LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

The Logical Structure of Sequential Prolog

by

James H. Andrews

LFCS Report Series ECS-LFCS-90-110

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

May 1990

The Logical Structure of Sequential Prolog

James H. Andrews
University of Edinburgh
Department of Computer Science
jha@lfcs.ed.ac.uk

May 7, 1990

Abstract: The standard logical semantics of sequential (left-to-right, depth-first) Prolog does not deal with questions of completeness, and therefore does not adequately describe termination. Denotational characterisations deal with termination properly, but do so in a very functional, non-logical manner. In contrast, this paper presents sound, complete, and logical characterisations of pure sequential Prolog with which termination and correctness properties of programs can be proved.

These two characterisations are in the form of proof systems. The first one is sound and fully complete, in the sense that a query formula **A** succeeds iff the judgment $S(\exists[\mathbf{A}])$ is a theorem, and fails iff the judgment $F(\exists[\mathbf{A}])$ is a theorem. More generally, $S(\mathbf{A})$ (resp. $F(\mathbf{A})$) is a theorem iff every ground instance of **A** succeeds (resp. fails).

This first characterisation is infinitary, because no finitary system can have these properties. The second system is finitary and sound, and has a weaker completeness property, but can form the basis for a useable system for proving properties of programs.

These characterisations provide further insight into the logical structure that sequential Prolog has as part of the logic programming paradigm.

Keywords: Sequential Prolog, termination, operational semantics, declarative semantics, proof theory, logics of programs

1 Introduction

This paper presents the results of research into finding a sound, complete and logical characterisation for sequential Prolog.

I use the phrase "sequential Prolog" to mean not the syntax of any particular language, but rather the common left-to-right, depth-first strategy for implementing logic programming languages in general. As readers will be aware, this strategy is incomplete: there are queries which have solutions according to the semantics of logic programming, but for which sequential Prolog cannot find solutions. However, the sequential strategy is still very popular, and is the one implemented in the vast majority of practical logic programming systems. This suggests that the subset of valid queries on which Prolog terminates is still practically useful, and that we might benefit from obtaining an abstract, declarative characterisation of it.

The abstract characterisations that have been given for sequential Prolog, however – even pure Prolog without negation and cut $\vdash$ have a tendency to be not as logical as we might want. Operational semantics [FGKP85] necessarily concentrate on such issues as representations of stacks and substitutions. Denotational semantics [Bau88, NF89] are supported by a more abstract mathematical theory, but give functions as the denotations of most program entities; this is disappointing because the whole point of the logic programming paradigm is that we do *not* have to view a program as a function from inputs to outputs. A reliance on functional semantics to describe the most common Prolog interpreter strategy tends to weaken claims that logic programming is a separate and independent paradigm.

In this paper, I describe proof-theoretic characterisations of pure (negationless, cut-free) sequential Prolog. These are proof systems, very similar to the standard sequent calculus and natural deduction systems of mathematical logic, which have soundness and completeness properties with respect to the operational semantics of sequential Prolog. Proof-theoretic characterisations have several advantages: they are highly compositional; they give very logical interpretations to the connectives used in logic programming; they abstract the essential meanings of programs without explicitly representing operational details; and, on a purely practical level, their soundness and completeness properties allow us to use them to formally prove termination and correctness properties of logic programs. Such characterisations have been increasingly studied in the last few years, but mostly in the context of nondeterministic or parallel logic programming [HS84, HSH88, And89a].

The sound, complete and logical characterisations given here allow us to affirm that the essential "logicalness" of the logic programming paradigm extends to the logical structure of its most common implementation, sequential Prolog. They deal only with pure negationless Prolog, but because they handle the crucial question of sequentiality in a logical way, they should form the basis of logical characterisations of richer sequential languages.

The organisation of this paper is as follows. After some notational preliminaries, I discuss the various semantic approaches in more detail in section 3. In section 4, I give a precise formal description of the operational semantics of sequential Prolog, which will act as a reference point for the soundness and completeness results. In section 5, I give an unsatisfactory attempt at a characterisation, which is similar to one outlined by Girard [Gir87]; I do this to motivate the notions of unfoldings of formulae that I use to construct

the final characterisations.

It is impossible to find a sound, complete, and finitary proof system of the kind that we want, because of a variant of Gödel's incompleteness theorem; however, the literature gives standard tricks which mitigate the effects of incompleteness. In section 6, I describe a proof system which is infinitary (contains a rule with an infinite number of premisses), but is sound and complete; and in section 7, I describe one which is finitary and sound, but which has only a limited (but still useful) form of completeness. For the interest of the readers, section 8 contains an outline of the incompleteness proof.

2 Definitional Preliminaries

For the purposes of this paper, the only logical connectives which will concern us are conjunction, disjunction, and the existential quantifier. I therefore define a "program formula" as one containing only these connectives. In the rest of this paper, "formula" refers to a program formula, unless otherwise specified.

Definition 2.1 A *program formula* in a first-order language $\mathcal{L}$ is a well-formed formula of $\mathcal{L}$, built up from predicate applications using only "and" (written "&"), "or" ("$\vee$"), and "there exists" ("$\exists$").

Instead of restricting *queries* to atoms or their negations, we will allow any program formula to be a query. Thus "query" and "program formula" are synonymous; but we will use the former when we want to refer to a formula, possibly having free variables, for which we ask a logic programming system to find a satisfying substitution.

In the traditional approach to logic programming, used to describe Prolog and similar languages, programs are defined as sets of Horn clauses. Because I wish to compare logic programming systems directly with proof systems, I adopt a form of predicate definition which looks more like the completion of a predicate [Cla78].

Definition 2.2 A *predicate definition* in $\mathcal{L}$ (defining the predicate $\mathbf{P}$) is an expression of the form

$\mathbf{P}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \leftrightarrow \mathbf{A}$

where the left- and right-hand sides of the $\leftrightarrow$ are program formulae in $\mathcal{L}$, and the right-hand side contains no free variables not in $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$.

Definition 2.3 A *program* in $\mathcal{L}$ is a finite set of predicate definitions in which all names of predicates being defined are distinct.

This form is no loss or gain of power over the clausal form, but it makes connectives explicit and allows us to examine their effect and processing directly. To get a program of this form from a Horn clause program, we need only take the completion of the program [Cla78]. Example: in a language with an equality predicate "=" and binary function symbol $[\_|\_]$ of list formation, the standard "member" predicate might be defined as follows:

$$mem(x, l) \leftrightarrow \exists h \exists t (l = [h|t] \& (x = h \vee mem(x, t)))$$

We will generally want to interpret a query **A** given to a logic programming system as a request to prove the existential closure of **A**. Since the logic we use will turn out to be constructive, this is the same as a request to find a satisfying substitution for **A**; the existential-closure view will just facilitate the comparison with the proof theory.

Definition 2.4 The *existential closure* of a formula **A**, in symbols $\exists[\mathbf{A}]$, is the formula $\exists \mathbf{x}_1 \ldots \exists \mathbf{x}_n \mathbf{A}$, where $\mathbf{x}_1 \ldots \mathbf{x}_n$ are all the free variables of **A**.

In the sequel we will assume the existence of a first-order language $\mathcal{L}$ with equality $=$ as the language of all programs. We will further assume that $\mathcal{L}$ has at least one nullary operation symbol. (This is not really a restriction for most practical applications.) We will write the nullary operator as 0, and define the formula *true* as an abbreviation for $0 = 0$.

Other notation is as follows. **A**, **B**, **C**, **D** are metavariables standing for arbitrary formulae; **P**, **Q**, and **R** stand for predicate names; **r**, **s**, and **t** stand for terms; and **x**, **y**, and **z** stand for variables.

I use the notation $\mathbf{A}[\mathbf{x} := \mathbf{t}]$ to stand for "**A**, with all free occurrences of **x** substituted by **t**" (possibly with renaming to avoid capture of free variables in **t**). I use the notation $\mathbf{A}(\mathbf{s})$ and then later $\mathbf{A}(\mathbf{t})$ to mean $\mathbf{A}[\mathbf{x} := \mathbf{s}]$ and then later $\mathbf{A}[\mathbf{x} := \mathbf{t}]$, for some given formula **A** with some given variable **x** free. ($\mathbf{A}(\mathbf{x})$ should not be confused with $\mathbf{P}(\mathbf{x})$, which is an application of predicate **P** to variable **x**.) Similarly I use $\mathbf{r}[\mathbf{s} := \mathbf{t}]$, $\mathbf{r}(\mathbf{s})$, $\mathbf{r}(\mathbf{t})$.

3 Approaches to Semantics

The traditional framework in which logic programming semantics has been studied is one with a declarative semantics (least model and/or fixpoint semantics) and an operational semantics (SLD-resolution with given search strategies). This framework describes the heart of the paradigm. But the assumptions made about search strategies in most research on foundations of logic programming (for instance, a fair search rule) are not satisfied by sequential Prolog; so sequential Prolog may diverge in cases where the traditional framework says it should succeed. The framework therefore does not deal with an issue crucial to practical Prolog programmers, namely the issue of program termination.

Various approaches have been taken to describing termination, including analyses of the operational semantics, and denotational descriptions that implicitly take termination into account. In this section I will discuss these approaches and motivate the approach used in this paper.

3.1 Declarative Semantics and Termination

First, I should describe exactly why the traditional declarative semantics (such as that described by Lloyd [Llo84]) does not handle issues of termination of sequential Prolog programs.

SLD-resolution is a nondeterministic proof procedure for Horn clause logic. If a goal is a logical consequence of a Horn clause program, then the negation of the goal has an SLD-refutation with respect to that program; but we need some way, outside the description of SLD-resolution, of finding the SLD-refutation within a tree of candidate refutations.

The sequential Prolog strategy is to just do a depth-first search in this tree, so a Prolog interpreter might not find a solution to the query if it strikes an infinite branch. For instance, consider a program which contains the predicate definition $Loop() \leftrightarrow Loop()$. With this program, the query $(Loop() \vee true)$, while clearly a consequence of any program, will cause a Prolog interpreter to go into infinite recursion. The strategy is at least sound, so we can tell that if Prolog tells us a solution exists (or that no solution exists) it is telling us the truth; but there are still cases in which Prolog will not find an answer, even if it exists.

This problem of termination is of great importance to Prolog programmers. It causes difficulties with reading programs, because a nice logical reading of a given program does not guarantee that it works properly. It also causes difficulties when programmers try to prove properties of their programs – something one would imagine would be easy in a logic-based language. Many approaches to proving properties of Prolog programs given in the literature [BC89, Der89] implicitly ignore sequential Prolog; these approaches work only for specific forms of program, or for complete, or-parallel, Prologs.

Because traditional semantics cannot describe the termination of sequential Prolog programs, it cannot assess the relative strengths of incomplete interpreters. Consider, for example, the trivial proof procedure which immediately goes into an infinite loop on every query. The traditional semantics cannot distinguish between this strategy and sequential Prolog; both are sound, in that they give no answer that is wrong, and both are incomplete.

There are various *ad hoc* semantics that would distinguish between the trivial procedure (or ones stronger than it) and sequential Prolog. However, the ideal would clearly be to find an abstract characterisation which is specific to sequential Prolog, which serves as a benchmark to judge the strengths of other strategies relative to it.

3.2 Operational Approaches

Descriptions of operational semantics are important to the description of logical languages, because it is there that the procedural reading is made. One way of proving properties of sequential programs is to study a formal operational semantics with mathematical tools.

In a paper of Francez et al. [FGKP85], for instance, a characterisation is given of terminating Prolog computations. One can prove that a computation terminates by giving conditions on the form of the tree of candidate solutions: if there are no infinite branches to the left of the first solution, then the program terminates. Francez et al. also give a proof system in which proofs of properties of programs can be made.

This is an adequate method of characterising termination. However, the operational semantics of a logic programming language is clearly secondary to the declarative semantics, which is where the whole purpose of the language comes from. A characterisation of termination in terms of the underlying logic of the language would be preferable to this purely operational description. The proof system approach, while having a logical structure, reifies such concepts as answer substitutions and unification, which are more properly of the operational semantics than the abstract logical structure of Prolog programs. We therefore achieve very little abstract mathematical insight from this technique.

3.3 Denotational Approaches

There have been several denotational analyses of sequential Prolog (some examples are [JM84, AB87, DM88, Bau88, dBdV89, NF89]), and these seem to bring out deeper and more abstract properties of the language.

Of these, one appealing example is Baudinet's [Bau88]. In this semantics, queries are given denotations in the set of functions which map substitutions to (essentially) sequences of substitutions. "Append" and "join" operators are defined which allow the derivation of denotations of composite goals, and of sequences of clauses, from the denotations of individual atoms. This semantics is therefore compositional and has clear connections to standard techniques in denotational semantics.

However, it is disappointing that this characterisation is so functional and so far removed from logic. Unification is again reified, and the result of a computation is viewed as the result of a composition of functions applied to the empty substitution. One might argue that this kind of functional denotational semantics is inappropriate for logic programming for philosophical reasons: denotational semantics views every program as a function from inputs to outputs, while the whole point of the logic programming paradigm is that it is not necessary to view a program in this way. The reason that programmers choose a logic programming language is often that their problem has some inherent logical structure which is reflected in the language, and not well described by functions.

One advantage of denotational semantics approaches like Baudinet's is that they allow a description of cut and negation as failure which is well-integrated with the rest of the semantics. The well-known "non-logicalness" of these features remains, however.

3.4 Proof-Theoretic Approaches

The goal of proof-theoretic characterisations of logic programming [HS84, HSH88, And89a] is to combine the logicalness of model-theoretic descriptions with the precision and formality of operational descriptions. A proof-theoretic characterisation associates a program with a proof system with respect to which the operational semantics is sound and complete. The soundness and completeness typically takes the following form: a query is successful iff an associated formula is a theorem of the proof system.

Proof-theoretic characterisations have several advantages. As in least model semantics, programs are described in terms of the logic underlying our intuitions about programs. But unlike model-theoretic approaches, the characterisation is purely formal and syntactic, so the proofs of soundness and completeness can be purely syntactic, with little reference to abstract notions. If the characterisations are in the form of natural deduction proof systems, they can be as clear and understandable as such proof systems are for describing conventional logics (see Gentzen [Gen69] or Sundholm [Sun83]). Finally, characterising proof systems can be used directly to build systems for verifying correctness and termination properties of logic programs [And89a].

For example, the following proof system can be proven to be a sound and complete characterisation of a simple logic programming language with parallel "or" and sequential "and" [And89a]:

$=$: $\dfrac{}{\mathbf{s} = \mathbf{s}}$ where $\mathbf{s}$ is closed

$\&$: $\dfrac{\mathbf{B} \quad \mathbf{C}}{\mathbf{B} \& \mathbf{C}}$

$\vee, 1$: $\dfrac{\mathbf{B}}{\mathbf{B} \vee \mathbf{C}}$

$\vee, 2$: $\dfrac{\mathbf{C}}{\mathbf{B} \vee \mathbf{C}}$

$\exists$: $\dfrac{\mathbf{B}[\mathbf{x} := \mathbf{t}]}{\exists \mathbf{x}(\mathbf{B})}$ where $\mathbf{t}$ is a closed term

$\mathbf{P}$: $\dfrac{\mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}{\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}$ where Π contains the definition $(\mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \leftrightarrow \mathbf{A}(\mathbf{x}_1, \ldots, \mathbf{x}_n))$

The judgments of this proof system are formulae; the characterisation theorem is that a query succeeds in the operational semantics if and only if its existential closure is derivable in this proof system. The proof system does not say anything about failing computations, however, because the problem of characterising failing computations in the presence of sequential "and" is essentially the dual of characterising successful computations in the presence of sequential "or".

In the remainder of this paper, I will take up this proof-theoretic view to develop similar characterisations of pure sequential Prolog with no cuts or negation. Along the way, both the problem of failure with sequential "and" and the problem of success with sequential "or" will be solved.

4 Operational Semantics

Our first task in this paper is to settle on exactly what we mean by the "sequential Prolog" strategy, in a manner which will be formal enough to be amenable to analysis. Sequential Prolog can be described as SLD-resolution with a left-to-right subgoal selection rule and a depth-first search rule. Since we want to focus on the logical interpretation of the connectives, however, it is more useful to develop a special-purpose operational semantics which gives rules telling how the interpreter decomposes a goal formula into its constituent subformulae.

The version of operational semantics that I will describe here is a straightforward formalisation of the standard "stack-of-stacks" implementation of a Prolog interpreter. Thus, most of the nomenclature from this section will be taken from that of interpreter technology.

Definition 4.1 A *goal stack* is just a sequence of formulae. We will generally use α, possibly subscripted, to stand for an arbitrary goal stack. We will write the concatenation of two goal stacks α_1 and α_2 as α_1, α_2.

A *closure* is an expression $(\theta : \alpha)$, where α is a goal stack and θ is a substitution. (In practice, θ will be some syntactic representation of a substitution on a finite number of variables.)

A *backtrack stack* is a sequence of closures. We will generally use β, possibly subscripted, to stand for an arbitrary backtrack stack. We will write the concatenation of two backtrack stacks β_1 and β_2 as $\beta_1;\beta_2$.

ϵ will represent the empty goal stack or backtrack stack (its use will be non-ambiguous).

There will be one operational semantics, called SP_Π, corresponding to each program Π. (Since the particular program being considered will usually not be important, we will usually drop the subscript.) SP is a rewriting system which consists of rules for a binary relation $\overset{\mathrm{SP}}{\Rightarrow}$ between backtrack stacks, and a definition of which backtrack stacks are to be considered "success states" and "failure states". The rewriting rules in SP are as follows:

1. &:
   $(\theta : (\mathbf{B}\&\mathbf{C}), \alpha);\beta \overset{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}, \mathbf{C}, \alpha);\beta$

2. $\vee$:
   $(\theta : (\mathbf{B} \vee \mathbf{C}), \alpha);\beta \overset{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}, \alpha);(\theta : \mathbf{C}, \alpha);\beta$

3. $\exists$:
   $(\theta : (\exists \mathbf{x}\mathbf{B}), \alpha);\beta \overset{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}'], \alpha);\beta$
   where $\mathbf{x}'$ is some variable not appearing to the left of the arrow

4. Defined predicates:
   $(\theta : \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n), \alpha);\beta \overset{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n), \alpha);\beta$
   where Π contains the definition $(\mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \leftrightarrow \mathbf{A}(\mathbf{x}_1, \ldots, \mathbf{x}_n))$

5. Unification, success:
   $(\theta : \mathbf{s} = \mathbf{t}, \alpha);\beta \overset{\mathrm{SP}}{\Rightarrow} (\theta\theta' : \alpha);\beta$
   where θ' is the mgu of $\mathbf{s}\theta$ and $\mathbf{t}\theta$

6. Unification, failure:
   $(\theta : \mathbf{s} = \mathbf{t}, \alpha);\beta \overset{\mathrm{SP}}{\Rightarrow} \beta$
   where $\mathbf{s}\theta$ and $\mathbf{t}\theta$ do not unify

The success states of SP are the backtrack stacks of the form

$$(\theta : \epsilon);\beta$$

The single failure state of SP is the empty backtrack stack, ϵ.

To execute a particular goal formula $\mathbf{A}$ in this interpreter, we form the backtrack stack consisting of the single closure $(() : \mathbf{A})$, where $()$ is the empty substitution. We then repeatedly apply the appropriate transitions (there will only be one appropriate one for any given stack). If we reach a failure state, we conclude that the query is unsatisfiable; if we reach a success state $(\theta : \epsilon);\beta$, we conclude that θ is a satisfying substitution for $\mathbf{A}$. Of course, for a given goal we might never reach a success or a failure state, due to repeated diverging applications of the defined predicate rule.

Definition 4.2 We say that a backtrack stack β *succeeds* if $\beta \stackrel{\mathrm{SP}}{\Rightarrow} \beta'$, where β' is a success state.

We say that β *fails* if $\beta \stackrel{\mathrm{SP}}{\Rightarrow} \epsilon$.

We say that β *diverges* if it neither succeeds nor fails.

We say that a query formula **A** succeeds (fails, diverges) if the backtrack stack $((): \mathbf{A})$ succeeds (fails, diverges), where () is the empty substitution.

The origin of the description of interpreters of this kind as "stack of stacks" interpreters should be clear. Nothing is relevant to the computation except the environment and the leftmost goal stack formula of the leftmost closure, so both the backtrack stack and the goal stacks within it can be implemented as true stacks (for instance, as LISP lists).

Note the difference between a step in this operational semantics and a step in SLD-resolution. An SLD-resolution step essentially combines a subgoal selection, clause selection, renaming, unification, and defined predicate expansion. In SP, since we are interested in the interpretation of the individual logical constructors, we analyse these notions into separate rules: the and, or, existential, unification, and defined predicate rules respectively.

This operational semantics can be easily proved to be equivalent to the depth-first, left-to-right version of SLD-resolution (modulo the translation between Horn clause programs and the form of programs used here); so SP really is just a *convenient* variant of SLD-resolution with a left-to-right subgoal selection rule and a depth-first search rule. Its convenience to our purpose may become clearer later, when we see how the completely formalised computations of SP, and its emphasis on decomposition of formulae, facilitates the proofs of important theorems.

We can make some observations about success and failure of goals in SP which reinforce the view that computations in SP really are talking about the existence of solutions to first order logic queries. These theorems will also be important later when we prove the soundness and completeness of various characterisations of SP.

Theorem 4.3 If the backtrack stack $(\theta : \alpha)$ fails, then for all substitutions ρ, $(\theta\rho : \alpha)$ also fails, with a computation with a smaller or equal number of steps.

Proof. By induction on the length of the computation of $(\theta : \alpha)$. At the base of the induction (0 steps), there are no formulae in the goal stack, so the backtrack stack does not fail; so the implication holds trivially. The cases when the goal stack is not empty are on the topmost formula in the goal stack.

Case $\mathbf{s} = \mathbf{t}$: If the unification of $\mathbf{s}\theta$ and $\mathbf{t}\theta$ fails, then it will fail under any more specific substitution. Otherwise, we have the computation

$$(\theta : \mathbf{s} = \mathbf{t}, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta\theta' : \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

where θ' is the mgu of $\mathbf{s}\theta$ and $\mathbf{t}\theta$. We also have that either $\mathbf{s}\theta\rho$ and $\mathbf{t}\theta\rho$ do not unify, or else

$$(\theta\rho : \mathbf{s} = \mathbf{t}, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta\rho\theta'' : \alpha)$$

But recall that by the definition of mgu, since $\mathbf{s}\theta(\rho\theta'') \equiv \mathbf{t}\theta(\rho\theta'')$, there is a ρ' such that $\theta\theta'\rho' \equiv \theta(\rho\theta'')$. Now, by the induction hypothesis we have that

$$(\theta\theta'\rho' : \alpha) \equiv (\theta\rho\theta'' : \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

Cases $\mathbf{B\&C}$, $\exists \mathbf{x}\ \mathbf{B}$, $\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$: The result is a straightforward consequence of the induction hypothesis.

Case $\mathbf{B} \vee \mathbf{C}$: we have the computation

$$(\theta : \mathbf{B} \vee \mathbf{C}, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}, \alpha); (\theta : \mathbf{C}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta : \mathbf{C}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

Both $(\theta : \mathbf{B}, \alpha)$ and $(\theta : \mathbf{C}, \alpha)$ fail independently; so applying the induction hypothesis twice,

$$(\theta\rho : \mathbf{B} \vee \mathbf{C}, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta\rho : \mathbf{B}, \alpha); (\theta\rho : \mathbf{C}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta\rho : \mathbf{C}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

with a smaller or equal number of steps.

□

Theorem 4.4 If $(\theta : \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta_f : \epsilon); \beta$, then, for any θ' more specific than θ but more general than θ_f (that is, for any θ' such that there is a ρ such that $\theta\theta'\rho \equiv \theta_f$), there is a β' such that $(\theta\theta' : \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta_f : \epsilon); \beta'$, with a smaller or equal number of steps.

Proof. By induction on the length of the SP-computation. Cases are on the first step of the computation.

Case $\mathbf{s} = \mathbf{t}$: We know that for some ρ',

$$(\theta\theta' : \mathbf{s} = \mathbf{t}, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta\theta'\rho' : \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta_f : \epsilon); \beta$$

because $\mathbf{s}$ and $\mathbf{t}$ certainly unify under θ_f (we can prove this by a simple induction).

Cases $\mathbf{B\&C}$, $\exists \mathbf{x}\ \mathbf{B}$, $\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$: The result is a straightforward consequence of the induction hypothesis.

Case $\mathbf{B} \vee \mathbf{C}$: we have the computation

$$(\theta : \mathbf{B} \vee \mathbf{C}, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}, \alpha); (\theta : \mathbf{C}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta_f : \epsilon); \beta$$

Now, if $(\theta : \mathbf{B}, \alpha)$ succeeds, the result follows from the induction hypothesis. If $(\theta : \mathbf{B}, \alpha)$ fails, then (by the previous theorem) it will fail under any more specific substitution, with a smaller or equal number of steps; also, $(\theta : \mathbf{C}, \alpha)$ must succeed, so the result follows from the induction hypothesis.

□

Theorem 4.5 If $(\theta : \alpha)$ succeeds, then for all variables $\mathbf{x}$ free in α but not mentioned in θ, there is a closed term $\mathbf{r}$ such that $(\theta[\mathbf{x} := \mathbf{r}] : \alpha)$ succeeds.

Proof. Let the final goal stack in the successful computation be $(\theta_f : \epsilon); \beta$. Select $\mathbf{r}$ to be a closed instance of whatever θ_f assigns to $\mathbf{x}$. (We use here the assumption that there is at least one term in the language.) Now consider the goal stack $(\theta : \alpha, \mathbf{x} = \mathbf{r})$; this succeeds with final goal stack $(\theta_f[\mathbf{x} := \mathbf{r}] : \epsilon); \beta'$. We know, from the properties of composition of substitutions, that $\theta[\mathbf{x} := \mathbf{r}]$ composed with $\theta_f[\mathbf{x} := \mathbf{r}]$ will give us $\theta_f[\mathbf{x} := \mathbf{r}]$; that is, $\theta[\mathbf{x} := \mathbf{r}]$ is more general than $\theta_f[\mathbf{x} := \mathbf{r}]$. By the previous theorem, the result holds.

□

These theorems tend to reinforce our intuitive view of the effect of computations. When we give a query to the system, we are essentially asking whether some instance of it is true. If it succeeds, we conclude that there is such an instance; if it fails, we conclude that there is not. The first theorem essentially says that if a formula fails, there is no instance of it that succeeds; that is, the failure is "sound". The second theorem says that if a formula succeeds under a given substitution θ, then it succeeds under all approximations to the solution substitution from θ. The third theorem shows that each free variable in a successful query has a closed term which acts as a "witness" for it.

5 Unfoldings of Formulae

The notion of the disjunctive unfolding of a formula is the main novelty of this paper, and the mechanism which captures the non-compositionality of sequential Prolog. The disjunctive unfolding of a formula **A** is a formula **A'** which is classically equivalent to **A**, but has the property that its satisfiability depends only on the satisfiability of its subformulae.

This requires some motivation. Once we have set out to develop a proof system characterising sequential Prolog, there is one fairly natural way to proceed (which has been followed independently, for example, by Girard [Gir87]). However, the resulting proof system still has completeness problems; as with full first order logic, we can still prove things which have no corresponding computation.

Unfoldings of formulae are exactly what we need to solve these completeness problems. This section will present the idea of unfoldings by giving an outline of the initial attempt at a proof system, describing that system's problems, defining the predicate and disjunctive unfoldings of a formula, and proving some essential properties of unfoldings.

5.1 An Initial Attempt at a Characterisation

Our goal is to give natural-deduction-style rules for each connective that characterise the behaviour of that connective with respect to success or failure in the sequential Prolog interpreter. For instance, in informal language, **B&C** succeeds if both **B** and **C** succeed, and **B&C** fails if either **B** fails, or **B** succeeds and **C** fails. We would like the proof system to reflect the compositionality and simplicity of this informal description.

We therefore need two kinds of judgment, one for failure and one for success of a query formula. These "signed formulae" will be the basic elements of all the subsequent proof systems.

Definition 5.1 A *signed formula* is an expression of the form $S(\mathbf{A})$ (read "**A** succeeds") or of the form $F(\mathbf{A})$ (read "**A** fails"). We use $\sigma(\mathbf{A})$ to denote an arbitrary signed formula.

I have chosen this notation, rather than use **A** and $\neg$**A** for $S(\mathbf{A})$ and $F(\mathbf{A})$ respectively, to avoid confusion with the issue of negation as failure in programs (not addressed by this paper). Signed formulae will be used in the rest of the proof systems mentioned in this paper.

Here is an initial attempt at describing the rules for the characterising proof system, similar to one described by Girard [Gir87]. We want $S(\mathbf{A})$ to be derivable when **A**

succeeds, and $F(\mathbf{A})$ when $\mathbf{A}$ fails, so we choose the following rules for conjunction:

$$\frac{S(\mathbf{B}) \quad S(\mathbf{C})}{S(\mathbf{B\&C})} \qquad \frac{F(\mathbf{B})}{F(\mathbf{B\&C})} \qquad \frac{S(\mathbf{B}) \quad F(\mathbf{C})}{F(\mathbf{B\&C})}$$

The behaviour of $\mathbf{B\&C}$ under the two "signs" reflects the informal description above. When $\mathbf{B}$ neither succeeds nor fails (ie. diverges) we can say nothing about whether $\mathbf{B\&C}$ succeeds or fails, which reflects the sequential strategy perfectly. The rules for disjunction are the duals of these rules:

$$\frac{S(\mathbf{B})}{S(\mathbf{B} \vee \mathbf{C})} \qquad \frac{F(\mathbf{B}) \quad S(\mathbf{C})}{S(\mathbf{B} \vee \mathbf{C})} \qquad \frac{F(\mathbf{B}) \quad F(\mathbf{C})}{F(\mathbf{B} \vee \mathbf{C})}$$

We would also give rules for equality and existential quantifier, but even with the rules so far we have problems.

Although we have been trying to get a proof system weaker than first order logic to reflect the weakness of sequential Prolog, these rules are still too strong; they are not sound with respect to the operational semantics (or, equivalently, the operational semantics is not complete with respect to them). Consider a program with a definition of a predicate *Loop*, which always causes the program to go into an infinite loop:

$$Loop() \leftrightarrow Loop()$$

Now consider the query $(true \vee Loop())\&false$. This query diverges according to the operational semantics; however, with the rules given above, we can prove that it fails.

$$\frac{\dfrac{\overline{S(true)}}{S(true \vee Loop())} \quad \overline{F(false)}}{F((true \vee Loop())\&false)}$$

But now consider the query $(true\&false) \vee (Loop()\&false)$. This is classically equivalent to the previous query, and is handled in the same way by the interpreter. However, we cannot get a derivation of $F((true\&false) \vee (Loop()\&false))$; in other words, the proof system behaves *correctly* in regard to this query.

Basically, if a query has all its disjunctions outside all its conjunctions and existential quantifiers, then it will be handled correctly by the simple proof rules above. Many queries will not have this property. However, a query can be transformed in this direction by taking the first disjunction encountered in a depth-first traversal of the formula, and "pulling it toward the outside of the formula" by distributing the conjunctions and quantifiers over it. There is the added complication that disjunctions may be "hidden" inside predicate calls, but this can be handled by expanding predicate calls.

5.2 Predicate and Disjunctive Unfoldings

The above discussion motivates the following definitions.

Definition 5.2 The *key subformula* of a formula is the leftmost disjunction or predicate application subformula which is not inside another disjunction. (See Figure 1.) Not every formula has a key subformula.

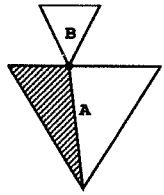

Figure 1: **B** is the key subformula of formula **A**. Informally, the shaded region consists only of $=$, $\&$, and $\exists$ formulae.

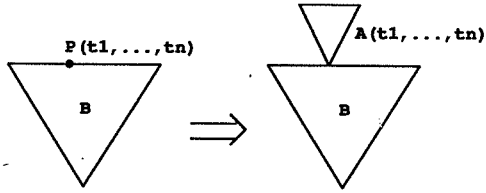

Figure 2: One predicate unfolding of formula **B**, where **P** is defined by $\mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \leftrightarrow \mathbf{A}(\mathbf{x}_1, \ldots, \mathbf{x}_n)$.

A formula $\mathbf{A}'$ is a *predicate unfolding* of a formula **A** if it is obtained from **A** by expanding some predicate application subformula into its body. (See Figure 2.) We write $\mathbf{A} \stackrel{\text{Pred}}{\Rightarrow} \mathbf{A}'$ if $\mathbf{A}'$ is a predicate unfolding of **A**.

The *disjunctive unfolding* of a formula **A** with key subformula $\mathbf{B} \vee \mathbf{C}$ is the formula $\mathbf{A}^{\mathbf{B}} \vee \mathbf{A}^{\mathbf{C}}$, where $\mathbf{A}^{\mathbf{B}}$ (resp. $\mathbf{A}^{\mathbf{C}}$) means **A** with its key subformula replaced by **B** (resp. **C**). (See Figure 3.) Formulae which do not have a disjunctive key subformula have no disjunctive unfolding; we write $\mathbf{A} \stackrel{\text{Disj}}{\Rightarrow} \mathbf{A}'$ if $\mathbf{A}'$ is the disjunctive unfolding of **A**.

As we will see in future sections, we can make the simple proof rules given in section 5.1 work properly if we first restrict certain formulae in the rules to be formulae without predicate application subformulae, and then add a rule allowing a formula to be transformed by a predicate or disjunctive unfolding.

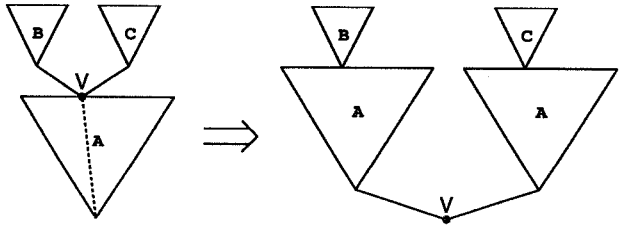

Figure 3: Disjunctive unfolding of a formula. The indicated disjunction, $\mathbf{B} \vee \mathbf{C}$, is the key subformula of **A**.

5.3 Some Facts about Unfoldings

For now, we must show that predicate and disjunctive unfoldings of a formula are equivalent to that formula under the operational semantics. These facts have some interest in themselves, but will also be necessary to prove some soundness and completeness results.

Theorem 5.3 (Operational Equivalence of Predicate Unfoldings) If $\mathbf{A} \overset{\text{Pred}}{\Rightarrow} \mathbf{A}'$, then $\beta_1;(\theta : \alpha_1, \mathbf{A}, \alpha_2);\beta_2$ succeeds (fails)
iff $\beta_1;(\theta : \alpha_1, \mathbf{A}', \alpha_2);\beta_2$ succeeds (fails).

Proof. By induction on the length of the computation. The only non-trivial case is when $\mathbf{A}$ is itself a predicate call, and it is at the left of the leftmost goal stack. In this case, it should be clear that a predicate call expansion step can be inserted or deleted to form one computation from the other.
□

So predicate unfoldings of $\mathbf{A}$ work the same way as $\mathbf{A}$. We must also prove that disjunctive unfoldings of $\mathbf{A}$ work the same way as $\mathbf{A}$. Unfortunately, this proof is not as straightforward as the one above, and involves a long case analysis.

Theorem 5.4 (Operational Equivalence of Disjunctive Unfoldings) If $\mathbf{A} \overset{\text{Disj}}{\Rightarrow} \mathbf{A}'$, then $(\theta : \mathbf{A}, \alpha);\beta$ succeeds (fails) iff $(\theta : \mathbf{A}', \alpha);\beta$ succeeds (fails).

Proof. ($\rightarrow$) By induction on the depth of the key subformula (which is a disjunction) in the tree of the formula $\mathbf{A}$. In what follows, I use the notation $\beta \overset{\text{SP}^*}{\Rightarrow} S/F$ to denote that β succeeds or fails, its meaning being consistent through the analysis of each case.

Case depth = 0: $\mathbf{A}$ is itself a disjunction, so its disjunctive unfolding is itself. The result holds trivially.

Case depth > 0: $\mathbf{A}$ can be either a conjunction or an existential formula.

- $\mathbf{A} \equiv \mathbf{B}\&\mathbf{C}$. We have the computation

$$(\theta : \mathbf{B}\&\mathbf{C}, \alpha);\beta \overset{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}, \mathbf{C}, \alpha);\beta \overset{\text{SP}^*}{\Rightarrow} S/F$$

  There are two subcases to consider.

  - The key subformula, $\mathbf{B}_1 \vee \mathbf{B}_2$, is in $\mathbf{B}$. The disjunctive unfolding of $\mathbf{B}$ is therefore $\mathbf{B}^{\mathbf{B}_1} \vee \mathbf{B}^{\mathbf{B}_2}$, and the disjunctive unfolding of $\mathbf{A}$ is $(\mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}) \vee (\mathbf{B}^{\mathbf{B}_2}\&\mathbf{C})$. By the induction hypothesis, we have that

$$(\theta : \mathbf{B}^{\mathbf{B}_1} \vee \mathbf{B}^{\mathbf{B}_2}, \mathbf{C}, \alpha);\beta \overset{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}^{\mathbf{B}_1}, \mathbf{C}, \alpha);(\theta : \mathbf{B}^{\mathbf{B}_2}, \mathbf{C}, \alpha);\beta \overset{\text{SP}^*}{\Rightarrow} S/F$$

    But then

$$(\theta : (\mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}) \vee (\mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}), \alpha);\beta \overset{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}, \alpha);(\theta : \mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}, \alpha);\beta \overset{\text{SP}}{\Rightarrow}$$

$$(\theta : \mathbf{B}^{\mathbf{B}_1}, \mathbf{C}, \alpha);(\theta : \mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}, \alpha);\beta \overset{\text{SP}^*}{\Rightarrow} S/F$$

    because we can insert the step for splitting the second conjunction at the appropriate point if necessary. So, since $(\mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}) \vee (\mathbf{B}^{\mathbf{B}_2}\&\mathbf{C})$ is the disjunctive unfolding of $\mathbf{A}$, the result holds.

– The key subformula, $\mathbf{C}_1 \vee \mathbf{C}_2$, is in $\mathbf{C}$. Thus $\mathbf{B}$ contains no disjunctions or predicate calls, and the disjunctive unfolding of $\mathbf{A}$ is $(\mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}) \vee (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_2})$. We have that

$$(\theta : \mathbf{B}, \mathbf{C}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* (\theta' : \mathbf{C}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* S/F$$

(because $\mathbf{B}$ contains no disjunctions, it cannot lengthen the backtrack stack). Thus, by the induction hypothesis, we have that

$$(\theta' : \mathbf{C}^{\mathbf{C}_1} \vee \mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow} (\theta' : \mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta' : \mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* S/F$$

But then

$$(\theta : (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}) \vee (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}), \alpha); \beta \overset{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}$$

$$(\theta : \mathbf{B}, \mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* (\theta' : \mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta$$

$$\overset{\text{SP}}{\Rightarrow}^* S/F$$

because we can insert the computation for deriving $(\theta' : \mathbf{C}^{\mathbf{C}_2}, \alpha); \beta$ from $(\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta$ at the appropriate point if necessary. So, since $(\mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}) \vee (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_2})$ is the disjunctive unfolding of $\mathbf{A}$, the result holds.

• $\mathbf{A} \equiv \exists \mathbf{x} \mathbf{B}$. The disjunctive unfolding is $\exists \mathbf{x} \mathbf{B}^{\mathbf{B}_1} \vee \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_2}$. We have the computation

$$(\theta : \exists \mathbf{x} \mathbf{B}, \alpha); \beta \overset{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}'], \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* S/F$$

By the induction hypothesis, we have that

$$(\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}']^{\mathbf{B}_1} \vee \mathbf{B}[\mathbf{x} := \mathbf{x}']^{\mathbf{B}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}$$

$$(\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}']^{\mathbf{B}_1}, \alpha); (\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}']^{\mathbf{B}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* S/F$$

But then

$$(\theta : \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_1} \vee \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow} (\theta : \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_1}, \alpha); (\theta : \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}$$

$$(\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}']^{\mathbf{B}_1}, \alpha); (\theta : \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_2}, \alpha); \beta \overset{\text{SP}}{\Rightarrow}^* S/F$$

because we can insert the step for discharging the existential quantifier at the appropriate point if necessary (we can even use the same new variable, $\mathbf{x}'$, because if we get to that point, all other occurrences of that variable will have disappeared due to failure). So, since $\exists \mathbf{x} \mathbf{B}^{\mathbf{B}_1} \vee \exists \mathbf{x} \mathbf{B}^{\mathbf{B}_2}$ is the disjunctive unfolding of $\mathbf{A}$, the result holds.

This completes the proof of the forward direction of the theorem statement. We now move on to the proof of the converse.

($\leftarrow$) By induction on the depth of the key subformula (which is a disjunction) in the tree of the formula $\mathbf{A}$.

Case depth = 0: $\mathbf{A}$ is itself a disjunction, so its disjunctive unfolding is itself. The result holds trivially.

Case depth > 0: $\mathbf{A}$ can be either a conjunction or an existential formula.

- $\mathbf{A} \equiv \mathbf{B}\&\mathbf{C}$. There are two subcases to consider.

  - The key subformula, $\mathbf{B}_1 \vee \mathbf{B}_2$, is in $\mathbf{B}$. The disjunctive unfolding of $\mathbf{A}$ is therefore $(\mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}) \vee (\mathbf{B}^{\mathbf{B}_2}\&\mathbf{C})$. We have the computation

    $$(\theta : (\mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}) \vee (\mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}), \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}^{\mathbf{B}_1}\&\mathbf{C}, \alpha); (\theta : \mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow}$$

    $$(\theta : \mathbf{B}^{\mathbf{B}_1}, \mathbf{C}, \alpha); (\theta : \mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

    But then we must have the computation

    $$(\theta : \mathbf{B}^{\mathbf{B}_1} \vee \mathbf{B}^{\mathbf{B}_2}, \mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}^{\mathbf{B}_1}, \mathbf{C}, \alpha); (\theta : \mathbf{B}^{\mathbf{B}_2}, \mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

    because the effect of the goal stack $(\mathbf{B}^{\mathbf{B}_2}, \mathbf{C}, \alpha)$ is identical to that of $(\mathbf{B}^{\mathbf{B}_2}\&\mathbf{C}, \alpha)$ So by the induction hypothesis, we have that

    $$(\theta : \mathbf{B}, \mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

    So, adding one step onto the front of the computation,

    $$(\theta : \mathbf{B}\&\mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

  - The key subformula, $\mathbf{C}_1 \vee \mathbf{C}_2$, is in $\mathbf{C}$. $\mathbf{B}$ is therefore non-disjunctive, and the disjunctive unfolding of $\mathbf{A}$ is $(\mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}) \vee (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_2})$. We have the computation

    $$(\theta : (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}) \vee (\mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}), \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow}$$

    $$(\theta : \mathbf{B}, \mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta' : \mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha); \beta$$
    $$\stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

    (The last sequence of steps does not lengthen the backtrack stack because $\mathbf{B}$ is a non-disjunctive formula.) But then we must have the computation

    $$(\theta' : \mathbf{C}^{\mathbf{C}_1} \vee \mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow} (\theta' : \mathbf{C}^{\mathbf{C}_1}, \alpha); (\theta' : \mathbf{C}^{\mathbf{C}_2}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

    because $(\theta : \mathbf{B}\&\mathbf{C}^{\mathbf{C}_2}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta' : \mathbf{C}^{\mathbf{C}_2}, \alpha)$. So by the induction hypothesis, we have that

    $$(\theta' : \mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

    So, adding some steps onto the front of the computation,

    $$(\theta : \mathbf{B}\&\mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}, \mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta' : \mathbf{C}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

- $\mathbf{A} \equiv \exists \mathbf{x}\mathbf{B}$. The disjunctive unfolding is $(\exists \mathbf{x}\mathbf{B}^{\mathbf{B}_1}) \vee (\exists \mathbf{x}\mathbf{B}^{\mathbf{B}_2})$. We have the computation

  $$(\theta : (\exists \mathbf{x}\mathbf{B}^{\mathbf{B}_1}) \vee (\exists \mathbf{x}\mathbf{B}^{\mathbf{B}_2}), \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \exists \mathbf{x}\mathbf{B}^{\mathbf{B}_1}, \alpha); (\theta : \exists \mathbf{x}\mathbf{B}^{\mathbf{B}_2}, \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow}$$

  $$(\theta : \mathbf{B}^{\mathbf{B}_1}[\mathbf{x} := \mathbf{x}'], \alpha); (\theta : \exists \mathbf{x}\mathbf{B}^{\mathbf{B}_2}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

But then we must have the computation

$$(\theta : \mathbf{B}^{\mathbf{B}_1}[\mathbf{x} := \mathbf{x}'] \vee \mathbf{B}^{\mathbf{B}_2}[\mathbf{x} := \mathbf{x}'], \alpha); \beta \stackrel{\mathrm{SP}}{\Rightarrow}$$

$$(\theta : \mathbf{B}^{\mathbf{B}_1}[\mathbf{x} := \mathbf{x}'], \alpha); (\theta : \mathbf{B}^{\mathbf{B}_2}[\mathbf{x} := \mathbf{x}'], \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

because $(\theta : \exists \mathbf{x}\ \mathbf{B}^{\mathbf{B}_2}, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta : \mathbf{B}^{\mathbf{B}_2}[\mathbf{x} := \mathbf{x}'], \alpha)$. So by the induction hypothesis, we have that

$$(\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}'], \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

So, adding one step onto the front of the computation,

$$(\theta : \exists \mathbf{x} \mathbf{B}, \alpha); \beta \stackrel{\mathrm{SP}^*}{\Rightarrow} S/F$$

This completes the proof of the converse direction.

□

Finally, we should note that the disjunctive unfolding of a formula is classically equivalent to the formula; this follows from the distributivity of & and $\exists$ over $\vee$. The corresponding result for predicate unfoldings also holds, if we take the universal closures of the predicate definitions as assumptions.

In summary, the predicate and disjunctive unfolding operations are transformations on formulae which preserve the classical meaning and the operational behaviour of the original formulae. These important properties will allow us to use unfoldings to develop sound and complete proof systems, as we will see in the next sections.

6 A Complete, Infinitary Proof System

With the notion of unfoldings in hand, we can give a sound and complete proof system for sequential Prolog. This section presents and discusses that system, and proves soundness, completeness, and related results.

This complete proof system, however, is *infinitary*: there is a rule with an infinite number of premisses. It must be infinitary because the problem we have set ourselves turns out to be vulnerable to a version of Gödel's incompleteness theorem (the proof of this is in Section 8). This incompleteness is not specific to the proof-theoretic approach, but turns up in other guises in all methods for proving properties of Prolog programs. It is annoying, but not really a problem from a mathematical point of view. The system exhibits a well-known form of incompleteness, which can be handled exactly by giving the kind of infinitary rule we have here.

The incompleteness of the system is a *practical* problem, of course, since in general infinitary derivations cannot be written down or represented on a computer. In the next section I give a proof system which is finitary and has a limited form of completeness, although it is not fully complete.

6.1 The Proof System SPndi

The judgments of the infinitary proof system, SPndi (Sequential Prolog, natural deduction, infinitary) are just signed formulae as described in section 5.1. The rules for SPndi are listed in Figure 4.

1. Success rules.

$S(=)$: $\dfrac{}{S(\mathbf{t}=\mathbf{t})}$ $\quad$ $S(\exists)$: $\dfrac{S(\mathbf{B}[\mathbf{x}:=\mathbf{t}])}{S(\exists\mathbf{x}\mathbf{B})}$ $(*a)$

$S(\&)$: $\dfrac{S(\mathbf{B}) \quad S(\mathbf{C})}{S(\mathbf{B}\&\mathbf{C})}$

$S(\vee),1$: $\dfrac{S(\mathbf{B})}{S(\mathbf{B}\vee\mathbf{C})}$ $\quad$ $S(\vee),2$: $\dfrac{F(\mathbf{B}) \quad S(\mathbf{C})}{S(\mathbf{B}\vee\mathbf{C})}$

2. Failure rules.

$F(=)$: $\dfrac{}{F(\mathbf{s}=\mathbf{t})}$ $(*b)$ $\quad$ $F(\exists)$: $\dfrac{F(\mathbf{B}[\mathbf{x}:=\mathbf{y}])}{F(\exists\mathbf{x}\mathbf{B})}$ $(*c)$

$F(\&),1$: $\dfrac{F(\mathbf{B})}{F(\mathbf{B}\&\mathbf{C})}$ $\quad$ $F(\&),2$: $\dfrac{F(\mathbf{C})}{F(\mathbf{B}\&\mathbf{C})}$ $(*a)$

$F(\vee)$: $\dfrac{F(\mathbf{B}) \quad F(\mathbf{C})}{F(\mathbf{B}\vee\mathbf{C})}$

3. Unfolding rule.

Unfold: $\dfrac{\sigma(\mathbf{A}')}{\sigma(\mathbf{A})}$

where $\mathbf{A}'$ is a predicate or disjunctive unfolding of $\mathbf{A}$

4. Infinitary rule.

Inf: $\dfrac{\sigma(\mathbf{A}[\mathbf{x}:=\mathbf{t}_1]) \quad \sigma(\mathbf{A}[\mathbf{x}:=\mathbf{t}_2]) \quad \ldots}{\sigma(\mathbf{A})}$

where $\mathbf{t}_1, \mathbf{t}_2, \ldots$ is an enumeration of all the terms in the language

Side-conditions:
$(*a)$: $\mathbf{B}$ contains no predicate calls
$(*b)$: $\mathbf{s}$ and $\mathbf{t}$ are non-identical closed terms
$(*c)$: the derivation of the upper signed formula contains no predicate unfolding steps, and $\mathbf{y}$ does not occur free in $\mathbf{B}$

Figure 4: The complete, infinitary proof system SPndi.

I will discuss the rules of SPndi in groups. Firstly, the rules for connectives and equality *look* sensible and understandable, and are closely related to the corresponding rules in first order natural deduction proof systems. The only things which make the rules weaker than first order rules are the non-commutativity of & and ∨, and the extra side-conditions.

I would call the readers' attention especially to the rules $S(\exists)$ and $F(\exists)$. These rules are almost identical to the rules for $\exists$ on the right- and left-hand side of sequents in the sequent calculus. This is significant, because it means that we have managed to obviate the need to use substitutions (or syntactic representations thereof) and formalisations of unification as essential parts of the characterisation. This distinguishes the proof system from all other characterisations of sequential Prolog that I am aware of, and affirms that Prolog computations really *are* talking about witnesses for existentially quantified variables, in a logic which is not far from first order logic.

Note the side-condition on the rules $S(\exists)$ and $F(\&)$: "where **B** contains no predicate calls". (I am using the phrase "predicate calls" to mean "applications of predicates other than =", of course.) This condition would seem, at first glance, to make the system incomplete, since there are many formulae which do not meet it. However, note that any such excluded formula can be the conclusion of an application of the unfoldings rule. As we will see in the completeness proof, we can use unfoldings to convert problem formulae into the right form. The same remarks apply to the rule $F(\exists)$, which has a similar but slightly weaker side-condition.

The unfoldings rule is the only rule which is non-compositional, in the sense that the premisses of the rule are not closely related to the immediate subformulae of the conclusion. This fact affirms that the notions of predicate and disjunctive unfolding capture all of the non-compositionality of sequential Prolog, and do so in a way that allows the rest of the characterisation to be logical.

Finally, consider the infinitary rule. The non-existence of a complete proof system arises, in part, from the presence of free variables in the judgments – which is, in turn, required by the logical form of the $F(\exists)$ rule. This is the kind of incompleteness that arises in proof systems for first order arithmetic, for example, and there it can be handled by the analogous rule.

The infinitary rule therefore says something significant about the *other* rules in the proof system: it says that the incompleteness of the system without the infinitary rule does not arise from any inadequacies in the form of the other rules, but only from the presence of free variables. As the unfoldings rule captures all of the non-compositionality of the problem, the infinitary rule captures all of the essential incompleteness of the problem. (For more discussion about the use and implications of infinitary rules in the logics of programs, see section 1.7 of Goldblatt's book [Gol82]).

Note that if the base language in which we are programming is finite, the "infinitary" rule is then not infinitary at all. This case does not come up in most logic programming, but it does come up when we are trying to model database applications with logic programming, using Datalog [Ull86].

6.2 Soundness

SPndi is sound: if a closed formula is derivable, it is "valid". The notion of validity here, however, comes not from first order logic, but from the operational semantics. There is a more general semantic notion, corresponding approximately to truth in a model; I use the verb "to work" for this, in order to make the operational connection clear. Validity is then defined in terms of "working".

Definition 6.1 A signed formula *works* with respect to a substitution θ just in the following two cases: $S(\mathbf{A})$ works wrt θ if the backtrack stack $(\theta : \mathbf{A})$ succeeds in the operational semantics SP; and $F(\mathbf{A})$ works wrt θ if $(\theta : \mathbf{A})$ fails in SP.

A signed formula $\sigma(\mathbf{A})$ is *valid* if, for every substitution θ such that $\mathbf{A}\theta$ is closed, $\sigma(\mathbf{A})$ works with respect to θ.

I will take the approach of proving soundness in three stages: first, proving that F-signed formulae with derivations containing no predicate expansion steps have a property slightly stronger than validity; second, proving that derivations of any F-signed formula are sound; and third, proving that derivations of any S-signed formula are sound.

Lemma 6.2 (Soundness of F Derivations) If $F(\mathbf{A})$ has a derivation, then (1) for all θ such that $\mathbf{A}\theta$ is closed, $(\theta : \mathbf{A})$ fails; and (2) if the derivation has no predicate unfolding steps, then for *all* θ, $(\theta : \mathbf{A})$ fails without performing any Defined Predicate steps.

Proof. By induction on the structure of the derivation. Cases are on the bottommost rule applied in the derivation.

$F(\mathbf{s} = \mathbf{t})$: trivial.

$F(\exists \mathbf{x}\ \mathbf{B})$: The derivation has no predicate unfoldings, so it suffices to prove (2). By the induction hypothesis, the backtrack stack $(\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}'])$ fails without performing any Defined Predicate steps; but because $(\theta : \exists \mathbf{x}\ \mathbf{B}) \stackrel{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}'])$, so does $(\theta : \exists \mathbf{x}\ \mathbf{B})$.

$F(\mathbf{B}\&\mathbf{C})$, 1: (1) By the induction hypothesis, $(\theta : \mathbf{B})$ fails. If we take the failing computation and graft $\mathbf{C}$ onto the right-hand ends of all goal stacks, we get a failing computation for $(\theta : \mathbf{B}, \mathbf{C})$, because $\mathbf{C}$ never gets to the top of any goal stack (otherwise the computation of $(\theta : \mathbf{B})$ would succeed). (2) Similar.

$F(\mathbf{B}\&\mathbf{C})$, 2: (1) Because $\mathbf{C}$ fails under θ, in fact any backtrack stack with closures of the form $(\theta' : \alpha, \mathbf{C})$ fails, where θ' is more specific than θ and α contains no predicate calls. This is because either α itself fails under θ', or α succeeds producing a θ'' which differs from θ only at variables which are not free in $\mathbf{C}$ (and therefore do not affect the computation of $\mathbf{C}$). α cannot diverge, because it has no predicate calls. (2) Similar. $\mathbf{B}$ either succeeds or fails under any substitution, and every computation must pass through $\mathbf{C}$; but by the induction hypothesis, $\mathbf{C}$ fails under every substitution.

$F(\mathbf{B} \vee \mathbf{C})$: (1) Both $\mathbf{B}$ and $\mathbf{C}$ fail under θ, so the closures in the backtrack stack $(\theta : \mathbf{B}); (\theta : \mathbf{C})$ fail one after the other. (2) Similar.

Predicate unfolding rule: We need only consider (1). By theorem 5.3, if $\mathbf{A}'$ is some predicate unfolding of $\mathbf{A}$, then we can substitute any occurrence of $\mathbf{A}$ for $\mathbf{A}'$, and vice versa, in any backtrack stack, and get the same result. Thus $(\theta : \mathbf{A})$ fails iff $(\theta : \mathbf{A}')$ fails.

Disjunctive Unfolding rule: By theorem 5.4, if $\mathbf{A}'$ is some disjunctive unfolding of $\mathbf{A}$, then we can substitute any occurrence of $\mathbf{A}$ for $\mathbf{A}'$, and vice versa, in any backtrack stack,

and get the same result. Thus $(\theta : \mathbf{A})$ fails iff $(\theta : \mathbf{A}')$ fails, under either the assumptions in (1) or those in (2). The property of not performing any Defined Predicate steps is preserved, by a generalisation of theorem 5.4.

Infinitary rule: (1) Assume the converse; that is, that all the premisses are valid but the conclusion is invalid. This means that there is some closed-term substitution of the form $\theta[\mathbf{x} := \mathbf{t}]$, with respect to which $F(\mathbf{A})$ does not work. But one of the premisses is of the form $\mathbf{A}[\mathbf{x} := \mathbf{t}]$, and this must work with respect to θ; contradiction. (2) $(\theta : \mathbf{A})$ cannot succeed, because that would contradict theorem 4.5. If the computation reaches a point at which the top of the topmost stack is a predicate call, however, then (by an easy generalisation of theorem 4.5) there must be a $\mathbf{t}$ such that $(\theta[\mathbf{x} := \mathbf{t}] : \mathbf{A})$ reaches such a point as well. By the induction hypothesis, all $(\theta[\mathbf{x} := \mathbf{t}] : \mathbf{A})$ fail without performing any Defined Predicate steps; so the computation of $(\theta : \mathbf{A})$ cannot reach such a point, and thus cannot diverge; therefore it must fail, having performed no Defined Predicate steps. □

Lemma 6.3 (Soundness of S Derivations) If $S(\mathbf{A})$ has a derivation, then it is valid.

Proof. Assume that $\sigma(\mathbf{A})$ is derivable. We must prove that for any substitution θ such that $\mathbf{A}\theta$ is closed, $(\theta : \mathbf{A})$ succeeds. We do this by induction on the structure of the SPndi-derivation. Cases are on the bottommost rule applied in the derivation.

$S(\mathbf{t} = \mathbf{t})$: trivial.

$S(\exists \mathbf{x}\ \mathbf{B})$: Assume that $S(\mathbf{B}[\mathbf{x} := \mathbf{t}])$ works wrt θ, but that $S(\exists \mathbf{x}\mathbf{B})$ does not work; then prove a contradiction. Since $(\theta : \exists \mathbf{x}\mathbf{B})$ does not succeed, it must fail, because there are no predicate calls in $\mathbf{B}$ which would make it diverge. But (by theorem 4.3) since $(\theta : \mathbf{B}[\mathbf{x} := \mathbf{x}'])$ fails, $(\theta[\mathbf{x}' := \mathbf{t}] : \mathbf{B}[\mathbf{x} := \mathbf{x}'])$ fails. But all unifications in the computation of $(\theta : \mathbf{B}[\mathbf{x} := \mathbf{t}])$ will behave identically to the corresponding ones in the computation of $(\theta[\mathbf{x}' := \mathbf{t}] : \mathbf{B}[\mathbf{x} := \mathbf{x}'])$; so $(\theta : \mathbf{B}[\mathbf{x} := \mathbf{t}])$ must fail. Contradiction; so we have that $(\theta : \exists \mathbf{x}\ \mathbf{B})$ succeeds.

$S(\mathbf{B}\&\mathbf{C})$: By the induction hypothesis, $S(\mathbf{B})$ and $S(\mathbf{C})$ work wrt θ. So the backtrack stack $(\theta : \mathbf{B})$ succeeds; that is, there is a derivation

$$(\theta : \mathbf{B}) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta' : \epsilon); \beta_1$$

But then there is a derivation

$$(\theta : \mathbf{B}, \mathbf{C}) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta' : \mathbf{C}); \beta_2$$

since the additional formula does not affect the rule applications. But since θ was a closed-term substitution, none of the additional information in θ' is relevant to the computation of $\mathbf{C}$; so since (by the induction hypothesis) $\mathbf{C}$ succeeds wrt θ, we have

$$(\theta' : \mathbf{C}); \beta_2 \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta'' :); \beta_3$$

Thus $(\theta : \mathbf{B}\&\mathbf{C})$ succeeds.

$S(\mathbf{B} \vee \mathbf{C}), 1$: By the induction hypothesis, $S(\mathbf{B})$ works wrt θ, so we have that

$$(\theta : \mathbf{B} \vee \mathbf{C}) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}); (\theta : \mathbf{C}) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta' : \epsilon); \beta; (\theta : \mathbf{C})$$

so $S(\mathbf{B} \vee \mathbf{C})$ works wrt θ.

$S(\mathbf{B} \vee \mathbf{C})$, 2: Both $F(\mathbf{B})$ (by the previous lemma) and $S(\mathbf{C})$ (by the induction hypothesis) work wrt θ; so we have that

$$(\theta : \mathbf{B} \vee \mathbf{C}) \stackrel{\text{SP}}{\Rightarrow} (\theta : \mathbf{B}); (\theta : \mathbf{C}) \stackrel{\text{SP}^*}{\Rightarrow} (\theta : \mathbf{C}) \stackrel{\text{SP}^*}{\Rightarrow} (\theta' : \epsilon); \beta$$

so again $S(\mathbf{B} \vee \mathbf{C})$ works.

Unfolding rules: By theorem 5.3 and theorem 5.4, if $\mathbf{A}'$ is some predicate unfolding or disjunctive unfolding of $\mathbf{A}$, then we can substitute any occurrence of $\mathbf{A}$ for $\mathbf{A}'$, and vice versa, in any backtrack stack, and get the same result. Thus $(\theta : \mathbf{A})$ succeeds iff $(\theta : \mathbf{A}')$ succeeds.

Infinitary rule: Assume the converse; that is, that all the premisses are valid but the conclusion is invalid. This means that there is some closed-term substitution of the form $\theta[\mathbf{x} := \mathbf{t}]$, with respect to which $S(\mathbf{A})$ does not work. But one of the premisses is of the form $S(\mathbf{A}[\mathbf{x} := \mathbf{t}])$, and this must work with respect to θ; contradiction.

□

Theorem 6.4 (Soundness of SPndi) If $\sigma(\mathbf{A})$ is derivable in SPndi, then it is valid.

Proof. By the previous two lemmata.

□

6.3 Completeness and Characterisation

The completeness result is the converse of soundness, and is best done in four stages: a lemma about the completeness of F derivations for failing computations with no Defined Predicate steps, the same lemma about all failing computations, and a lemma about the completeness of S derivations, all for *closed* formulae; and finally the main theorem about all formulae.

Lemma 6.5 If $\mathbf{A}$ is a closed formula, and for all θ, $(\theta : \mathbf{A})$ fails without performing any Defined Predicate steps, then $F(\mathbf{A})$ has a derivation having no predicate unfolding steps.

Proof. It suffices to prove the case where θ is the empty substitution (), because since $\mathbf{A}$ is closed, the substitution cannot affect the success or failure of the formula. The proof is by induction on the length of the SP-computation. The cases are on the form of $\mathbf{A}$.

$F(\mathbf{s} = \mathbf{t})$: If $\mathbf{s}$ and $\mathbf{t}$ are closed and do not unify, it is because they are non-identical terms.

$F(\exists \mathbf{x}\ \mathbf{B})$: The first step in the failing derivation is $((): \exists \mathbf{x}\ \mathbf{B}) \stackrel{\text{SP}}{\Rightarrow} ((): \mathbf{B}[\mathbf{x} := \mathbf{x}'])$. By Theorem 4.3, we have that $((): \mathbf{B}[\mathbf{x} := \mathbf{t}])$ fails as well, for any $\mathbf{t}$, taking a smaller number of steps than the original computation and (by an easy generalisation) not making any Defined Predicate steps either. By the induction hypothesis, then, we have an SPndi-derivation which ends with

$$\frac{\dfrac{F(\mathbf{B}[\mathbf{x} := \mathbf{t}_1]) \quad F(\mathbf{B}[\mathbf{x} := \mathbf{t}_2]) \quad \ldots}{F(\mathbf{B}[\mathbf{x} := \mathbf{x}'])}}{F(\exists \mathbf{x}.\mathbf{B})}$$

and the result holds.

$F(\mathbf{B\&C})$: The first step is $(() : \mathbf{B\&C}) \stackrel{\mathrm{SP}}{\Rightarrow} (() : \mathbf{B}, \mathbf{C})$. If $\mathbf{B}$ fails, then by the induction hypothesis we can use rule $F(\&), 1$ to derive $F(\mathbf{B\&C})$. Otherwise, $\mathbf{B}$ must succeed and $\mathbf{C}$ must fail. There are two subcases.

- $\mathbf{B}$ contains no predicate calls. In this case, we can use rule $F(\&), 2$ straightforwardly.
- $\mathbf{B}$ contains predicate calls, and therefore has a key subformula; therefore $\mathbf{A} \equiv \mathbf{B\&C}$ also has a key subformula. First note that the key subformula of $\mathbf{A}$ must eventually get to the top of the topmost goal stack in the computation, because it has only & and $\exists$ connectives enclosing it. The key subformula of $\mathbf{A}$ therefore cannot be a predicate call – it must be a disjunction $\mathbf{B}_1 \vee \mathbf{B}_2$. (The predicate call in $\mathbf{B}$ is in $\mathbf{B}_1$ or $\mathbf{B}_2$, or to the right of them.)

  The computation of $\mathbf{A}$ is therefore of the form

  $$(() : \mathbf{A}) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta : \mathbf{B}_1 \vee \mathbf{B}_2, \alpha) \stackrel{\mathrm{SP}}{\Rightarrow} (\theta : \mathbf{B}_1, \alpha); (\theta : \mathbf{B}_2, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

  Now, the disjunctive unfolding of $\mathbf{A}$ is $(\mathbf{A}^{\mathbf{B}_1} \vee \mathbf{A}^{\mathbf{B}_2})$. By theorem 5.4, it has an SP-computation with the same result; this computation must take the form

  $$(() : \mathbf{A}^{\mathbf{B}_1} \vee \mathbf{A}^{\mathbf{B}_2}) \stackrel{\mathrm{SP}}{\Rightarrow} (() : \mathbf{A}^{\mathbf{B}_1}); (() : \mathbf{A}^{\mathbf{B}_2}) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta : \mathbf{B}_1, \alpha); (() : \mathbf{A}^{\mathbf{B}_2}) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

  $(\theta : \mathbf{B}_1, \alpha)$ will fail, as in the computation of $\mathbf{A}$, after which the derivation will go

  $$(() : \mathbf{A}^{\mathbf{B}_2}) \stackrel{\mathrm{SP}^*}{\Rightarrow} (\theta : \mathbf{B}_2, \alpha) \stackrel{\mathrm{SP}^*}{\Rightarrow} \epsilon$$

  and $(\theta : \mathbf{B}_2, \alpha)$ fails too. So both $\mathbf{A}^{\mathbf{B}_1}$ and $\mathbf{A}^{\mathbf{B}_2}$ fail with a shorter computation than $\mathbf{A}$ (the $\vee$ step and all the steps involving the failure of the other branch are missing). By the unfolding rule and the rules for $\vee$, we have an SPndi-derivation which ends as follows:

  $$\dfrac{\dfrac{F(\mathbf{A}^{\mathbf{B}_1}) \quad F(\mathbf{A}^{\mathbf{B}_2})}{F(\mathbf{A}^{\mathbf{B}_1} \vee \mathbf{A}^{\mathbf{B}_2})}}{F(\mathbf{A})}$$

  and the result holds.

$F(\mathbf{B} \vee \mathbf{C})$: The first step in the computation is $(() : \mathbf{B} \vee \mathbf{C}) \stackrel{\mathrm{SP}}{\Rightarrow} (() : \mathbf{B}); (() : \mathbf{C})$. Each of these closures fails in turn, so by rule $F(\vee)$, we have an SPndi-derivation for $F(\mathbf{B} \vee \mathbf{C})$.
□

Lemma 6.6 (Completeness of Failing Closed Formulae) If $\mathbf{A}$ is a closed formula, and for all θ, $(\theta : \mathbf{A})$ fails, then $F(\mathbf{A})$ has an SPndi-derivation.

Proof. Again, it suffices to consider only the case where θ is the empty substitution. $\mathbf{A}$ must fail after having performed a finite number of Defined Predicate steps. There is therefore some predicate unfolding of $\mathbf{A}$ (the one in which all occurrences of predicates which will get to the top of the stacks are unfolded) which fails without performing any Defined Predicate steps at all. By the previous theorem and the predicate unfolding rule, $F(\mathbf{A})$ has an SPndi-derivation.
□

Lemma 6.7 (Completeness of Successful Closed Formulae) If $\mathbf{A}$ is a closed formula, and for all θ, $(\theta : \mathbf{A})$ succeeds, then $S(\mathbf{A})$ has an SPndi-derivation.

Proof. Again, it suffices to consider only the case where θ is the empty substitution. The proof is by induction on the length of the computation, the cases on the first step of the computation.

$S(\mathbf{s} = \mathbf{t})$: If $\mathbf{s} = \mathbf{t}$ succeeds, then since $\mathbf{s}$ and $\mathbf{t}$ are closed, they must be identical.

$S(\mathbf{B}\&\mathbf{C})$: we have the computation

$$((): \mathbf{B}\&\mathbf{C}) \stackrel{\text{SP}}{\Rightarrow} ((): \mathbf{B}, \mathbf{C}) \stackrel{\text{SP}^*}{\Rightarrow} (\theta : \mathbf{C}); \beta \stackrel{\text{SP}^*}{\Rightarrow} (\theta' : \epsilon); \beta'$$

But if we take $\mathbf{C}$ off all goal stacks from the second to the third steps above, we get a successful (shorter) computation of $\mathbf{B}$; and since the computation from the third to the fourth step must involve a successful (shorter) computation of $\mathbf{C}$, then $\mathbf{C}$ succeeds wrt the empty environment too (because it is closed and the environment cannot affect its success or failure). Thus, by the induction hypothesis and the SPndi-rule $S(\&)$, the result holds.

$S(\mathbf{B} \vee \mathbf{C})$: we have the computation

$$((): \mathbf{B} \vee \mathbf{C}) \stackrel{\text{SP}}{\Rightarrow} ((): \mathbf{B}); ((): \mathbf{C}) \stackrel{\text{SP}}{\Rightarrow} (\theta : \epsilon); \beta$$

So either $\mathbf{B}$ must succeed (with a computation one step shorter), or else $\mathbf{B}$ must fail (with a shorter computation) and $\mathbf{C}$ must succeed (with a shorter computation). By rule $S(\vee), 1$ or $S(\vee), 2$ and by the previous lemma about the completeness of failing computations, the result holds.

$S(\exists \mathbf{x}\ \mathbf{B})$: the first step in the computation is

$$((): \exists \mathbf{x}\ \mathbf{B}) \stackrel{\text{SP}}{\Rightarrow} ((): \mathbf{B}[\mathbf{x} := \mathbf{x}'])$$

There are two subcases.

- $\mathbf{B}$ contains no predicate calls. By theorem 4.5, there must be a $\mathbf{t}$ such that $\mathbf{B}[\mathbf{x} := \mathbf{t}]$ succeeds wrt the empty environment. This successful computation is the same length as for $\mathbf{B}[\mathbf{x} := \mathbf{x}']$, namely one step shorter than the computation of $\exists \mathbf{x}\ \mathbf{B}$. Thus by rule $S(\exists)$, the result holds.

- $\mathbf{B}$ contains predicate calls. In this case we can follow the same reasoning as in the case for $F(\&), 2$: $\mathbf{B}$'s key subformula cannot be a predicate call, so it must be a disjunction; we can therefore apply the disjunctive unfolding rule and one of the $S(\vee)$ rules, and by the induction hypothesis get the rest of the derivation.

$S(\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n))$: the result follows immediately from the induction hypothesis, using the predicate unfolding rule.

□

Corollary 6.8 (Completeness) If $\sigma(\mathbf{A})$ is valid, then $\sigma(\mathbf{A})$ is derivable in SPndi.

Proof. By induction on the number n of free variables in **A**. The case $n = 0$ is the previous theorem. The case $n > 0$ can be handled by eliminating one variable with the infinitary rule.
□

We end this section with the main comprehensive result concerning SPndi: that it is an exact (sound and complete) characterisation of terminating sequential Prolog computations.

Corollary 6.9 (Characterisation)
(1) For every closed formula **A**:
The query **A** succeeds iff $S(\mathbf{A})$ is derivable in SPndi.
The query **A** fails iff $F(\mathbf{A})$ is derivable in SPndi.
(2) For every formula **A**:
The query **A** succeeds iff $S(\exists[\mathbf{A}])$ is derivable in SPndi.
The query **A** fails iff $F(\exists[\mathbf{A}])$ is derivable in SPndi.

Proof. A straightforward consequence of the Soundness and Completeness theorems, and the operational rules for the existential quantifier.

To summarise, the proof system SPndi is a sound, complete, and logical characterisation of sequential Prolog. It would be nice if it could be finitary as well, but this is not possible due to the nature of the problem. The solution we have followed to the problem of incompleteness is a standard one from the literature.

7 A Weakly Complete, Finitary Proof System

In this section, we study a finitary (and thus more practical) proof system which is sound and has a limited form of completeness, but is not complete in the fuller sense of SPndi.

This system is called SPndf (Sequential Prolog, natural deduction, finitary), and the form of completeness that it has is the following: any valid signed formula *containing no calls to recursive predicates* is derivable in SPndf. This result is not very useful practically, but it does show that the incompleteness of SPndf arises exclusively from the possibility of recursion in predicate calls, and not from inadequacies in how it describes the connectives.

Many valid signed formulae containing recursive predicate calls are also derivable in SPndf, but there are important ones which are not. I will discuss this problem, and suggest ways of handling it by adding rules for induction to the system.

7.1 The Proof System SPndf

The system SPndf can be regarded as being derived from SPndi by first using a more general form of judgments, then removing the infinitary rule, and then adding some rules about equality to ensure weak completeness.

The more general form of judgment that SPndf uses is a set of signed formulae, interpreted disjunctively. Because a set of signed formulae interpreted in this way behaves like the right-hand half of a sequent, we call these objects "semi-sequents".

Definition 7.1 A *semi-sequent* is a set of signed formulae.

The disjunctive interpretation is reflected in the validity condition for semi-sequents.

Definition 7.2 A semi-sequent $\sigma_1(\mathbf{A}_1) \ldots \sigma_n(\mathbf{A}_n)$ is *valid* if, for every substitution θ mapping free variables of the semi-sequent onto closed terms, there is some $i, 1 \leq i \leq n$, such that $\sigma_i(\mathbf{A}_i)$ works wrt θ.

Note that a semi-sequent containing only one signed formula is valid iff that formula is valid according to the previous definition of validity. This definition of validity is therefore just an extension of the previous definition.

If we used semi-sequents in SPndi instead of sequents, we could use the same rules, just augmenting every occurrence of a signed formulae $\sigma(\mathbf{A})$ by a set of signed formulae Γ. We would then be able to write some of the rules in a different way. Consider the rules for success of a disjunction, augmented by the rest of the semi-sequent:

$$\frac{\Gamma, S(\mathbf{B})}{\Gamma, S(\mathbf{B} \vee \mathbf{C})} \qquad \frac{\Gamma, F(\mathbf{B}) \quad \Gamma, S(\mathbf{C})}{\Gamma, S(\mathbf{B} \vee \mathbf{C})}$$

We could use the implicit disjunction in semi-sequents to write these two rules as one rule:

$$\frac{\Gamma, S(\mathbf{B}), F(\mathbf{B}) \quad \Gamma, S(\mathbf{B}), S(\mathbf{C})}{\Gamma, S(\mathbf{B} \vee \mathbf{C})}$$

This "amalgamated" rule gives us another way of explaining when a disjunction succeeds: it succeeds when $\mathbf{B}$ *terminates* (with either success or failure), and either $\mathbf{B}$ or $\mathbf{C}$ succeeds.

SPndf uses semi-sequents and the amalgamated versions of the $S(\vee)$ and $F(\&)$ rules. As will become clear, this structure is necessary so that we can give rules for equality which compensate for the loss of the infinitary rule.

The rules for SPndf are in Figure 5. In all rules, the order of signed formulae within the semi-sequent is unimportant. The combined effect of the semi-sequent judgments, the amalgamated rules, and the equality rules allow us to prove things with finitary derivations that we could do in SPndi only with infinitary derivations. For instance, consider the signed sentence $F(\exists x(x = 0 \& x = 1))$. This is clearly valid. Its derivation in SPndi would look like this:

$$\frac{\dfrac{\dfrac{}{F(0=1)}}{F(0=0\&0=1)} \quad \dfrac{\dfrac{}{F(1=0)}}{F(1=0\&1=1)} \quad \dfrac{\dfrac{}{F(2=0)}}{F(2=0\&2=1)} \quad \ldots}{\dfrac{F(x=0\&x=1)}{F(\exists x(x=0\&x=1))}}$$

The corresponding derivation in SPndf would look like this:

$$\frac{\dfrac{\dfrac{}{S(0=0), F(x=0)}}{S(x=0), F(x=0)} \quad \dfrac{\dfrac{}{F(x=0), F(0=1)}}{F(x=0), F(x=1)}}{\dfrac{F(x=0\&x=1)}{F(\exists x(x=0\&x=1))}}$$

Note how signed sentences of the form $F(\mathbf{s} = \mathbf{t})$ behave as assumptions that $\mathbf{s}$ and $\mathbf{t}$ are equal. This behaviour can be explained by noting that, if we interpret the semi-sequent "comma" as "or" and failure as negation, we can view $F(x = 0), S(x = 0)$ as meaning $\neg(x = 0) \vee x = 0$; that is, $x = 0 \rightarrow x = 0$.

1. Equality.

Eq: $\dfrac{}{\Gamma, S(\mathbf{t} = \mathbf{t})}$ Occurs: $\dfrac{}{\Gamma, F(\mathbf{s} = \mathbf{t})}$ $(*a)$

Ineq: $\dfrac{}{\Gamma, F(\mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) = \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_m))}$ $(*b)$

Comp: $\dfrac{\Gamma, F(\mathbf{s}_1 = \mathbf{t}_1), \ldots, F(\mathbf{s}_n = \mathbf{t}_n)}{\Gamma, F(\mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) = \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n))}$

Subst: $\dfrac{\Gamma, F(\mathbf{s} = \mathbf{t}), \sigma(\mathbf{A}(\mathbf{s}))}{\Gamma, F(\mathbf{s} = \mathbf{t}), \sigma(\mathbf{A}(\mathbf{t}))}$

2. Structural.

Thin: $\dfrac{\Gamma}{\Gamma, \sigma(\mathbf{A})}$ Cut: $\dfrac{\Gamma, S(\mathbf{A}) \quad \Gamma, F(\mathbf{A})}{\Gamma}$

3. Connectives.

$S(\&)$: $\dfrac{\Gamma, S(\mathbf{B}) \quad \Gamma, S(\mathbf{C})}{\Gamma, S(\mathbf{B}\&\mathbf{C})}$ $F(\&)$: $\dfrac{\Gamma, S(\mathbf{B}), F(\mathbf{B}) \quad \Gamma, F(\mathbf{B}), F(\mathbf{C})}{\Gamma, F(\mathbf{B}\&\mathbf{C})}$ $(*c)$

$S(\vee)$: $\dfrac{\Gamma, S(\mathbf{B}), F(\mathbf{B}) \quad \Gamma, S(\mathbf{B}), S(\mathbf{C})}{\Gamma, S(\mathbf{B} \vee \mathbf{C})}$ $F(\vee)$: $\dfrac{\Gamma, F(\mathbf{B}) \quad \Gamma, F(\mathbf{C})}{\Gamma, F(\mathbf{B} \vee \mathbf{C})}$

$S(\exists)$: $\dfrac{\Gamma, S(\mathbf{B}[\mathbf{x} := \mathbf{t}])}{\Gamma, S(\exists \mathbf{x}\mathbf{B})}$ $(*c)$ $F(\exists)$: $\dfrac{\Gamma, F(\mathbf{B}[\mathbf{x} := \mathbf{y}])}{\Gamma, F(\exists \mathbf{x}\mathbf{B})}$ $(*d)$

4. Unfoldings.

Unfold: $\dfrac{\Gamma, \sigma(\mathbf{A}')}{\Gamma, \sigma(\mathbf{A})}$

where $\mathbf{A}'$ is a predicate unfolding or disjunctive unfolding of $\mathbf{A}$

Side-conditions:
$(*a)$: $\mathbf{s}$ is a proper subterm of $\mathbf{t}$ or vice versa
$(*b)$: $\mathbf{f} \not\equiv \mathbf{g}$
$(*c)$: $\mathbf{B}$ contains no predicate calls
$(*d)$: the proof of the upper semi-sequent does not contain any predicate unfolding steps, and $\mathbf{y}$ does not appear in the lower semi-sequent

Figure 5: The finitary, weakly complete proof system SPndf.

The two "structural" rules which appear in SPndf are the thinning and cut rules. These were not necessary in SPndi because the judgments were single signed formulae. As we will see later, they are not important for the soundness and completeness proofs; however, their use allows us to make derivations clearer. The form of the cut rule may seem a little mysterious, but it is in fact just the analogue of the usual cut rule from the intuitionistic sequent calculus:

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

7.2 Soundness

SPndf is sound according to the new definition of validity.

Theorem 7.3 (Soundness of SPndf) Every derivable semi-sequent $\sigma_1(\mathbf{A}_1) \ldots \sigma_n(\mathbf{A}_n)$ is valid.

Proof. Except for the case of the Cut rule, it suffices to prove the following. Consider an application of a given rule. Let θ be a closed-term substitution for all the free variables of all the semi-sequents in the rule application. Assume further that none of the formulae in Γ work wrt θ, and that some other formula in each premiss works wrt θ. Then the indicated formula in the conclusion works wrt θ.

Equality, Inequality, Occurs check, Decomposition: recalling that θ substitutes closed terms for all free variables in all terms, and that closed terms unify only if they are identical, these rules are straightforward.

Substitution: Assume that $F(\mathbf{s} = \mathbf{t})$ does not work; that is, that $\mathbf{s}\theta$ is identical to $\mathbf{t}\theta$. Then the effect of replacing $\mathbf{s}$ by $\mathbf{t}$ will only be to cause $\mathbf{s}$ to be replaced by $\mathbf{t}$ in subsequent unifications in the processing of $\mathbf{A}$. But all such unifications will take place after a substitution of the form $\theta\theta'$; and by assumption, $\mathbf{s}$ and $\mathbf{t}$ are identical under θ.

Thinning: trivial.

Cut: $\mathbf{A}$ cannot both succeed and fail, so the only way both of the premisses can be valid is for one of the Γ formulae to work wrt θ.

$S(\&)$, $S(\vee)$, $S(\exists)$, $F(\&)$, $F(\vee)$, $F(\exists)$, Unfoldings rule: very similar to the corresponding cases for SPndi.

□

7.3 Weak Completeness of SPndf

SPndf as it stands is not complete: there are semi-sequents such as $S(add(x, 0, x))$ which are valid but not derivable because every instance of them has different derivation.

However, it turns out that under the (not very restrictive) condition that the underlying language $\mathcal{L}$ has an infinite number of closed terms, every valid semi-sequent *which does not contain calls to recursive predicates* is indeed derivable.

We begin with a technical lemma that we will need in one case in the proof.

Lemma 7.4 (Commutativity of Equality in SPndf) The semi-sequent $(\Gamma, \sigma(\mathbf{s} = \mathbf{t}))$ is derivable in SPndf iff the semi-sequent $(\Gamma, \sigma(\mathbf{t} = \mathbf{s}))$ is derivable.

Proof. The F case follows from the following derivation fragment, using the Thinning and Cut rules and the Equality axiom.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, F(\mathbf{s}=\mathbf{t})}}{\Gamma, F(\mathbf{s}=\mathbf{t}), F(\mathbf{t}=\mathbf{s})} \quad \dfrac{\dfrac{\vdots}{\Gamma, S(\mathbf{t}=\mathbf{t}), F(\mathbf{t}=\mathbf{s})}}{\Gamma, S(\mathbf{s}=\mathbf{t}), F(\mathbf{t}=\mathbf{s})}}{\Gamma, F(\mathbf{t}=\mathbf{s})}$$

There is a similar derivation for the S case.
□

Now we move on to the completeness theorem, starting with a slightly weaker theorem which states that SPndf is complete for queries with no predicate calls at all.

Theorem 7.5 (Weak Completeness of SPndf) If the language $\mathcal{L}$ contains an infinite number of closed terms, then every valid semi-sequent $\sigma_1(\mathbf{A}_1) \ldots \sigma_n(\mathbf{A}_n)$, in which the $\mathbf{A}_i$'s contain no applications of predicates other than $=$, is derivable.

Proof. We begin by assuming the opposite, and proving a contradiction. That is, we assume that there is a valid, underivable semi-sequent $\mathcal{S}$; we prove a contradiction by proving, in stages, that the existence of $\mathcal{S}$ implies the existence of such a semi-sequent in a smaller and smaller class of semi-sequents – and then proving that there is no such semi-sequent in the smallest class.

1. We define the following measure on semi-sequents.

$$m_1(\sigma_1(\mathbf{A}_1), \ldots, \sigma_n(\mathbf{A}_n)) = \sum_{i=1}^{n} \omega^{d(\mathbf{A}_i)}$$

   where $d^{\mathbf{A}_i}$ is the complexity (depth in connectives), of $\mathbf{A}_i$, with equality formulae counting as having 0 complexity. Note that we get a semi-sequent with lower m_1 if we replace a signed formula by an arbitrary number of signed formulae of lower complexity.

2. If $\mathcal{S}$ is underivable and valid, and contains formulae with connectives, then there is an underivable, valid $\mathcal{S}'$ such that $m_1(\mathcal{S}') < m_1(\mathcal{S})$. We prove this by selecting a signed formula $\sigma(\mathbf{A})$ in $\mathcal{S}$ with a connective, and forming $\mathcal{S}'$ from it by running the connective rules of SPndf backwards:

   - Case $\mathcal{S} \equiv \Gamma, S(\mathbf{B}\&\mathbf{C})$: either $\Gamma, S(\mathbf{B})$ or $\Gamma, S(\mathbf{C})$ must be underivable (otherwise $\mathcal{S}$ would be derivable). Let $\mathcal{S}'$ be the one which is underivable. If $\mathcal{S}$ is valid, then $\mathcal{S}'$ must be valid: for every θ such that $(\mathbf{B}\&\mathbf{C})\theta$ is closed, either one of the Γ signed formulae works wrt θ, or else $S(\mathbf{B}\&\mathbf{C})$ works wrt θ, in which case both $\mathbf{B}$ and $\mathbf{C}$ must succeed wrt θ (see the completeness proof for SPndi). $m_1(\mathcal{S}') < m_1(\mathcal{S})$, because $\mathbf{B}$ and $\mathbf{C}$ have smaller depth than $(\mathbf{B}\&\mathbf{C})$.
   - Other cases are similar, and involve similar reasoning to the completeness proof of SPndi. In all cases, $\mathcal{S}'$ has one fewer formula of some depth n, and one or two formulae of depth less than n; so $m_1(\mathcal{S}') < m_1(\mathcal{S})$.

3. Thus, by induction, if $\mathcal{S}$ is underivable, valid, and has connectives, then there is a $\mathcal{S}'$ which is underivable, valid, and has no connectives ($m_1(\mathcal{S}') < \omega$). So we can now assume that $\mathcal{S}$ has only signed formulae of the form $S(\mathbf{s} = \mathbf{t})$ or $F(\mathbf{s} = \mathbf{t})$.

4. We define another measure on semi-sequents, as follows.

$$m_2(\sigma_1(\mathbf{s}_1 = \mathbf{t}_1), \ldots, \sigma_n(\mathbf{s}_n = \mathbf{t}_n)) = k\omega + d$$

where k is the number of free variables in the sequent, and d is the number of function symbols and *occurrences* of free variables in all the $\mathbf{s}_i$'s and $\mathbf{t}_i$'s. Note that we get a semi-sequent with lower m_2 if we eliminate one free variable – or if we replace a formula by an arbitrary number of formulae in which the total number of function symbols and variables is lower but the number of free variables is the same.

5. Let $\mathcal{S}$ be underivable and valid, and of the form

$$F(\mathbf{s}_1 = \mathbf{t}_1), \ldots, F(\mathbf{s}_j = \mathbf{t}_j), S(\mathbf{s}_{j+1} = \mathbf{t}_{j+1}), \ldots, S(\mathbf{s}_n = \mathbf{t}_n)$$

where $j \geq 1$. Then there is an underivable, valid $\mathcal{S}'$ such that $m_2(\mathcal{S}') < m_2(\mathcal{S})$. We prove this by selecting a signed formula $F(\mathbf{s} = \mathbf{t})$ in $\mathcal{S}$, and forming $\mathcal{S}'$ by eliminating it. There are three subcases:

- The signed formula is of the form $F(\mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n)) = \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n))$. In this case we form $\mathcal{S}'$ by replacing that signed formula by the n signed formulae $F(\mathbf{s}_i = \mathbf{t}_i)$. If $\mathcal{S}'$ is derivable, then so is $\mathcal{S}$ by the Decomposition rule; if $\mathcal{S}$ is valid, then $\mathcal{S}'$ is valid due to the nature of unification; and $m_2(\mathcal{S}') < m_2(\mathcal{S})$.
- The signed formula is of the form $F(\mathbf{x} = \mathbf{x})$. In this case, we form $\mathcal{S}'$ from $\mathcal{S}$ by eliminating this signed formula. The derivability and validity conditions hold ($\mathbf{F}(\mathbf{x} = \mathbf{x})$ cannot work wrt any substitution), and $m_2(\mathcal{S}') < m_2(\mathcal{S})$.
- The signed formula is of the form $F(\mathbf{x} = \mathbf{t})$ or $F(\mathbf{t} = \mathbf{x})$, where $\mathbf{t}$ does not contain $\mathbf{x}$ at all. (It doesn't matter which way the equality goes, due to the commutativity of equality proved above). In this case, assume Γ is the rest of the signed formulae in $\mathcal{S}$. Let $\mathcal{S}'$ be $\Gamma[\mathbf{x} := \mathbf{t}]$. If $\mathcal{S}'$ is derivable, then so is $\mathcal{S}$, by one application of Thinning and a finite number of applications of Substitution.

  If $\mathcal{S}$ is valid, then one of the signed formulae in it must work for all substitutions of the form $[\mathbf{x} := \mathbf{t}]\rho$ which map all free variables to closed terms. But the one that works in these cases cannot be the selected signed formula, because under such a substitution it would be $F(\mathbf{t}\rho = \mathbf{t}\rho)$; so it must be one of the others in Γ. But since this is independent of ρ, one of the signed formulae in $Sseq'$ works wrt any ρ; so $\mathcal{S}'$ is valid. Since $\mathcal{S}'$ has one fewer free variable, $m_2(\mathcal{S}') < m_2(\mathcal{S})$.

Note that the selected formula cannot be of the form $F(\mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) = \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_m))$, or of the form $F(\mathbf{s} = \mathbf{t})$ where $\mathbf{s}$ is a proper subterm of $\mathbf{t}$ or vice versa; otherwise $\mathcal{S}$ would be derivable.

6. Thus, by induction, if $\mathcal{S}$ is underivable, valid, and contains no connectives but does contain signed formulae of the form $F(\mathbf{s} = \mathbf{t})$, then there is a $\mathcal{S}'$ consisting only of

formulae of the form $S(\mathbf{s} = \mathbf{t})$ which is also underivable and valid. ($m_2(\mathcal{S}')$ is some multiple of ω.) So we can now assume that $\mathcal{S}$ consists only of signed formulae of the form $S(\mathbf{s} = \mathbf{t})$.

7. If $\mathcal{S}$ is underivable, valid, contains free variables, and is of the form

$$S(\mathbf{s}_1 = \mathbf{t}_1), \ldots, S(\mathbf{s}_n = \mathbf{t}_n)$$

then there is an $\mathcal{S}'$ with one fewer free variable which is underivable, valid, and of this form. Choose one free variable $\mathbf{x}$ in $\mathcal{S}$. $\mathbf{x}$ is being compared with only a finite number of closed terms in these equations; choose some other closed term $\mathbf{t}$ out of the infinite number of closed terms. Then construct $\mathcal{S}'$ by replacing $\mathbf{x}$ everywhere by $\mathbf{t}$. This operation will not cause any of the signed formulae to become of the form $S(\mathbf{s} = \mathbf{s})$, by our choice of $\mathbf{t}$; so if $\mathcal{S}$ is underivable then $\mathcal{S}'$ is underivable. But if $\mathcal{S}$ is valid, then one of its signed formulae must work wrt any substitution; so $\mathcal{S}'$ must be valid.

8. So, by induction, if there is *any* underivable and valid semi-sequent, then there is an underivable and valid semi-sequent which contains only signed formulae of the form $S(\mathbf{s} = \mathbf{t})$, where $\mathbf{s}$ and $\mathbf{t}$ have no free variables. But none of these can be of the form $S(\mathbf{s} = \mathbf{s})$, otherwise the semi-sequent would be derivable; so, since the $\mathbf{s}$'s and $\mathbf{t}$'s are always closed and non-identical, none of them unify, so none of the signed formulae work; that is, the semi-sequent is not valid after all. This is the final contradiction which leads us to the conclusion that there is no underivable, valid semi-sequent.

□

There are a few points to note about this proof. One is that it does not involve the Thinning and Unfoldings rules to any essential degree, and that if we were to add the rule

$$\frac{\Gamma, F(\mathbf{t} = \mathbf{s}), \sigma(\mathbf{A}(\mathbf{s}))}{\Gamma, F(\mathbf{t} = \mathbf{s}), \sigma(\mathbf{A}(\mathbf{t}))}$$

we would not need the Cut rule either (which is only needed to prove the commutativity of equality). So even a much weaker system would be complete in this *weak* sense.

Another point is that if our language did not have an infinite number of terms, the proof would not go through. Consider the language consisting only of the constants a and b, and the semi-sequent

$$S(x = a), S(x = b)$$

This semi-sequent is clearly underivable and valid. Fortunately, however, this case is exactly the case in which the "infinitary" proof system SPndi is no longer infinitary (see section 6.1), so we actually have a finitary proof system with *strong* completeness in this case!

The weak completeness result leads to the stronger result about the completeness of SPndf for queries containing no calls to recursive predicates. We can prove this through defining the class of programs that have no recursion or mutual recursion.

Definition 7.6 A *strictly stratified* program is one in which each predicate $\mathbf{P}$ can be assigned a natural number $n_{\mathbf{P}} > 0$ such that if $\mathbf{P}$ calls $\mathbf{Q}$, $n_{\mathbf{Q}} < n_{\mathbf{P}}$.

Corollary 7.7 If Π is a strictly stratified program in a language with an infinite number of closed terms, then every semi-sequent which is valid with respect to SP_Π is derivable in $SPndf_\Pi$.

Proof. Every predicate call in such a semi-sequent can be expanded via the unfoldings rule. After a finite number of such expansions, the semi-sequent will contain no predicate calls, at which point the previous theorem applies.
□

This slightly stronger result suggests that the incompleteness of SPndf can be viewed as arising exclusively from a failure to handle recursive predicate calls. In other words, it is the ability of sequential Prolog to compute general recursive functions which makes it powerful enough to have no complete proof system.

Finally, we have a characterisation result which corresponds to the one of the last section.

Corollary 7.8 A query formula **A**, containing no applications of predicates other than $=$, succeeds iff $S(\exists[\mathbf{A}])$ is a semi-sequent derivable in SPndf; **A** fails iff $F(\exists[\mathbf{A}])$ is a semi-sequent derivable in SPndf.

Proof. A consequence of the Soundness and Weak Completeness theorems.

7.4 Discussion

SPndf represents another kind of solution to the problem of the formal undecidability of signed formulae to that represented by SPndi. Instead of a sound, complete, but infinitary proof system, it is a sound, finitary, but incomplete proof system.

The kind of valid semi-sequents which are not derivable in SPndf are those which require inductive proofs to prove valid. For instance, say our language had only 0 and s as function symbols, and our program contained the standard definition for addition of integers represented in Peano arithmetic form:

$$Add(x, y, z) \leftrightarrow (x = 0 \& y = z) \vee \exists x' \exists z' (x = s(x') \& z = s(z') \& Add(x', y, z')$$

Then the semi-sequent $S(Add(x, 0, x))$ would be valid, but we could not prove it in SPndf because each instance would have a different proof structure. We can alleviate this problem to a large extent by deriving a full sequent calculus from SPndf, and then adding induction rules to it. An earlier paper of mine [And89b] describes this technique in more detail.

The addition of induction and other rules, to either strengthen the system or make it easier to use, should allow SPndf to act as the basis of a usable system for proving or formally developing usable Prolog programs.

8 Incompleteness of SP

In this section, I give a sketch of a proof that there is no sound, complete, finitary proof system for signed formulae. This proof is based on Hofstadter's explanation of Gödel's

proof [Hof79]. For simplicity, the exact version of theorem I will prove will be about slightly different kinds of judgments than signed formulae. I trust that this will convince readers of the truth of the more specific theorem.

Definition 8.1 A *program-formula judgment* in a language $\mathcal{L}$ is an expression of the form $\langle \Pi, \sigma(\mathbf{A}) \rangle$, where Π is (some syntactic representation of) a program in $\mathcal{L}$, and $\sigma(\mathbf{A})$ is a signed formula of $\mathcal{L}$.

A judgment $\langle \Pi, \sigma(\mathbf{A}) \rangle$ is *valid* if $\sigma(\mathbf{A})$ is valid with respect to the program Π (as defined in section 6.2).

Theorem 8.2 (Incompleteness) For some language $\mathcal{L}$, there is no sound, complete, and finitary proof system S with program-formula judgments. That is, for every proof system S with program-formula judgments, there is some such judgment that is valid but not derivable in S.

Proof. Assume that, for any $\mathcal{L}$, there is such a proof system; then prove a contradiction.

Choose $\mathcal{L}$ so that we can represent variables, terms, formulae, signed formulae, finite programs, judgments, and derivations for such a proof system, all as closed terms. As Gödel proved, any language with at least one nullary function symbol and at least one non-nullary function symbol suffices. Applying the assumption, we have a sound and complete finitary proof system S for judgments in this language.

Let us write $\ulcorner \mathbf{X} \urcorner$ for the representation in this language of an expression $\mathbf{X}$.

The readers can convince themselves that we can write a program Π_G containing predicates *Subst* and *Deriv* with the following operational properties:

- The query $Subst(\mathbf{r}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t})$ succeeds iff $\mathbf{r}$ is some $\ulcorner \sigma(\mathbf{A}) \urcorner$, and we can obtain $\mathbf{t}$ from $\mathbf{r}$ by substituting all occurrences of $\ulcorner z_1 \urcorner$ by $\mathbf{s}_1$ and all occurrences of $\ulcorner z_2 \urcorner$ by $\mathbf{s}_2$. Moreover, if $\mathbf{r}, \mathbf{s}_1, \mathbf{s}_2$ are all closed and $\mathbf{t}$ is a variable, then the query does not fail, and the resultant substitution contains a closed term for the variable.

- The query $Deriv(\mathbf{r}, \mathbf{s}, \mathbf{t})$ succeeds iff $\mathbf{s}$ is some $\ulcorner \Pi \urcorner$, $\mathbf{t}$ is some $\ulcorner \sigma(\mathbf{A}) \urcorner$, and $\mathbf{r}$ is the representation of a derivation, in the proof system S, of $\langle \Pi, \sigma(\mathbf{A}) \rangle$. Moreover, if all the arguments are closed terms, then the query terminates (either succeeds or fails).

Now, let U be the signed formula

$$F(\exists y(Subst(z_2, z_1, z_2, y) \& Deriv(x, z_1, y)))$$

Let G be the signed formula $U[z_1 := \ulcorner \Pi_G \urcorner, z_2 := \ulcorner U \urcorner]$; that is, let G be

$$F(\exists y(Subst(\ulcorner U \urcorner, \ulcorner \Pi_G \urcorner, \ulcorner U \urcorner, y) \& Deriv(x, \ulcorner \Pi_G \urcorner, y)))$$

Is $\langle \Pi_G, G \rangle$ derivable in S? If so, then it must be valid, since S is sound. Therefore G is valid with respect to Π; that is, for any closed term $\mathbf{t}$ we substitute in for x in G, we have that the query

$$(\exists y(Subst(\ulcorner U \urcorner, \ulcorner \Pi_G \urcorner, \ulcorner U \urcorner, y) \& Deriv(\mathbf{t}, \ulcorner \Pi_G \urcorner, y)))$$

fails. However, the call to Subst cannot fail, and must in fact produce a substitution mapping y to some closed term s.

But what is this closed term s? It is in fact $\lceil G \rceil$. From this we must conclude that the query $Deriv(\mathrm{t}, \lceil \Pi_G \rceil, \lceil G \rceil))$ fails, for every t. But what this means is that there is no derivation for $\langle \Pi_G, G \rangle$, even though that was what we just assumed. Thus, $\langle \Pi_G, G \rangle$ must not be derivable in S after all.

Since $\langle \Pi_G, G \rangle$ is not derivable, we can follow much the same line of reasoning to show that it is in fact valid: the call to *Subst* must succeed, and the call to *Deriv* must fail, so the whole query fails, for any choice of t. So there is a judgment which is valid but not derivable in S. This contradicts our first assumption; so for this choice of $\mathcal{L}$, there can be no sound, complete, and finitary proof system for program-formula judgments.
□

The kind of proof systems we have been studying in the other sections are essentially parameterised by the program; that is, there is a different proof system for each program. This parameterisation was done only for convenience, however, and we could instead express either SPndi or SPndf as being proof systems with program-formula judgments. It seems clear that the incompleteness proof will still apply to these parameterised proof systems, as long as the parameterisation does not hide any operation that cannot be described using general recursive functions.

9 Summary and Future Work

SPndi and SPndf are two proof systems which characterise the behaviour of sequential Prolog computations in a sound, complete and logical manner. SPndi is infinitary but fully complete, and SPndf is finitary and has weak completeness properties. Both have rules which use the predicate and disjunctive unfolding transformations on formulae, allowing them to come much closer to the strength of sequential Prolog than traditional logical characterisations. Since both are highly compositional, treat the connectives in a logical manner, and avoid manipulating substitutions directly, both are more "logical" in nature than denotational characterisations.

This presentation of a logical characterisation for pure, negationless sequential Prolog brings this programming language firmly into the realm of logical analysis, and should allow us to apply other techniques from mathematical logic to study its properties.

The model theory of these proof systems deserves some study. The equivalence relation induced by the disjunctive unfolding operation would probably be an important element of such a model theory. Following Fitting [Fit85], we might define the notion of upward- and downward-saturated sets of signed sentences (modulo a sequential consequence relation rather than the standard first order consequence relation), and the notion of a "sequential interpretation" from this; a "sequential model" of a program would then be a sequential interpretation in which predicate calls have the same sign as their expanded bodies. But this is just speculation.

It would be very nice to be able to enrich the language under consideration with some form of negation, say negation as failure. One great advantage of Baudinet's and other denotational approaches is that they are able to account for such features. The combination of backtracking and the renowned semantic thorniness of negation as failure

seems to make them very difficult to characterise logically.

However, it would appear that we could add inequality to the language without too much difficulty, as in Colmerauer's Prolog II [CKvC83] or in my earlier paper on logic programming with parallel "or" [And89a]. I would also expect that if we used some moded logic programming system, and restricted negated formulae to contain only "input" variables, we could describe negation in a logical manner.

The addition of modes, ground negation as failure, and features such as if-then-else (in preference to cut) to our language would make it fairly powerful and practical. Combining this with the addition of induction rules to a full sequent calculus version of SPndf would form a good basis for a system for proving or formally developing practical Prolog programs.

10 Acknowledgements

My thanks to Paul Voda, who saw the importance of trying to find logical ways of characterising sequential Prolog long ago, and who suggested to me the course of research which led to this paper; and to my advisors at Edinburgh, Don Sannella and Stuart Anderson, who made many valuable comments. Thanks also to Inge-Maria Bethke, Lars Hallnäs, Gordon Plotkin, and David Walker for helpful suggestions.

References

[AB87] Bijan Arbab and Daniel M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4:309–329, 1987.

[And89a] James H. Andrews. Proof-theoretic characterisations of logic programming. In *Proceedings of the 14th International Symposium on the Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 145–154, Porąbka-Kozubnik, Poland, August-September 1989. Springer.

[And89b] James H. Andrews. Proof-theoretic characterisations of logic programming. Technical Report LFCS-89-77, Laboratory for the Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, May 1989.

[Bau88] Marianne Baudinet. Proving termination properties of Prolog programs: A semantic approach. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, pages 336–347, Edinburgh, Scotland, July 1988.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Theory and Practice of Software Engineering*, volume 352 of *Lecture Notes in Computer Science*, pages 96–110, Barcelona, Spain, 1989. Springer-Verlag.

[CKvC83] Alain Colmerauer, Henry Kanoui, and Michel van Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Information*, 2(4):255–292, 1983.

[Cla78] K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, New York, 1978. Plenum Press.

[dBdV89] A. de Bruin and E. P. de Vink. Continuation semantics for Prolog with cut. In *Theory and Practice of Software Engineering*, volume 351 of *Lecture Notes in Computer Science*, pages 178–192, Barcelona, Spain, 1989. Springer-Verlag.

[Der89] Pierre Deransart. Proofs of declarative properties of logic programs. In *Theory and Practice of Software Engineering*, volume 351 of *Lecture Notes in Computer Science*, pages 207–226, Barcelona, Spain, 1989. Springer-Verlag.

[DM88] Saumya Debray and Prateek Mishra. Denotational and operational semantics of Prolog. *Journal of Logic Programming*, 5:61–91, 1988.

[FGKP85] N. Francez, O. Grumberg, S. Katz, and A. Pnueli. Proving termination of Prolog programs. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 89–105, Berlin, July 1985. Springer-Verlag.

[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. Ed. M. E. Szabo.

[Gir87] Jean-Yves Girard. Towards a geometry of interaction. In *Proceedings of the AMS Conference on Categories, Logic, and Computer Science*, Boulder, Colorado, June 1987. To appear.

[Gol82] Robert Goldblatt. *Axiomatising the Logic of Computer Programming*, volume 130 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1982.

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1979.

[HS84] Masami Hagiya and Takafumi Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2:59–77, 1984.

[HSH88] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming. Technical Report R88005, Swedish Institute of Computer Science, 1988.

[JM84] Neil D. Jones and Alan Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*, February 1984.

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

[NF89] Tim Nicholson and Norman Foo. A denotational semantics for prolog. *ACM Transactions on Programming Languages and Systems*, 11:650–665, October 1989.

[Sun83] Göran Sundholm. Systems of deduction. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, pages 133–188. D. Reidel, Dordrecht, 1983.

[Ull86] Jeffrey Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1986.