{x}_{i,j} \neq \mathbf{y}_{i,j}$ for all $1 \leq j \leq m_i$, or all disjuncts have been replaced by **fail**:

   (a) If $\mathbf{a}_i$ is not dereferenced, dereference $\mathbf{a}_i$. (From lemma B..20 we know that this procedure terminates.)

   (b) Otherwise, if there is a pointer $\mathbf{x}\alpha$ appearing in any $\mathbf{C}_{i,j}$ such that $\mathbf{a}_i/\mathbf{x} \equiv \mathbf{x}$ and $\alpha \not\equiv \epsilon$, then for each such pointer:

      i. Replace every occurrence of $\mathbf{x}\beta$ in $\mathbf{D}_i$, for every β, with 0.

      ii. Create a new disjunct which is identical to $\mathbf{D}_i$, except that every occurrence of $\mathbf{x}$ in $\mathbf{a}_i$ is replaced by an occurrence of $[\mathbf{x}h, \mathbf{x}t]$.

      (This loop terminates because there are a finite number of dangling pointers in the original $\mathbf{D}_i$, and no new dangling pointers are introduced into $\mathbf{D}_i$ by the procedure.)

   (c) Otherwise, if there is any pointer $\mathbf{x}$ appearing in any $\mathbf{C}_{i,j}$ such that $\mathbf{a}_i/\mathbf{x} \not\equiv \mathbf{x}$, then for each such pointer, replace it by $\mathbf{a}_i/\mathbf{x}$. (Since there are a finite number of pointers in the list of conjuncts, this procedure clearly terminates.

After this step, due to the definition of a dereferenced environment, every pointer $\mathbf{x}$ in every $\mathbf{b}_{i,j}$ or $\mathbf{c}_{i,j}$ points to a self-pointer; that is, $\mathbf{a}_i/\mathbf{x} \equiv \mathbf{x}$.)

(d) Otherwise, if any conjunct $\mathbf{b}_{i,j} = \mathbf{c}_{i,j}$ of depth measure $\omega^{d_{i,j}}$ is such that $\mathbf{b}_{i,j} \equiv [\mathbf{b}^h_{i,j}, \mathbf{b}^t_{i,j}]$ and $\mathbf{c}_{i,j} \equiv [\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$, then for each such conjunct, replace the conjunct by $\mathbf{b}^h_{i,j} = \mathbf{c}^h_{i,j}$ and create a new conjunct of the form $\mathbf{b}^t_{i,j} = \mathbf{c}^t_{i,j}$. (The depth measure of each new conjunct must be $\leq \omega^{d_{i,j}-1}$, so the sum of the depth measures of the equality conjuncts has been decreased by at least $\omega^{d_{i,j}} - 2\omega^{d_{i,j}-1}$.)

(e) Otherwise, if there are any conjuncts of the form $0 = 0$, eliminate them. (Each elimination decreases the sum of the equality conjunct depth measures by 1.)

(f) Otherwise, if any conjunct is of the form $0 = [\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$, replace the entire disjunct by **fail**.

(g) Otherwise, if any conjunct is of the form $\mathbf{x} = \mathbf{c}_{i,j}$ or its inverse, where $\mathbf{c}_{i,j}$ is either 0 or a pointer to the right of $\mathbf{x}$, then for each such conjunct, replace occurrences of $\mathbf{x}$ in $\mathbf{D}_i$ by occurrences of $\mathbf{c}_{i,j}$, and eliminate the original conjunct. (Due to the fact that all pointers pointed to self-pointers, the environment remains fully dereferenced, and all pointers in all conjuncts still point to self-pointers. The depths of the other conjuncts have not been changed, but one conjunct has been eliminated, so the sum of the equality conjunct depth measures has been decreased. At this point in the algorithm, we can assume without loss of generality that every equality conjunct is of the form $\mathbf{x} = [\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$.)

(h) Otherwise, if every pointer $\mathbf{b}_{i,j}$ in an equality conjunct appears in some $\mathbf{c}_{i,k}$ in an equality conjunct, then by Lemma B..22, replace the entire disjunct by **fail**.

(i) Otherwise, if there are still equality conjuncts in the disjunct, let $\mathbf{C}_{i,j}$ be an equality conjunct such that $\mathbf{b}_{i,j} \equiv \mathbf{x}$ does not appear in any $\mathbf{c}_{i,k}$ in an equality conjunct.

   i. Replace every occurrence of $\mathbf{x}$ in $\mathbf{a}_i$ with an occurrence of $\mathbf{c}_{i,j}$.

   ii. A finite number of pointers in $\mathbf{a}_i$ will now point to a self-pointer to the left. For each such pointer $\mathbf{y}$ at part $\mathbf{a}_i/\mathbf{z}$, replace all occurrences of $\mathbf{y}$ in $\mathbf{a}_i$ by occurrences of $\mathbf{z}$. (This will effectively shift all pointers so that they point to the right.)

   iii. Eliminate the original conjunct, $\mathbf{x} = \mathbf{c}_{i,j}$.

   iv. Replace every occurrence of $\mathbf{x}$ in the disjunct with an occurrence of $\mathbf{a}_i/\mathbf{x}$. (These occurrences can only be on the left hand side of conjuncts.)

Say that the depth of the original $\mathbf{c}_{i,j}$ was d. The depth measures of some conjuncts have been raised to ω^d. However, at the next repetition of step 4a, because all right hand sides are pairs, they will all be split into two conjuncts of depth $d-1$. Therefore, a factor of ω^d will have been subtracted from the sum of the equality conjunct depth measures, but only a finite number of factors of ω^{d-1} will have been introduced. There will therefore have been a net decrease in the sum of the equality conjunct depth measures.

(j) Otherwise, if there are conjuncts of the form $[\mathbf{b}^h_{i,j}, \mathbf{b}^t_{i,j}] \neq [\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$, then for each such conjunct,

i. Replace the conjunct by $\mathbf{b}^h_{i,j} \neq \mathbf{c}^h_{i,j}$.

ii. Create a new disjunct which is identical to $\mathbf{D}_i$, except that $\mathbf{C}_{i,j}$ is replaced by $\mathbf{b}^t_{i,j} \neq \mathbf{c}^t_{i,j}$.

(k) Otherwise, if there are conjuncts of the form $0 \neq 0$ or $\mathbf{x} \neq \mathbf{x}$, replace the entire disjunct by **fail**.

(l) Otherwise, if there are conjuncts of the form $0 \neq [\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$, eliminate them. (At this point we can assume without loss of generality that all conjuncts are of the form $\mathbf{x}_{i,j} \neq \mathbf{c}_{i,j}$, where $\mathbf{x}_{i,j}$ is a pointer to a self-pointer in $\mathbf{a}_i$.)

(m) Otherwise, if there are conjuncts of the form $\mathbf{x}_{i,j} \neq [\mathbf{c}^h_{i,j}, \mathbf{c}^t_{i,j}]$, then for each such conjunct:

i. Create a new disjunct which is identical to $\mathbf{D}_i$, except that every occurrence of $\mathbf{x}_{i,j}$ is replaced by an occurrence of $[\mathbf{x}_{i,j}h, \mathbf{x}_{i,j}t]$.

ii. Replace every occurrence of $\mathbf{x}_{i,j}$ in $\mathbf{D}_i$ by an occurrence of 0.

iii. Eliminate the original conjunct.

(n) Otherwise, there must be conjuncts of the form $\mathbf{x}_{i,j} \neq 0$. For each such conjunct, replace all occurrences of $\mathbf{x}_{i,j}$ in $\mathbf{D}_i$ by occurrences of $[\mathbf{x}_{i,j}h, \mathbf{x}_{i,j}t]$, and eliminate the original conjunct.

2. If all disjuncts are of the form **fail**, terminate returning **fail**.

3. Otherwise, if one disjunct $\mathbf{D}_i$ is such that $\mathbf{a}_i$ is a proper environment and each $\mathbf{C}_{i,j}$ is of the form $\mathbf{x}_{i,j} \neq \mathbf{y}_{i,j}$, terminate returning $(w = \mathbf{a}_i \;\&\; \mathbf{C}_{i,1} \;\&\; \ldots \;\&\; \mathbf{C}_{i,m_i}) \vee \mathbf{D}_1 \vee \ldots \vee \mathbf{D}_{i-1} \vee \mathbf{D}_{i+1} \vee \ldots \vee \mathbf{D}_n$.

End of algorithm.

Theorem B..24 Given an equation system, the environment solution algorithm above will terminate finding an equivalent equation system of the form $(w = \mathbf{a} \;\&\; \mathbf{x}_1 \neq \mathbf{y}_1 \;\&\; \ldots \;\&\; \mathbf{x}_m \neq \mathbf{y}_m) \vee \mathbf{B}$, if and only if there is such an equation system. If such an equation system does not exist, the given equation system will be equivalent to **fail**.

Proof. Note the following things about the algorithm.

1. All of the clauses contain a condition and an action; the action is performed if and only if the condition holds and none of the conditions in the previous clauses hold.

2. All of the actions in all the clauses terminate individually.

3. If the action in clause 1(a) has been performed for a particular disjunct $\mathbf{D}_i$, none of the subsequent clauses will allow its condition to become true again; similarly for clauses 1(b) and 1(c).

4. As long as there are equality conjuncts in a particular disjunct, one of the clauses 1(d) - 1(i) will be performed on that disjunct at every repetition of the loop.

5. The action in each of the clauses 1(d) - 1(i), when applied to any disjunct $\mathbf{D}_i$, causes the sum of the depth measures of the equality conjuncts to decrease; therefore, since the sum of the depth measures is always a countable ordinal, by transfinite induction the conditions on all the clauses must not hold after some finite number of repetitions.

6. None of the clauses 1(j) - 1(n) causes any of the conditions in the previous clauses to become true.

7. The action in each of the clauses 1(j) - 1(n), when applied to any disjunct $\mathbf{D}_i$, causes the sum of the depth measures of the remaining conjuncts to decrease. Again, by transfinite induction these clauses can be repeated only a finite number of times for each $\mathbf{D}_i$.

From the above points it should be clear that for each disjunct, each clause in step (1) can be applied to it only a finite number of times before it becomes converted to one of the success forms. However, by that time it may have generated more disjuncts at the end of the list.

Each original disjunct can generate only a finite number of additional disjuncts by virtue of clause 1(b), because the new disjunct created there will have either fewer dangling pointers, or one pointer will be dangling at a lower level. Similarly, only a finite number of additional disjuncts can be created by virtue of clause 1(j), because the new disjunct will have a lower sum of depth measures than the original. The problematic clause is clause 1(m).

Consider the disjunct $w = \mathbf{a}\ \&\ (\mathbf{x} \neq \mathbf{y}\ \&\ \mathbf{x} \neq [\mathbf{y}, \mathbf{z}]\ \&\ w = w)$. When clause 1(m) is applied to this disjunct, it will create a new disjunct of the form $w = \mathbf{a}\ \&\ ([\mathbf{x}h, \mathbf{x}t] \neq \mathbf{y}\ \&\ [\mathbf{x}h, \mathbf{x}t] \neq [\mathbf{y}, \mathbf{z}]\ \&\ w = w)$. Clause 1(j) will then be applied, converting the new

disjunct to the form $w = \mathbf{a}$ & $([\mathbf{x}h, \mathbf{x}t] \neq \mathbf{y}$ & $\mathbf{x}h \neq \mathbf{y}$ & $w = w)$ and creating another disjunct. But the new disjunct will be of the same form as the original; therefore an infinite production of new disjuncts will be generated, unless the algorithm terminates due to some other disjunct succeeding.

However, note that at each iteration of this creation of disjuncts, the depth of the environment $\mathbf{a}_i$ will be increased. If the entire disjunction has a solution, however, it must have a solution with an environment of finite depth. Therefore, the infinite iteration of applications of clause 1(m) can only take place if the entire equation system has no solution.

The algorithm will convert the equation system to an equivalent one of the solution form if such an equivalent form exists; if one does not exist, the equation system will be equivalent to **fail**, and in that case the algorithm may or may not terminate. □

Thus, the environment solution algorithm is similar to the rest of the computation algorithm in the following sense. If a sequent containing only $+\mathbf{E}$ & $\mathbf{a} = \mathbf{b}$ or $+\mathbf{E}$ & $\neg\mathbf{a} = \mathbf{b}$ is valid, where $\mathbf{E}$ is an environment characterization, then the algorithm will terminate finding an environment characterization $\mathbf{F}$ such that $\{\mathbf{F} \rightarrow \mathbf{E}$ & $\mathbf{a} = \mathbf{b}\}$ is valid. However, if the sequent is not valid, there is no guarantee that the algorithm will invariably so conclude.

Although this is sufficient for the purposes of this theory, it is not entirely satisfying, since we may be able to describe deterministically the set of all equation systems which fail. In doing so, we will be forced to change the form or use of an environment characterization. We should preserve the property of the environment characterization that it be easy to pass back through a quantifier, and this may not be easy to do.