LFCS

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

Proof-Theoretic Characterisations of Logic Programming

Proof-Theoretic Characterisations of Logic Programming

by

James H. Andrews

LFCS Report Series

ECS-LFCS-89-77
(also published as CSR-295-89)

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

June 1989

Proof-Theoretic Characterisations of Logic Programming

James H. Andrews

July 20, 1989

Abstract

A characterisation of a logic programming system is given in the form of a natural deduction proof system. The proof system is proven to be "equivalent" to an operational semantics for the logic programming system, in the sense that the set of theorems of the proof system is exactly the set of existential closures of queries solvable in the operational semantics.

The flexibility of this proof-theoretic characterisation is then demonstrated by extending it in one direction to a sequent calculus for proving properties of logic programs, and in another direction to a constraint logic programming language.

It is argued that such proof-theoretic characterisations have at least the same depth and generality as standard characterisations, such as those using resolution or fixpoint semantics, while at the same time being less conceptually complex.

This is the companion technical report to the MFCS'89 paper [And89].

1 Introduction

Systems of logic are usually characterised by means of proof systems or truth-functional semantics, which are fairly direct formalisations of our intuitions about logical truth and deduction. Logic programming systems, on the other hand, are often characterised using notions of resolution theorem proving, decision procedures, or fixpoint semantics – notions which are somewhat removed from our intuitions about logic.

However, the readability of logic programs (and thus the power of the logic programming paradigm) seems to stem primarily from our knowledge about the meanings of logical formulae, rather than from our knowledge about resolution and fixpoint semantics. So if we can give a characterisation of a logic programming system in the form of a clear and understandable proof system, we should be able to capture better our intuitions about logic programs. In this paper, I provide such a characterisation.

For a given logic program Π, I give a formal, compositional operational semantics, LP_Π, for a simple logic programming system with parallel "or" (section 3). I also give a Gentzen-style natural deduction proof system, LPnd_Π [Gen69] (section 4), and prove that it characterises LP_Π exactly, in the sense that the set of theorems of LPnd_Π is exactly the set of existential closures of solvable queries of LP_Π.

The operational semantics is of interest in itself, as it makes explicit the assumptions about search strategy made in much of logic programming research. In logic programming jargon, the system described uses a left-to-right computation rule and a fair search rule. The proof system is valuable because it can be seen as a natural formalisation of the notions of truth and consequence expressed in Π, with all considerations of search strategy abstracted away.

This characterisation is very flexible in practical ways. Section 5 shows how the LP/LPnd system can be "Gentzenised" to form a sequent calculus which acts as a system for proving properties of logic programs, including inductive properties. Section 6 shows how LP/LPnd can be extended in another direction, to a constraint logic programming system in which inequality can be expressed as well as equality. In the future I hope to also characterise sequential "or" and some form of negation in this proof-theoretic framework.

In section 7, I discuss the issue of the conceptual complexity and generality of this proof-theoretic characterisation compared with the standard one. I hope to convince readers that this framework is at least a valuable addition to our overall understanding of the logic programming paradigm.

2 Definitional Preliminaries

As we will see in more detail later, the only connectives which are significant in logic programming are conjunction, disjunction, and the existential quantifier. We can 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 built up from predicate applications using only "and" (written " & "), "or" ("$\vee$"), and "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, *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 $\mathcal{L}$ 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. 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 $\mathbf{A}$ given to a logic programming system as a request to prove the existential closure of $\mathbf{A}$. Since the logic we use will turn out to be constructive, this is the same as a request to find a satisfying substitution for $\mathbf{A}$; the existential-closure view will just facilitate the comparison with the proof theory.

Definition 2.4 The *existential closure* of a formula $\mathbf{A}$, 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 $\mathbf{A}$.

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; **E**, **F**, **I**, **J** stand for special kinds of formulae called environments (to be defined later); **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 The Operational Semantics

As a reference point for the logic programming system being described, I give an operational semantics which is compositional in the sense of Plotkin [Plo81]. This formal system, LP, describes a non-deterministic machine; the machine solves queries by the usual method of maintaining an environment of bindings for variables, and doing sequences of unifications.

The judgments of LP will be expressions of the form $\mathbf{E} : \mathbf{A} \overset{\mathrm{LP}_\Pi}{\Rightarrow} \mathbf{F}$. We will read such a judgment as meaning "given the program Π, computing the formula **A** in the environment **E** results in the solution environment **F**".

3.1 Environments and Unification

LP models the method of doing unifications on environments by using special formulae to hold variable bindings, and by using a unification function $\mathrm{Unify}(\mathbf{s}, \mathbf{t}, \mathbf{E})$ which unifies terms **s** and **t** in the environment **E**, possibly returning a new environment **F**.

Definition 3.1 An *environment* is a formula of the form

$$\mathbf{x}_1 = \mathbf{t}_1 \;\&\; \ldots \;\&\; \mathbf{x}_n = \mathbf{t}_n \;\&\; \mathit{true}$$

$n \geq 0$, meeting the following conditions:

1. All the $\mathbf{x}_i$'s are distinct variables;
2. (Non-extraneity) Every variable appearing in the environment is one of the $\mathbf{x}_i$'s;

3. (Dereferencedness) For all $\mathbf{x}_i$ appearing in some $\mathbf{t}_j$, $\mathbf{t}_i \equiv \mathbf{x}_i$ (i.e. $\mathbf{t}_i$ is syntactically identical to $\mathbf{x}_i$).

The $\mathbf{t}_i$'s in the environment are called *values*; in particular, $\mathbf{t}_i$ is the *value of* $\mathbf{x}_i$.

An environment is thus just a formal representation of a variable substitution. For logic programming with the theory of simple first-order equality, we could actually use any such formalisation. I use this formulaic version because it takes the form of a list of logical constraints, and can therefore be generalised to other base theories. For instance, as we will see in section 6, generalising to the theory of first-order equality *and inequality* will require only a new definition of environment and unification, and the re-proving of some basic theorems.

The three conditions on environments are to ensure the termination and correctness of the Unify algorithm. The "dereferencedness" property, in particular, ensures that an environment has no loops of reference (e.g., $x = s(x)$, or $x = s(y)$ & $y = s(x)$). The terminology comes from the analogy of a variable with a pointer, and its value with the object it points to. A variable which points to itself, i.e. has itself as value, is one which has not yet been instantiated by any unification.

Definition 3.2 If $\mathbf{E}$ contains the equation $\mathbf{x} = \mathbf{x}$, then we call $\mathbf{x}$ an *uninstantiated variable* in $\mathbf{E}$.

Unify($\mathbf{s}$, $\mathbf{t}$, $\mathbf{E}$) returns an environment $\mathbf{F}$ iff $\mathbf{s}$ and $\mathbf{t}$ are unifiable, and returns an indication of failure otherwise. Its algorithm is equivalent to any other standard unification algorithm, for example the one from [Llo84]; but since we will want to prove it correct with respect to a logic *weaker* than first order, it seems best to give the algorithm and the proof of correctness in full.

1. If $\mathbf{s}$ or $\mathbf{t}$ contain variables which do not appear in $\mathbf{E}$, then return failure.
2. Dereference $\mathbf{s}$ and $\mathbf{t}$ with respect to $\mathbf{E}$.
3. If $\mathbf{s} \equiv \mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, $\mathbf{t} \equiv \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_m)$, and $\mathbf{f} \not\equiv \mathbf{g}$, then return failure.
4. If $\mathbf{s} \equiv \mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ and $\mathbf{t} \equiv \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$:
   (a) If $n = 0$, then return $\mathbf{E}$.
   (b) Otherwise, do n unifications: Unify($\mathbf{s}_1, \mathbf{t}_1, \mathbf{E}$) = $\mathbf{E}_1$, Unify($\mathbf{s}_2, \mathbf{t}_2, \mathbf{E}_1$) = $\mathbf{E}_2$, ..., Unify($\mathbf{s}_n, \mathbf{t}_n, \mathbf{E}_{n-1}$) = $\mathbf{F}$. If any of the unifications fails, then return failure; otherwise, return $\mathbf{F}$.
5. Otherwise, one of $\mathbf{s}$ and $\mathbf{t}$ must be a variable $\mathbf{x}$; let the other be the term $\mathbf{r}$.
   (a) If $\mathbf{x}$ occurs in $\mathbf{r}$, then return failure. (This is the celebrated "occurs check".)

(b) Otherwise, form $\mathbf{F}$ by replacing $\mathbf{x}$ by $\mathbf{r}$ everywhere in *values* in $\mathbf{E}$, and return $\mathbf{F}$.

To "dereference" a term $\mathbf{t}$ with respect to an environment $\mathbf{E}$ means simply to replace in it all occurrences of variables $\mathbf{x}_i$ from $\mathbf{E}$ by their corresponding values $\mathbf{t}_i$ from $\mathbf{E}$.

Theorem 3.3 Unify($\mathbf{s}$, $\mathbf{t}$, $\mathbf{E}$) always terminates.

Proof. The only way that Unify($\mathbf{s}$, $\mathbf{t}$, $\mathbf{E}$) could diverge is if it were to go into infinite recursion on case 4(b). However, note that if any call to Unify($\mathbf{s}$, $\mathbf{t}$, $\mathbf{E}$) succeeds returning some $\mathbf{F}$, and $\mathbf{F}$ is not identical to $\mathbf{E}$, then $\mathbf{F}$ has fewer uninstantiated variables than $\mathbf{E}$. Each recursive call, therefore, involves either a smaller number of uninstantiated variables in the environment, or the same number of uninstantiated variables and terms of smaller size. When there are no uninstantiated variables in the environment and the terms are of minimum size (i.e. nullary function symbols), the algorithm clearly terminates by case 3 or 4(a). Therefore, by induction (specifically, strong induction up to ω^ω), the algorithm for Unify($\mathbf{s}$, $\mathbf{t}$, $\mathbf{E}$) terminates for any choice of $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{E}$.
□

3.2 The System LP

In logic programming, we give the system a program Π and a query formula $\mathbf{A}$, and ask it to compute an answer substitution for the free variables in $\mathbf{A}$ – that is, a substitution θ such that $\mathbf{A}\theta$ is a theorem. Another way of looking at this is that we ask the system to prove the existential closure of $\mathbf{A}$ in a *constructive* way, by actually giving us the terms which correspond to the quantified variables.

The operational semantics LP_Π models this in the following way. We compute a query formula $\mathbf{A}$ given the program Π by finding an environment $\mathbf{F}$ such that $\mathbf{E}_0 : \exists[\mathbf{A}] \overset{\mathrm{LP}_\Pi}{\Rightarrow} \mathbf{F}$, where $\mathbf{E}_0$ is just the "empty environment" *true*. In general, the subsidiary steps in such a derivation will be judgments of the form $\mathbf{E} : \mathbf{B} \overset{\mathrm{LP}_\Pi}{\Rightarrow} \mathbf{E}'$, where $\mathbf{E}'$ is the result of computing the formula $\mathbf{B}$ in the environment $\mathbf{E}$.

The rules of derivation for LP (we will drop the subscript when convenient) are in Figure 1. See Figure 2 for an example derivation. In the example, the language $\mathcal{L}$ is assumed to have the unary operation symbols a, b, and *nil*, the binary operation $[\_|\_]$, and the predicate symbol *mem*. The program Π is assumed to contain the definition of the *mem* predicate from section 2.

LP describes an interpreter which derives the solution environment $\mathbf{F}$ by inspecting the form of the query formula. For instance, to find an $\mathbf{F}$ such that $\mathbf{E} : (\mathbf{B} \,\&\, \mathbf{C}) \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$, it first recursively finds an $\mathbf{E}'$ such that $\mathbf{E} : \mathbf{B} \overset{\mathrm{LP}}{\Rightarrow} \mathbf{E}'$, and then recursively finds an $\mathbf{F}$ such that $\mathbf{E}' : \mathbf{C} \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$. Each predicate definition from

$$=: \quad \frac{}{\mathbf{E} : \mathbf{s} = \mathbf{t} \overset{\text{LP}}{\Rightarrow} \mathbf{F}} \qquad \& : \quad \frac{\mathbf{E} : \mathbf{B} \overset{\text{LP}}{\Rightarrow} \mathbf{E}' \quad \mathbf{E}' : \mathbf{C} \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : (\mathbf{B} \,\&\, \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

where $\text{Unify}(\mathbf{s}, \mathbf{t}, \mathbf{E})$ succeeds returning $\mathbf{F}$

$$\vee, 1 : \quad \frac{\mathbf{E} : \mathbf{B} \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : (\mathbf{B} \vee \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}} \qquad \vee, 2 : \quad \frac{\mathbf{E} : \mathbf{C} \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : (\mathbf{B} \vee \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

$$\exists : \frac{(\mathbf{x}' = \mathbf{x}' \,\&\, \mathbf{E}) : \mathbf{B}[\mathbf{x} := \mathbf{x}'] \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : \exists \mathbf{x}(\mathbf{B}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}} \qquad \mathbf{P} : \quad \frac{\mathbf{E} : \mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

where $\mathbf{x}'$ does not appear in $\mathbf{E}$

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

Figure 1: The rules for the operational semantics LP.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{E} : \mathbf{B} \overset{\text{LP}}{\Rightarrow} \overbrace{(t = [b|nil] \,\&\, h = a \,\&\, x = x \,\&\, true)}^{\mathbf{E}'}} \quad \dfrac{}{\mathbf{E}' : x = h \overset{\text{LP}}{\Rightarrow} \mathbf{F}}}{\overbrace{(t = t \,\&\, h = h \,\&\, x = x \,\&\, true)}^{\mathbf{E}} : (\mathbf{B} \,\&\, \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}} \quad \dfrac{}{\mathbf{E}' : \mathbf{C} \overset{\text{LP}}{\Rightarrow} \mathbf{F}}}{(h = h \,\&\, x = x \,\&\, true) : \exists t(\mathbf{B} \,\&\, \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}}{(x = x \,\&\, true) : \exists h \exists t(\overbrace{[a|[b|nil]] = [h|t]}^{\mathbf{B}} \,\&\, \overbrace{(x = h \vee mem(x, t))}^{\mathbf{C}}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}}{(x = x \,\&\, true) : mem(x, [a|[b|nil]]) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}}{true : \exists x \; mem(x, [a|[b|nil]]) \overset{\text{LP}}{\Rightarrow} \overbrace{(t = [b|nil] \,\&\, h = a \,\&\, x = a \,\&\, true)}^{\mathbf{F}}}$$

Figure 2: A simple derivation in LP.

Π serves to define a rule in LP_Π; each recursion "bottoms out" at a unification (=) step.

There are two rules for the case of a disjunctive query. The interpreter described by LP therefore uses a "parallel or" strategy to handle such queries. When it comes to find an $\mathbf{F}$ such that $\mathbf{E} : (\mathbf{B} \vee \mathbf{C}) \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$, it does so by simultaneously finding an $\mathbf{F}_1$ such that $\mathbf{E} : \mathbf{B} \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}_1$ and an $\mathbf{F}_2$ such that $\mathbf{E} : \mathbf{C} \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}_2$. An implementation can do this either by dovetailing (timesharing) or by actual parallel computation on a separate processor. The interpreter takes the result of the computation to be the first result returned by the two subsidiary computations, either $\mathbf{F}_1$ or $\mathbf{F}_2$.

Note the renaming that occurs in the $\exists$ rule to avoid variable capture. This is essential when working with recursive predicates, which may introduce many versions of the same local variable into a given computation.

4 The Natural Deduction System

Derivations in natural deduction [Gen69] and tableau [Smu68] proof systems follow closely the structure of our intuitions about deductive reasoning. They are thus closely related to the understandability which underlies the logic programming paradigm, and are good tools for characterising logical systems.

For each program Π, there is a natural-deduction style proof system, LPnd_Π (Figure 3), which will be shown to be equivalent to LP_Π in the following sense:

For all formulae $\mathbf{A}$, $\mathbf{E}_0 : \exists[\mathbf{A}] \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$ for some $\mathbf{F}$ if and only if $\vdash_{\mathbf{LPnd}} \exists[\mathbf{A}]$.

LPnd can thus be seen as a natural characterisation of the set of successful LP queries. In particular, the LPnd derivation corresponding to the example LP derivation from the last section can be found in Figure 4.

LPnd is closely related to traditional natural deduction systems with equality, and can be seen as an instance of Martin-Löf's intuitionistic theory of iterated inductive definitions [ML71]. The fact that our LP operational semantics is equivalent to this very weak proof system points up how weak standard logic programming languages are, compared with full first order logic.

The first observation leading to the soundness and completeness results of this section has to do with the unification algorithm. It confirms what we should expect of a sensible unification algorithm: that if two terms unify, it finds their most general unifier, and that otherwise it fails. Two lemmata lead to this result.

Lemma 4.1 (Unification Failure) For all terms $\mathbf{s}$ and $\mathbf{t}$ and environments $\mathbf{E}$ containing all the variables in $\mathbf{s}$ and $\mathbf{t}$: if $\mathrm{Unify}(\mathbf{s}, \mathbf{t}, \mathbf{E})$ fails, then there is no θ such that $\vdash_{\mathbf{LPnd}} (\mathbf{E} \,\&\, (\mathbf{s} = \mathbf{t}))\theta$.

Proof. By induction on the number n of recursive calls in the failed unification. Step 2, the dereferencing operation, does not affect the result, because θ satisfies

$$=: \quad \frac{}{\mathbf{s} = \mathbf{s}} \qquad \& : \quad \frac{\mathbf{B} \quad \mathbf{C}}{\mathbf{B} \,\&\, \mathbf{C}}$$

where $\mathbf{s}$ is closed

$$\vee,1 : \quad \frac{\mathbf{B}}{\mathbf{B} \vee \mathbf{C}} \qquad \vee,2 : \quad \frac{\mathbf{C}}{\mathbf{B} \vee \mathbf{C}}$$

$$\exists : \quad \frac{\mathbf{B}[\mathbf{x} := \mathbf{t}]}{\exists \mathbf{x}(\mathbf{B})} \qquad \mathbf{P} : \quad \frac{\mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}{\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}$$

where $\mathbf{t}$ is a closed term

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

Figure 3: The rules for the proof system LPnd.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[a|[b|nil]] = [a|[b|nil]]} \quad \dfrac{\dfrac{}{a = a}}{a = a \vee mem(a, [b|nil])}}{[a|[b|nil]] = [a|[b|nil]] \;\&\; (a = a \vee mem(a, [b|nil]))}}{\exists t([a|[b|nil]] = [a|t] \;\&\; (a = a \vee mem(a, t)))}}{\exists h \exists t([a|[b|nil]] = [h|t] \;\&\; (a = h \vee mem(a, t)))}}{mem(a, [a|[b|nil]])}}{\exists x \; mem(x, [a|[b|nil]])}$$

Figure 4: A simple derivation in LPnd.

$(\mathbf{E} \;\&\; (\mathbf{s} = \mathbf{t}))$ exactly when it satisfies $(\mathbf{E} \;\&\; (\mathbf{s}' = \mathbf{t}'))$, where $\mathbf{s}'$ and $\mathbf{t}'$ are the dereferenced versions of $\mathbf{s}$ and $\mathbf{t}$.

Case $n = 0$: Unify fails due to either step 3 or step 5(a). In both of these cases, there is clearly no θ under which $\mathbf{s}$ and $\mathbf{t}$ are identical.

Case $n > 0$: Unify fails due to the failure, in step 4(b), of some Unify$(\mathbf{s}_i, \mathbf{t}_i, \mathbf{E}_{i-1})$. By the induction hypothesis, there is no θ under which $\mathbf{s}_i$ and $\mathbf{t}_i$ are identical; but for $\mathbf{s}$ and $\mathbf{t}$ to be identical under some θ, all of their subparts must be identical under θ. So there is no θ under which $\mathbf{s}$ and $\mathbf{t}$ are identical, either.

□

Lemma 4.2 (Unification Success) For all terms $\mathbf{s}$ and $\mathbf{t}$, environments $\mathbf{E}$ containing all the variables in $\mathbf{s}$ and $\mathbf{t}$, and all closed-term substitutions θ referring only to the free variables of $\mathbf{E}$: if Unify$(\mathbf{s}, \mathbf{t}, \mathbf{E})$ succeeds returning $\mathbf{F}$, then $\vdash_{\mathbf{LPnd}} (\mathbf{E} \;\&\; (\mathbf{s} = \mathbf{t}))\theta$ iff $\vdash_{\mathbf{LPnd}} \mathbf{F}\theta$.

Proof. By induction on the number n of recursive calls in the unification. The remarks in the last lemma about the dereferencing operation still apply.

Case $n = 0$: Unify succeeds due to step 4(a) or 5(b). In case 4(a), clearly any θ will satisfy $\mathbf{f} = \mathbf{f}$. In case 5(b), θ satisfies $\mathbf{E}$ and $\mathbf{x} = \mathbf{r}$ iff it satisfies $\mathbf{E}$ and makes $\mathbf{x}$ and $\mathbf{r}$ identical, iff it satisfies $\mathbf{E}$ with all occurrences of $\mathbf{x}$ in values (including in the equation $\mathbf{x} = \mathbf{x}$) replaced by $\mathbf{r}$.

Case $n > 0$: Unify succeeds due to step 4(b). θ satisfies $\mathbf{E}$ and $\mathbf{s} = \mathbf{t}$ iff it satisfies $\mathbf{E}$ and each of $\mathbf{s}_1 = \mathbf{t}_1 \ldots \mathbf{s}_n = \mathbf{t}_n$, iff (by applications of the induction hypothesis) it satisfies each of the $\mathbf{E}_i$'s in turn, iff it satisfies $\mathbf{F}$.

□

Theorem 4.3 (Correctness of Unification) For all terms $\mathbf{s}$ and $\mathbf{t}$, environments $\mathbf{E}$ containing all the variables in $\mathbf{s}$ and $\mathbf{t}$, and all closed-term substitutions θ referring only to the free variables of $\mathbf{E}$, $\vdash_{\mathbf{LPnd}} (\mathbf{E} \;\&\; (\mathbf{s} = \mathbf{t}))\theta$ where Π contains the definition iff Unify$(\mathbf{s}, \mathbf{t}, \mathbf{E})$ succeeds, returning an $\mathbf{F}$ such that $\vdash_{\mathbf{LPnd}} \mathbf{F}\theta$.

Proof. By the previous two lemmata.

□

This correctness result is vacuous for environments with no satisfying substitutions; but no such environments exist, because of the following theorem.

Theorem 4.4 (Environment Satisfiability) For all environments $\mathbf{F}$, there is a closed-term substitution σ mentioning only the variables of $\mathbf{F}$ such that σ satisfies $\mathbf{F}$, i.e. such that $\vdash_{\mathbf{LPnd}} \mathbf{F}\sigma$.

Proof. Let $\mathbf{F}$ be

$$\mathbf{x}_1 = \mathbf{t}_1 \;\&\; \ldots \;\&\; \mathbf{x}_n = \mathbf{t}_n \;\&\; \mathit{true}$$

and let $\mathbf{y}_1, \ldots, \mathbf{y}_m$ be all the uninstantiated variables in $\mathbf{F}$. Then, from the definition of environment, if σ is the substitution

$$[\mathbf{x}_1 := \mathbf{t}_1, \ldots, \mathbf{x}_n := \mathbf{t}_n][\mathbf{y}_1 := 0, \ldots, \mathbf{y}_m := 0]$$

then σ satisfies $\mathbf{F}$.

□

We now wish to prove that all computations in the operational semantics LP correspond to theorems of LPnd. I call this the "soundness" of LP because derivations in LPnd correspond better to our intuitions about the truth of formulae; it could also be seen as the "completeness" of LPnd.

Theorem 4.5 (Soundness) (1) For all formulae $\mathbf{A}$, environments $\mathbf{E}$ containing all the free variables of $\mathbf{A}$, and all closed-term substitutions θ: if there is an $\mathbf{F}$ such that $\mathbf{E} : \mathbf{A} \overset{\mathbf{LP}}{\Rightarrow} \mathbf{F}$ and $\vdash_{\mathbf{LPnd}} \mathbf{F}\theta$, then $\vdash_{\mathbf{LPnd}} \mathbf{E}\theta$ and $\vdash_{\mathbf{LPnd}} \mathbf{A}\theta$.
(2) For all closed formulae $\mathbf{A}$, if there is an environment $\mathbf{F}$ such that $\mathbf{E}_0 : \mathbf{A} \overset{\mathbf{LP}}{\Rightarrow} \mathbf{F}$, then $\vdash_{\mathbf{LPnd}} \mathbf{A}$.

Proof of (1). By induction on the structure of the LP-derivation.

Case $\mathbf{A} \equiv (\mathbf{s} = \mathbf{t})$. From the Unification Correctness theorem.

Case $\mathbf{A} \equiv (\mathbf{B} \,\&\, \mathbf{C})$. Assume there is such an LP-derivation. Its last step looks like this:

$$\frac{\mathbf{E} : \mathbf{B} \overset{\text{LP}}{\Rightarrow} \mathbf{E}' \quad \mathbf{E}' : \mathbf{C} \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : (\mathbf{B} \,\&\, \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

By the induction hypothesis on the second subderivation, we have that $\vdash_{\text{LPnd}} \mathbf{E}'\theta$ and $\vdash_{\text{LPnd}} \mathbf{C}\theta$. Therefore by the induction hypothesis on the first subderivation, we have that $\vdash_{\text{LPnd}} \mathbf{E}\theta$ and $\vdash_{\text{LPnd}} \mathbf{B}\theta$, and thus that $\vdash_{\text{LPnd}} (\mathbf{B} \,\&\, \mathbf{C})\theta$.

Case $\mathbf{A} \equiv (\mathbf{B} \vee \mathbf{C})$. Assume there is such an LP-derivation. If its last step uses the first $\vee$ rule, then that step looks like this:

$$\frac{\mathbf{E} : \mathbf{B} \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : (\mathbf{B} \vee \mathbf{C}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

By the induction hypothesis, $\vdash_{\text{LPnd}} \mathbf{E}\theta$ and $\vdash_{\text{LPnd}} \mathbf{B}\theta$, and thus $\vdash_{\text{LPnd}} (\mathbf{B} \vee \mathbf{C})\theta$. The case when the last step uses the second $\vee$ rule is similar.

Case $\mathbf{A} \equiv (\exists \mathbf{x}(\mathbf{B}))$. Assume there is such an LP-derivation. Its last step looks like this:

$$\frac{(\mathbf{x}' = \mathbf{x}' \,\&\, \mathbf{E}) : \mathbf{B}[\mathbf{x} := \mathbf{x}'] \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : \exists \mathbf{x}(\mathbf{B}) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

By the induction hypothesis, $\vdash_{\text{LPnd}} (\mathbf{x}' = \mathbf{x}' \,\&\, \mathbf{E})\theta$ and $\vdash_{\text{LPnd}} (\mathbf{B}[\mathbf{x} := \mathbf{x}'])\theta$; so $\vdash_{\text{LPnd}} \mathbf{E}\theta$. θ must be either $\sigma[\mathbf{x}' := \mathbf{t}]$ or $\sigma[\mathbf{x} := \mathbf{s}][\mathbf{x}' := \mathbf{t}]$, for some σ mentioning neither $\mathbf{x}$ nor $\mathbf{x}'$. But then from the definition of substitution, $\mathbf{B}[\mathbf{x} := \mathbf{x}']\theta \equiv \mathbf{B}\sigma[\mathbf{x} := \mathbf{t}]$. We thus have an LPnd-derivation whose last step is

$$\frac{(\mathbf{B}\sigma)[\mathbf{x} := \mathbf{t}]}{\exists \mathbf{x}(\mathbf{B}\sigma)}$$

But from the definition of substitution, because σ does not mention $\mathbf{x}$, and $\mathbf{x}'$ does not appear in $\mathbf{B}$, $\exists \mathbf{x}(\mathbf{B}\sigma) \equiv (\exists \mathbf{x}(\mathbf{B}))\sigma \equiv (\exists \mathbf{x}(\mathbf{B}))\theta$. So $\vdash_{\text{LPnd}} \exists \mathbf{x}(\mathbf{B})\theta$.

Case $\mathbf{A} \equiv (\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n))$. Assume there is such an LP-derivation. Its last step looks like this:

$$\frac{\mathbf{E} : \mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}{\mathbf{E} : \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \overset{\text{LP}}{\Rightarrow} \mathbf{F}}$$

By the induction hypothesis, therefore, $\vdash_{\text{LPnd}} \mathbf{E}\theta$ and $\vdash_{\text{LPnd}} \mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n)\theta$, and thus $\vdash_{\text{LPnd}} \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)\theta$.

□(1)

Proof of (2). A consequence of (1), taking $\mathbf{E}$ to be $\mathbf{E}_0$ and θ to be the satisfying substitution of $\mathbf{F}$.

□(2)

So terminating queries of LP correspond to theorems of LPnd. If we had used a proof system for full first order logic with the **P**-rules from Π, in the place of LPnd, this would still hold. But LPnd also has the converse property, which such a proof system would *not* have: that its theorems all correspond to terminating queries of LP.

Theorem 4.6 (Completeness) (1) For all formulae $\mathbf{A}$, environments $\mathbf{E}$ containing all the free variables of $\mathbf{A}$, and all closed-term substitutions θ referring only to the free variables of $\mathbf{E}$: if $\vdash_{\mathrm{LPnd}} \mathbf{E}\theta$ and $\vdash_{\mathrm{LPnd}} \mathbf{A}\theta$, then there is an $\mathbf{F}$ and a closed-term substitution σ such that $\mathbf{E} : \mathbf{A} \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$ and $\vdash_{\mathrm{LPnd}} (\mathbf{F}\theta)\sigma$.
(2) For all closed formulae $\mathbf{A}$, if $\vdash_{\mathrm{LPnd}} \mathbf{A}$, then there is an $\mathbf{F}$ such that $\mathbf{E}_0 : \mathbf{A} \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$.

Proof of (1). By induction on the structure of the LPnd-derivation.

Case $\mathbf{A} \equiv (\mathbf{s} = \mathbf{t})$. From the Unification Correctness theorem.

Case $\mathbf{A} \equiv (\mathbf{B} \,\&\, \mathbf{C})$. Assume that $\vdash_{\mathrm{LPnd}} \mathbf{E}\theta$ and $\vdash_{\mathrm{LPnd}} \mathbf{A}\theta$. The last step of the latter derivation looks like this:

$$\frac{\mathbf{B} \quad \mathbf{C}}{\mathbf{B} \,\&\, \mathbf{C}}$$

From the induction hypothesis on the first subderivation, we have that there is an $\mathbf{E}'$ and σ' such that $\mathbf{E} : \mathbf{B} \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{E}'$ and $\vdash_{\mathrm{LPnd}} (\mathbf{E}'\theta)\sigma'$. Now, from the induction hypothesis on the second subderivation, we have that there is an $\mathbf{F}$ and σ'' such that $\mathbf{E}' : \mathbf{C} \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$ and $\vdash_{\mathrm{LPnd}} (\mathbf{F}(\theta\sigma'))\sigma''$. Taking σ to be $(\sigma'\sigma'')$, we have that $\mathbf{E} : \mathbf{A} \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$ and $\vdash_{\mathrm{LPnd}} (\mathbf{F}\theta)\sigma$.

Case $\mathbf{A} \equiv (\mathbf{B} \vee \mathbf{C})$. Assume that $\vdash_{\mathrm{LPnd}} \mathbf{E}\theta$ and $\vdash_{\mathrm{LPnd}} \mathbf{A}\theta$. If the latter derivation uses the first rule for $\vee$ in its last step, then that last step looks like this:

$$\frac{\mathbf{B}}{\mathbf{B} \vee \mathbf{C}}$$

From the induction hypothesis, we have that there is an $\mathbf{F}$ and σ such that $\mathbf{E} : \mathbf{B} \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$ and $\vdash_{\mathrm{LPnd}} (\mathbf{F}\theta)\sigma$. But by the first LP-rule for $\vee$, we have that $\mathbf{E} : (\mathbf{B} \vee \mathbf{C}) \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$. The case when the derivation of $\mathbf{A}$ uses the second rule for $\vee$ is similar.

Case $\mathbf{A} \equiv (\exists \mathbf{x}\mathbf{B})$. Assume that $\vdash_{\mathrm{LPnd}} \mathbf{E}\theta$ and $\vdash_{\mathrm{LPnd}} \mathbf{A}\theta$. Choose an $\mathbf{x}'$ which does not appear in $\mathbf{E}$, $\mathbf{B}$, or θ. Since $\vdash_{\mathrm{LPnd}} (\exists \mathbf{x}\mathbf{B})\theta$, clearly $\vdash_{\mathrm{LPnd}} (\exists \mathbf{x}'(\mathbf{B}[\mathbf{x} := \mathbf{x}']))\theta$; so $\vdash_{\mathrm{LPnd}} (\mathbf{B}[\mathbf{x} := \mathbf{x}'])\theta[\mathbf{x}' := \mathbf{t}]$, for some closed term $\mathbf{t}$. But then $\vdash_{\mathrm{LPnd}} (\mathbf{x}' = \mathbf{x}' \,\&\, \mathbf{E})\theta[\mathbf{x}' := \mathbf{t}]$; so by the induction hypothesis, there are $\mathbf{F}, \sigma'$ such that $(\mathbf{x}' = \mathbf{x}' \,\&\, \mathbf{E}) : \mathbf{B}[\mathbf{x} := \mathbf{x}'] \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}$ and $\vdash_{\mathrm{LPnd}} (\mathbf{F}\theta[\mathbf{x}' := \mathbf{t}])\sigma'$. So we have an LP-derivation whose last step is

$$\frac{(\mathbf{x}' = \mathbf{x}' \,\&\, \mathbf{E}) : \mathbf{B}[\mathbf{x} := \mathbf{x}'] \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}}{\mathbf{E} : \exists \mathbf{x}(\mathbf{B}) \overset{\mathrm{LP}}{\Rrightarrow} \mathbf{F}}$$

and taking $\sigma \equiv [\mathbf{x}' := \mathbf{t}]\sigma'$, we have that $\vdash_{\mathbf{LPnd}} (\mathbf{F}\theta)\sigma$.

Case $\mathbf{A} \equiv (\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n))$. Assume that $\vdash_{\mathbf{LPnd}} \mathbf{E}\theta$ and $\vdash_{\mathbf{LPnd}} \mathbf{A}\theta$. The last step of the latter derivation looks like this:

$$\frac{\mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}{\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}$$

From the induction hypothesis, there is an appropriate $\mathbf{F}$ and σ, and by the LP-rule for defined predicates, $\mathbf{E} : \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$.

□(1)

Proof of (2). A consequence of (1), taking $\mathbf{E}$ to be $\mathbf{E}_0$ and θ to be the empty substitution.

□(2)

So the set of solvable queries is exactly the same as the set of formulae whose closures are theorems of LPnd. LPnd is thus a more *precise* characterisation of LP than first order logic.

Corollary 4.7 (Characterisation) For all formulae $\mathbf{A}$, $\mathbf{E}_0 : \exists[\mathbf{A}] \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$ for some $\mathbf{F}$ iff $\vdash_{\mathbf{LPnd}} \exists[\mathbf{A}]$.

Proof. $\exists[\mathbf{A}]$ is a closed formula, so the soundness and completeness results apply.

□

A more useful form of this characterisation result states that the operational semantics LP *computes* the substitution θ which will make the query a theorem of LPnd. This is what we wanted originally from the logic programming system: the ability to show constructively that a given query has a satisfying substitution.

Corollary 4.8 (Correctness of Result) For all formulae $\mathbf{A}$:

1. If $\vdash_{\mathbf{LPnd}} \exists[\mathbf{A}]$, then
   (a) there is an $\mathbf{F}$ such that $\mathbf{E}_0 : \exists[\mathbf{A}] \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$, and moreover
   (b) there is a θ, effectively computable from $\mathbf{F}$, such that $\vdash_{\mathbf{LPnd}} \mathbf{A}\theta$.
2. Otherwise, there is no $\mathbf{F}$ such that $\mathbf{E}_0 : \exists[\mathbf{A}] \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$.

Proof.

1(a), 2. From the characterisation theorem.

1(b). Let $\mathbf{x}_1 \ldots \mathbf{x}_n$ be the free variables of $\mathbf{A}$. Then one step in the LP-derivation is

$$(\mathbf{x}_1 = \mathbf{x}_1 \;\&\; \ldots \;\&\; \mathbf{x}_n = \mathbf{x}_n \;\&\; \mathit{true}) : \mathbf{A} \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$$

Take θ to be the substitution satisfying $\mathbf{F}$ (this is indeed effectively computable from $\mathbf{F}$). Then by the soundness theorem, θ satisfies $\mathbf{A}$.

□

5 The Sequent Calculus

LPnd does characterise LP, but it is not as useful for proving things about programs as we might like; we often want to be able to prove things from assumptions. There are two main reasons for this. Firstly, when a version of the "cut" rule is present, we can use and re-use "lemma" proofs in other, larger proofs. Secondly, some form of proof from assumptions is an essential part of any inductive proof.

In this section, I give a Gentzen-style sequent calculus proof system, LPsc, which is sound with respect to LPnd in the sense that if a sequent

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n \rightarrow \mathbf{D}$$

is a theorem of LPsc, and (essentially) all the $\mathbf{A}_i$'s are theorems of LPnd, then $\mathbf{D}$ is also a theorem of LPnd. I then extend the calculus to one in which the universal quantifier ("$\forall$") and implication ("$\supset$") can be used to express meta-level properties of programs. Finally, I give a fairly general induction rule and prove its soundness. Some examples are given at the end of the section.

5.1 The Basic Calculus

Basic properties of equality between terms:

=, l1: $$\frac{\Gamma, \mathbf{s}_1 = \mathbf{t}_1, \ldots, \mathbf{s}_n = \mathbf{t}_n \rightarrow \Delta}{\Gamma, \mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) = \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \rightarrow \Delta}$$

=, l2: $$\frac{}{\Gamma, \mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) = \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_m) \rightarrow \Delta}$$

where $\mathbf{f} \not\equiv \mathbf{g}$

=, r: $$\frac{}{\Gamma \rightarrow \mathbf{t} = \mathbf{t}}$$

sub, l: $$\frac{\Gamma, \mathbf{A}(\mathbf{s}), \mathbf{s} = \mathbf{t} \rightarrow \Delta}{\Gamma, \mathbf{A}(\mathbf{t}), \mathbf{s} = \mathbf{t} \rightarrow \Delta}$$

sub, r: $$\frac{\Gamma, \mathbf{s} = \mathbf{t} \rightarrow \mathbf{A}(\mathbf{s})}{\Gamma, \mathbf{s} = \mathbf{t} \rightarrow \mathbf{A}(\mathbf{t})}$$

Connectives:

& , l: $$\frac{\Gamma, \mathbf{B}, \mathbf{C} \rightarrow \Delta}{\Gamma, (\mathbf{B} \,\&\, \mathbf{C}) \rightarrow \Delta}$$

& , r: $$\frac{\Gamma \rightarrow \mathbf{B} \quad \Gamma \rightarrow \mathbf{C}}{\Gamma \rightarrow \mathbf{B} \,\&\, \mathbf{C}}$$

$\vee$, l: $$\frac{\Gamma, \mathbf{B} \rightarrow \Delta \quad \Gamma, \mathbf{C} \rightarrow \Delta}{\Gamma, (\mathbf{B} \vee \mathbf{C}) \rightarrow \Delta}$$

$\vee$, r1: $\dfrac{\Gamma \to \mathbf{B}}{\Gamma \to \mathbf{B} \vee \mathbf{C}}$ $\vee$, r2: $\dfrac{\Gamma \to \mathbf{C}}{\Gamma \to \mathbf{B} \vee \mathbf{C}}$

$\exists$, l $\dfrac{\Gamma, \mathbf{B}[\mathbf{x} := \mathbf{y}] \to \Delta}{\Gamma, \exists \mathbf{x} \mathbf{B} \to \Delta}$ $\exists$, r $\dfrac{\Gamma \to \mathbf{B}[\mathbf{x} := \mathbf{t}]}{\Gamma \to \exists \mathbf{x} \mathbf{B}}$

where $\mathbf{y}$ does not appear in the lower sequent

where $\mathbf{t}$ is any term

Defined predicates:

left: $\dfrac{\Gamma, \mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \to \Delta}{\Gamma, \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n) \to \Delta}$ right: $\dfrac{\Gamma \to \mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}{\Gamma \to \mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)}$

where $(\mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \leftrightarrow \mathbf{A}(\mathbf{x}_1, \ldots, \mathbf{x}_n))$ is in the program Π

Structural rules:

Cut: $\dfrac{\Gamma \to \mathbf{B} \quad \Gamma, \mathbf{B} \to \Delta}{\Gamma \to \Delta}$ Thinning: $\dfrac{\Gamma \to \Delta}{\Gamma, \mathbf{A} \to \Delta}$

Axiom: $\dfrac{}{\Gamma, \mathbf{A} \to \mathbf{A}}$

In LPnd, free variables were not allowed in any formulae. In LPsc, they must be allowed because of the need for a rule to introduce an existential quantifier on the left-hand side of the sequent. The presence of free variables, in turn, requires the presence of more complex rules for equality.

Free variables in LPsc derivations can stand for *any* term, however, so LPsc has more representational power than LPnd. The fact that free variables have this role is a consequence of the soundness theorem of LPsc:

Theorem 5.1 (Soundness of LPsc) Let the sequent S be

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n \to \mathbf{D}$$

Let θ be a closed-term substitution which mentions all the free variables in S. Then if S is derivable in LPsc, and if

$$\vdash_{\mathbf{LPnd}} \mathbf{A}_1\theta, \vdash_{\mathbf{LPnd}} \mathbf{A}_2\theta, \ldots, \vdash_{\mathbf{LPnd}} \mathbf{A}_n\theta$$

then $\vdash_{\mathbf{LPnd}} \mathbf{D}\theta$.

Proof. By induction on the structure of the LPsc derivation. Cases are on the last rule application in the derivation. In all cases, if one of the formulae in

Γ is not a theorem under θ, or if Δ is a formula which is a theorem under θ (as applicable), the result follows immediately. I use the phrase "$\mathbf{A}$ is a theorem under θ" to mean that $\vdash_{\mathrm{LPnd}} \mathbf{A}\theta$.

=, ll. Assume that the equality in the lower sequent is a theorem under θ. Then each $\mathbf{s}_i$ must be identical to its respective $\mathbf{t}_i$ under θ, so all the equalities in the upper sequent must also be theorems under θ. By the induction hypothesis, then, $\mathbf{D}$ is a theorem under θ.

=, l2. The equality cannot be a theorem under any substitution, so the result trivially holds.

=, r. The equality must be a theorem under any closed-term substitution which mentions all the free variables of S.

sub, l and r. Assume that the equality in the lower sequent is a theorem under θ. Then $\mathbf{s}$ and $\mathbf{t}$ (and thus $\mathbf{A}(\mathbf{s})$ and $\mathbf{A}(\mathbf{t})$) must be identical under θ. By the induction hypothesis, the result follows.

& , l. The only way that $(\mathbf{B} \,\&\, \mathbf{C})\theta$ can be a theorem is if both $\mathbf{B}\theta$ and $\mathbf{C}\theta$ are theorems. The result follows from the induction hypothesis.

$\vee$, l. If $(\mathbf{B} \vee \mathbf{C})\theta$ is a theorem, then either $\mathbf{B}\theta$ or $\mathbf{C}\theta$ must be a theorem. But then the result follows by applying the induction hypothesis to the sequent containing the theorem.

$\exists$, l. Assume that $\vdash_{\mathrm{LPnd}} (\exists \mathbf{x}\mathbf{B})\theta$. Then there must be a $\theta' \equiv \theta[\mathbf{y} := \mathbf{t}]$, for some $\mathbf{t}$, such that $\vdash_{\mathrm{LPnd}} (\mathbf{B}[\mathbf{x} := \mathbf{y}])\theta'$. But since $\mathbf{y}$ does not appear in Γ, we have that $\vdash_{\mathrm{LPnd}} \mathbf{A}_i\theta'$ for all $\mathbf{A}_i$ in Γ; so by the induction hypothesis, $\vdash_{\mathrm{LPnd}} \mathbf{D}\theta'$. But since $\mathbf{y}$ does not appear in $\mathbf{D}$ either, we know that $\vdash_{\mathrm{LPnd}} \mathbf{D}\theta$.

Defined predicates, left. There is only one way to prove predicate application formulae in LPnd, so whenever $\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ is provable, so is $\mathbf{A}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$. (This, perhaps, can be taken as the final justification for our use of the "$\leftrightarrow$" symbol in predicate definitions.)

& , r; $\vee$, r; $\exists$, r; and defined predicates, right. These just follow straightforwardly from the corresponding rules in LPnd.

Cut. By the induction hypothesis, whenever the Γ formulae and $\mathbf{B}$ are all theorems under θ, so is $\mathbf{D}$. However, by the induction hypothesis again, whenever all the Γ formulae are theorems under θ, $\mathbf{B}$ is a theorem under θ, and thus so is $\mathbf{D}$.

Thinning, structural axiom: trivial.

□

An important practical corollary to this theorem shows that sequents have a deep operational interpretation. Let us say that a formula $\mathbf{A}$ "succeeds as a query" if there is some $\mathbf{F}$ such that $\mathbf{E}_0 : \exists[\mathbf{A}] \overset{\mathrm{LP}}{\Rightarrow} \mathbf{F}$.

Theorem 5.2 (Operational Interpretation of LPsc) (1) Let the sequent S be

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n \rightarrow \mathbf{D}$$

Let θ be *any* substitution. Then if S is derivable in LPsc, and if $\mathbf{A}_1\theta$ & $\mathbf{A}_2\theta$ & ... & $\mathbf{A}_n\theta$ succeeds as a query, then $\mathbf{D}\theta$ succeeds as a query.

(2) If $\rightarrow \mathbf{D}$ is derivable in LPsc, then for any θ, $\mathbf{D}\theta$ succeeds as a query.

Proof of (1). From the soundness of LP, we know that there is some closed-term substitution σ such that $\mathbf{A}_1\theta\sigma$ & $\mathbf{A}_2\theta\sigma$ & ... & $\mathbf{A}_n\theta\sigma$ is a theorem of LPnd; therefore each $\mathbf{A}_i\theta\sigma$ is also a theorem of LPnd. By the soundness of LPsc, we have that $\mathbf{D}\theta\sigma$ is a theorem of LPnd; so by the completeness of LP, we have that $\mathbf{D}\theta$ succeeds as a query.
□(1)

Proof of (2). Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be the free variables of $\mathbf{D}\theta$. Then by the soundness of LPsc, we have that

$$\vdash_{\text{LPnd}} \mathbf{D}\theta[\mathbf{x}_1 := 0, \ldots, \mathbf{x}_n := 0]$$

and by the completeness of LP, we have that $\mathbf{D}\theta$ succeeds as a query.
□(2)

We now have the ability to prove that a query terminates successfully based on the assumption that some other query terminates successfully, without having to prove the assumption. This is the first step towards a useful sequent calculus.

5.2 Adding More Connectives

A problem with LPsc as presented in the last section is that it ties us to the connectives found in logic programs: & , ∨, and ∃. This does not give us much power to express useful properties of logic programs, properties that we would like to be able to prove. For instance, with ∀ (for all) and ⊃ (implies), we could express the interesting property

$$\forall x \forall y(mem(x, y) \supset \forall z\ mem(x, [z|y]))$$

which could not be expressed well using the logic programming connectives. Here I introduce the notion of *assertions*, first-order formulae which make these kind of meta-level statements about the satisfiability of queries.

Definition 5.3 Recall that a program formula is one containing only predicate applications, & , ∨, and ∃. An *assertion* is defined recursively as follows:

- A program formula is an assertion.
- If $\mathbf{A}$ and $\mathbf{B}$ are assertions, then so are $\forall \mathbf{x}(\mathbf{A})$ and $\mathbf{A} \supset \mathbf{B}$.

An *assertion sequent* is a sequent made up of assertions.

Assertions are just a class of first-order formulae, but because they cannot appear in programs, we interpret them as making meta-level statements about the LPnd-provability (and thus success as queries) of their constituent program formulae.

Definition 5.4 An assertion is *LPnd-valid* just in the following cases.

- A program formula $\mathbf{A}$ is LPnd-valid if $\vdash_{\mathrm{LPnd}} \mathbf{A}$.
- A formula $\forall \mathbf{x}\mathbf{B}$ is LPnd-valid if, for all closed terms $\mathbf{t}$, $\mathbf{B}[\mathbf{x} := \mathbf{t}]$ is LPnd-valid.
- A formula $\mathbf{B} \supset \mathbf{C}$ is LPnd-valid if either $\mathbf{B}$ is not LPnd-valid or $\mathbf{C}$ is LPnd-valid.

By this interpretation, an informal reading of the example assertion $\forall x \forall y(mem(x,y) \supset \forall z\ mem(x,[z|y]))$ would be "for any closed terms $\mathbf{r}$ and $\mathbf{s}$, if $\vdash_{\mathrm{LPnd}} mem(\mathbf{r},\mathbf{s})$, then for any closed term $\mathbf{t}$, $\vdash_{\mathrm{LPnd}} mem(\mathbf{r},[\mathbf{s}|\mathbf{t}])$". By the completeness theorem for LP, we can also read this as: "if $mem(\mathbf{r},\mathbf{s})$ succeeds as a query to LP, so does $mem(\mathbf{r},[\mathbf{s}|\mathbf{t}])$."

With this interpretation in mind, we now can define a new calculus, LPsca_{Π}, in which the judgments are assertion sequents. The rules of LPsca_{Π} consist of the rules for LPsc_{Π} and the following additional rules:

$\forall$, l: $$\frac{\Gamma, \mathbf{B}[\mathbf{x} := \mathbf{t}] \rightarrow \Delta}{\Gamma, \forall \mathbf{x}(\mathbf{B}) \rightarrow \Delta}$$ where $\mathbf{t}$ is any term

$\forall$, r: $$\frac{\Gamma \rightarrow \mathbf{B}[\mathbf{x} := \mathbf{y}]}{\Gamma \rightarrow \forall \mathbf{x}(\mathbf{B})}$$ where $\mathbf{y}$ does not appear in the lower sequent

$\supset$, l: $$\frac{\Gamma, \mathbf{C} \rightarrow \Delta \quad \Gamma \rightarrow \mathbf{B}}{\Gamma, \mathbf{B} \supset \mathbf{C} \rightarrow \Delta}$$

$\supset$, r: $$\frac{\Gamma, \mathbf{B} \rightarrow \mathbf{C}}{\Gamma \rightarrow \mathbf{B} \supset \mathbf{C}}$$

LPsca is sound in the following sense:

Theorem 5.5 (Soundness of LPsca) Let the assertion sequent S be

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n \rightarrow \mathbf{D}$$

Let θ be a closed-term substitution which mentions all the free variables in S. Then if S is a theorem of LPsc, and if

$$\mathbf{A}_1\theta, \mathbf{A}_2\theta, \ldots, \mathbf{A}_n\theta$$

are all LPnd-valid, then so is $\mathbf{D}\theta$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{mem(x,y) \to [z|y] = [z|y]} \quad \dfrac{\dfrac{}{mem(x,y) \to mem(x,y)}}{mem(x,y) \to x = z \vee mem(x,y)}}{mem(x,y) \to [z|y] = [z|y] \;\&\; (x = z \vee mem(x,y))}}{mem(x,y) \to \exists t([z|y] = [z|t] \;\&\; (x = z \vee mem(x,t)))}}{mem(x,y) \to \exists h \exists t([z|y] = [h|t] \;\&\; (x = h \vee mem(x,t)))}}{mem(x,y) \to mem(x,[z|y])}}{mem(x,y) \to \forall z\ mem(x,[z|y]))}}{\to (mem(x,y) \supset \forall z\ mem(x,[z|y]))}}{\to \forall y(mem(x,y) \supset \forall z\ mem(x,[z|y]))}}{\to \forall x \forall y(mem(x,y) \supset \forall z\ mem(x,[z|y]))}$$

Figure 5: A simple LPsca derivation.

Proof. By induction on the structure of the LPsca derivation.

Cases from the LPsc rules: as in the proof of soundness of LPsc. The rules for substitution and the structural rules are unaffected by the presence of assertions rather than program formulae; and the other rules apply only to program formulae, which are interpreted in the same way here as in the proof of soundness of LPsc.

Case $\forall$, l. Assume that $\forall \mathbf{x}\mathbf{B}\theta$ is LPnd-valid. Then for any closed term $\mathbf{t}$, including the one mentioned in the derivation, $\mathbf{B}[\mathbf{x} := \mathbf{t}]\theta$ is LPnd-valid. By the induction hypothesis, then, $\mathbf{D}\theta$ must be LPnd-valid.

Case $\forall$, r. Assume that all the Γ formulae are LPnd-valid under θ. Then, because none of the Γ formulae have $\mathbf{y}$ free, they are also LPnd-valid under $[\mathbf{y} := \mathbf{t}]\theta$, for any closed $\mathbf{t}$. Then by the induction hypothesis, $\mathbf{B}[\mathbf{x} := \mathbf{y}][\mathbf{y} := \mathbf{t}]\theta$, which is just $\mathbf{B}[\mathbf{x} := \mathbf{t}]\theta$, is also LPnd-valid for any closed $\mathbf{t}$. Therefore $\forall \mathbf{x}\mathbf{B}\theta$ is also LPnd-valid.

Case $\supset$, l. Assume that the Γ formulae and $(\mathbf{B} \supset \mathbf{C})$ are all LPnd-valid under θ. By the induction hypothesis on the second premiss, $\mathbf{B}$ must be LPnd-valid under θ as well, so by the definition of LPnd-validity, so must $\mathbf{C}$. But then by the induction hypothesis on the first premiss, $\mathbf{D}$ must also be LPnd-valid under θ.

Case $\supset$, r. Assume that all the Γ formulae are LPnd-valid under θ. Then by the induction hypothesis, if $\mathbf{B}$ is also LPnd-valid under θ, so is $\mathbf{C}$. Thus either $\mathbf{B}$ is not LPnd-valid under θ, or $\mathbf{C}$ is; so $(\mathbf{B} \supset \mathbf{C})\theta$ is LPnd-valid.

□

LPnd-valid assertion sequents also have the expected operational interpretation, so LPsca-derivable sequents say the same kinds of things about queries that LPsc-derivable sequents do. An LPsca derivation of the example formula from earlier in this subsection can be found in figure 5.

5.3 Adding Induction

The soundness of LPsca means that we can prove meaningful things about successful termination of queries, based on the successful termination of other queries. We would also like this calculus to be complete in the following sense: if whenever

$$\mathbf{A}_1\theta, \mathbf{A}_2\theta, \ldots, \mathbf{A}_n\theta$$

are all LPnd-valid we have that $\mathbf{D}\theta$ is also valid, then

$$\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n \rightarrow \mathbf{D}$$

is a theorem of LPsc.

Unfortunately, we can never achieve this result for all languages $\mathcal{L}$. Say $\mathcal{L}$ is the language of Peano arithmetic and the program Π contains rules for addition and multiplication. Then the presence of universal quantification, implication, and the formula $false \equiv (0 = s(0))$, gives us the power of Peano arithmetic, which has no complete proof system.

However, we can significantly increase the extent to which LPsca is complete in this sense, by adding a rule for induction. If $\mathbf{f}$ is an n-ary function symbol, $S_{\mathbf{f}}$ be an abbreviation for the sequent

$$\Gamma, \mathbf{B}[\mathbf{x} := \mathbf{t}_1], \ldots, \mathbf{B}[\mathbf{x} := \mathbf{t}_n] \rightarrow \mathbf{B}[\mathbf{x} := \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)]$$

Then we can add the following rule to LPsca_Π, where Π is a program over the language $\mathcal{L}$:

$$\frac{S_{\mathbf{f}_1} \quad \ldots \quad S_{\mathbf{f}_m}}{\Gamma \rightarrow \forall \mathbf{x} \mathbf{B}}$$

where $\mathbf{f}_1, \ldots, \mathbf{f}_m$ are all the function symbols in $\mathcal{L}$.

Example: let the language $\mathcal{L}$ have only the nullary function 0 and the unary function s. Let the program Π contain the definitions

$$\begin{array}{rll} Add(x,y,z) \leftrightarrow & (x = 0 \ \& \ y = z) \\ \vee & (\exists x_1 \exists z_1 (x = s(x_1) \ \& \ z = s(z_1) \ \& \ Add(x_1, y, z_1))) \end{array}$$

Then we can prove, for example, the celebrated theorem that the sum of any number and 0 is that number.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\rightarrow 0 = 0} \quad \dfrac{}{\rightarrow 0 = 0}}{\rightarrow (0 = 0 \ \& \ 0 = 0)}}{\rightarrow (0 = 0 \ \& \ 0 = 0) \vee \ldots} \quad \dfrac{\dfrac{}{\rightarrow Add(0,0,0)} \qquad \vdots}{Add(x,0,x) \rightarrow Add(s(x),0,s(x))}}{\rightarrow \forall z Add(z,0,z)}$$

The proof of $Add(x,0,x) \rightarrow Add(s(x),0,s(x))$ in the vertical ellipsis ($\vdots$) is as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\rightarrow s(x) = s(x)} \quad \dfrac{}{\rightarrow s(x) = s(x)}}{\rightarrow s(x) = s(x)\ \&\ s(x) = s(x)}}{Add(x,0,x) \rightarrow s(x) = s(x)\ \&\ s(x) = s(x)} \quad \dfrac{}{Add(x,0,x) \rightarrow Add(x,0,x)}}{Add(x,0,x) \rightarrow s(x) = s(x)\ \&\ s(x) = s(x)\ \&\ Add(x,0,x)}}{Add(x,0,x) \rightarrow \exists z_1(s(x) = s(x)\ \&\ s(x) = s(z_1)\ \&\ Add(x,0,z_1))}}{Add(x,0,x) \rightarrow \exists x_1 \exists z_1(s(x) = s(x_1)\ \&\ s(x) = s(z_1)\ \&\ Add(x_1,0,z_1))}}{Add(x,0,x) \rightarrow \ldots \vee (\exists x_1 \exists z_1(s(x) = s(x_1)\ \&\ s(x) = s(z_1)\ \&\ Add(x_1,0,z_1)))}}{Add(x,0,x) \rightarrow Add(s(x),0,s(x))}$$

We can now prove that LPsca with induction is sound in the same sense as it was before. The proof is the same, except for an additional clause in the case that the last rule used was the induction rule.

Proof. Assume that all the Γ formulae are LPnd-valid under θ. It suffices to prove that for every term $\mathbf{t}$, $\mathbf{B[x := t]}$ is LPnd-valid under θ. We can do this by induction on the structure of $\mathbf{t}$; each case corresponds to one of the premisses of the induction rule, and one application of the induction hypothesis.
□

6 Equality and Inequality

In the last section, I described one extension to the LP/LPnd system, namely the derivation of a sequent calculus LPsca. In this section, I describe how we can extend LP and LPnd in a different direction, to a system which handles inequality as well as equality over first-order terms. I would expect that the resultant system, LPi/LPndi, could itself be extended to a sequent calculus in the same manner as LP/LPnd was extended to LPsca.

The inequality extension will allow us to write useful predicates such as *notmem*, which tests whether an element is *not* in a list:

$$notmem(x,l) \leftrightarrow l = Nil \vee \exists y \exists m(l = [y|m]\ \&\ x \neq y\ \&\ notmem(x,m))$$

The extension will involve simple changes to the characterising proof system and operational semantics, an expanded version of the unification algorithm, and the re-proving of some of the lemmata leading to the characterisation result.

I will also discuss connections this work has with other research in constraint logic programming.

6.1 The Language and Characterising Proof System

We will find it convenient to modify the characterising proof system first, to give us a goal to work towards in the development of the modified operational semantics.

We will start by making further assumptions about the language $\mathcal{L}$ in which programs are written. For simplicity, we will treat inequality as a predicate like

any other: $\mathcal{L}$ is assumed to contain a binary predicate $\neq$, written in infix. Using the form $\mathbf{s} \neq \mathbf{t}$ instead of $\neg(\mathbf{s} = \mathbf{t})$ will allow us to avoid the complications which accompany explicit negation.

Another simplifying assumption we will make is that $\mathcal{L}$ has an infinite number of terms – that is, that it has at least one non-nullary function symbol. This assumption allows us to avoid the complication of deciding inequality over a finite domain, which is a much more computationally and algorithmically complex problem than inequality over an infinite domain. (Artificial intelligence applications sometimes involve such problems, where they are referred to as constraint satisfaction problems; Waltz's algorithm for labelling the convexity of vertices in line drawings is one such application. See [Mac85] for more details.)

This said, we have only a simple extension to make to LPnd for it to characterise inequality. We want the inequality predicate, $\neq$, to be the complement of equality for pairs of closed terms. Let us define LPndi as the proof system having all the rules of LPnd, and the following additional rule:

$\neq$: $\dfrac{}{\mathbf{s} \neq \mathbf{t}}$

where $\mathbf{s}$ and $\mathbf{t}$ are non-identical closed terms

6.2 Subsumption and Operational Semantics

We must now alter the definitions of environment and unification to handle this new predicate $\neq$. We want to preserve the important properties of environment satisfiability and unification termination and correctness. More general versions of unification are usually called *subsumption* algorithms, and that is the terminology we will use here.

First, a definition of the new environments. In this section, environments of the old form will be called "equality environments" and those of the new form will be called "inequality environments".

Definition 6.1 An *inequality list* is a formula of the form

$$\mathbf{s}_1 \neq \mathbf{t}_1 \vee \ldots \vee \mathbf{s}_n \neq \mathbf{t}_n \vee \mathit{false}$$

where $n \geq 0$.

An *inequality environment* is a formula of the form

$$\mathbf{E} \;\&\; \mathbf{L}_1 \;\&\; \ldots \;\&\; \mathbf{L}_m$$

where $\mathbf{E}$ is an equality environment, $m \geq 0$, and each $\mathbf{L}_i$ is an inequality list formula referring only to variables in $\mathbf{E}$.

An inequality environment is in *normal form* if all inequality subformulae in it are of the form $\mathbf{x} \neq \mathbf{t}$.

Now, the subsumption function Subsume($\mathbf{A}$, $\mathbf{I}$) will take a formula $\mathbf{A}$ of *either* the form $\mathbf{s} = \mathbf{t}$ *or* the form $\mathbf{s} \neq \mathbf{t}$, and an inequality environment $\mathbf{I}$ in normal form. It will either return a failure indication, or succeed returning a new normal form inequality environment $\mathbf{J}$. Its algorithm is as follows.

1. If $\mathbf{A}$ is of the form $\mathbf{s} = \mathbf{t}$, then call Unify($\mathbf{s}$, $\mathbf{t}$, $\mathbf{E}$). If this fails, return failure. If it succeeds returning $\mathbf{F}$, call Split($\mathbf{F}$ & $\mathbf{L}_1$ & ... & $\mathbf{L}_m$), returning whatever it returns.

2. Otherwise, $\mathbf{A}$ is of the form $\mathbf{s} \neq \mathbf{t}$; call Split($\mathbf{E}$ & $\mathbf{L}_1$ & ... & $\mathbf{L}_m$ & ($\mathbf{s} \neq \mathbf{t} \vee \mathit{false}$)), returning whatever it returns.

Split is a function which takes an inequality environment $\mathbf{I}$ and returns failure if it is not satisfiable, and otherwise returns a normal form inequality environment equivalent to $\mathbf{I}$. The algorithm for Split($\mathbf{I}$) is as follows:

1. Dereference all inequality formulae (replace all variables in all the $\mathbf{L}_i$'s by their values from $\mathbf{E}$).

2. While there is still an inequality in $\mathbf{I}$ of the form $\mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \neq \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$, replace it in its inequality list by $\mathbf{s}_1 \neq \mathbf{t}_1 \vee \ldots \vee \mathbf{s}_n \neq \mathbf{t}_n$.

3. If any inequality lists contain formulae of the form $\mathbf{f}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \neq \mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_m)$, or of the form $\mathbf{s} = \mathbf{t}$ where $\mathbf{s}$ is a proper subterm of $\mathbf{t}$ or vice versa, then eliminate those inequality lists from $\mathbf{I}$.

4. If any inequalities are of the form $\mathbf{t} \neq \mathbf{t}$, then eliminate them from their respective inequality lists.

5. If any inequality lists are simply false (the "null list"), return failure.

6. Now all inequality subformulae must be of either the form $\mathbf{x} \neq \mathbf{t}$ or the form $\mathbf{t} \neq \mathbf{x}$; replace those of the latter form by their inverses, $\mathbf{x} \neq \mathbf{t}$.

7. Return the modified environment.

This algorithm terminates, as it can only execute the body of the loop in step 2 a finite number of times; therefore the algorithm for Subsume also terminates. Split clearly returns a normal form inequality environment.

The new operational semantics LPi is formed from LP by replacing the rule for equality by the following, more general rule:

$=, \neq$: $\dfrac{}{\mathbf{I} : \mathbf{A} \overset{\text{LPi}}{\Rightarrow} \mathbf{J}}$

where $\mathbf{A}$ is of the form $\mathbf{s} = \mathbf{t}$ or of the form $\mathbf{s} \neq \mathbf{t}$,
and Subsume($\mathbf{A}$, $\mathbf{I}$) succeeds returning $\mathbf{J}$

6.3 The Characterisation Theorems

We have now to prove only the correctness of the new subsumption algorithm, and the satisfiability of normal form inequality environments. Then all the rest of the results which obtain for LP and LPnd will obtain for LPi and LPndi.

We have the following lemma to the main subsumption correctness theorem.

Lemma 6.2 For all normal form inequality environments $\mathbf{I}$ and all closed-term substitutions θ referring only to the free variables of $\mathbf{I}$, $\vdash_{\text{LPndi}} (\mathbf{I})\theta$ iff Split($\mathbf{I}$) succeeds, returning a $\mathbf{J}$ such that $\vdash_{\text{LPnd}} \mathbf{J}\theta$.

Proof. It suffices to show that if Split($\mathbf{I}$) fails, there is no satisfying substitution for $\mathbf{I}$; and if Split($\mathbf{I}$) succeeds returning $\mathbf{J}$, $\vdash_{\text{LPnd}} \mathbf{I}\theta$ iff $\vdash_{\text{LPnd}} \mathbf{J}\theta$.

If Split($\mathbf{I}$) fails, it must be because of step 5. Clearly, no conjunction containing *false* as a conjunct can be satisfiable.

If Split($\mathbf{I}$) succeeds returning $\mathbf{J}$, it suffices to show that every transformation done on the environment preserves the property of being a theorem under the substitution θ, and that the inverse of the transformation does as well.

(Step 1) Clearly the dereferencing preserves this property.

(Step 2) If the derivation of $\mathbf{I}\theta$ depends on a derivation of $\mathbf{f}(\mathbf{s}_1,\ldots,\mathbf{s}_n)\theta \neq \mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)\theta$, then it must be the case that the two terms are not identical; thus they must have some non-identical corresponding parts, ie. there must be an i such that $\vdash_{\text{LPndi}} \mathbf{s}_i\theta \neq \mathbf{t}_i\theta$. This is the case iff $\vdash_{\text{LPndi}} (\mathbf{s}_1 \neq \mathbf{t}_1 \vee \ldots \vee \mathbf{s}_n \neq \mathbf{t}_n)\theta$. The converse also applies.

(Step 3) If $\mathbf{I} \equiv \mathbf{E}$ & $\mathbf{L}_1$ & ... & $\mathbf{L}_m$, and some $\mathbf{L}_i$ has such an inequality, then clearly that $\mathbf{L}_i$ is satisfied by any substitution; and thus $\mathbf{I}$ is satisfied by θ iff $\mathbf{E}$ & $\mathbf{L}_1$ & ... & $\mathbf{L}_{i-1}$ & $\mathbf{L}_{i+1}$ & ... & $\mathbf{L}_m$ is satisfied by θ.

(Step 4) If an $\mathbf{L}_i$ is satisfied by θ and contains such an inequality, that inequality cannot be the disjunct which is derivable under the substitution; so eliminating it produces an environment which is still satisfied by θ. The converse holds because adding a disjunct to an inequality list cannot make it underivable.

(Step 6) Clearly the inequality relation defined by the LPndi rule is symmetric.

□

Theorem 6.3 (Correctness of Subsumption) For all formulae $\mathbf{A}$ of the form $\mathbf{s} = \mathbf{t}$ or $\mathbf{s} \neq \mathbf{t}$, inequality environments $\mathbf{I}$ containing all the variables in $\mathbf{A}$, and all closed-term substitutions θ referring only to the free variables of $\mathbf{I}$, $\vdash_{\text{LPnd}} (\mathbf{I}$ & $\mathbf{A})\theta$ iff Subsume($\mathbf{A}, \mathbf{E}$) succeeds, returning an $\mathbf{J}$ such that $\vdash_{\text{LPnd}} \mathbf{J}\theta$.

Proof. A straightforward consequence of the Unification Correctness theorem and the previous lemma.

□

As it was with equality environments, we must now prove that there is a satisfying substitution for inequality environments; since we will only ever be

working with normal form inequality environments, it suffices to prove that they are satisfiable.

Theorem 6.4 (Inequality Environment Satisfiability) For all normal form inequality environments $\mathbf{J}$, there is a closed-term substitution σ mentioning only the variables of $\mathbf{J}$ such that σ satisfies $\mathbf{J}$, i.e. such that $\vdash_{\mathrm{LPnd}} \mathbf{J}\sigma$.

Proof. By induction on the number n of uninstantiated variables in the equality environment $\mathbf{E}$ in $\mathbf{J}$ (variables $\mathbf{x}$ such that $\mathbf{x} = \mathbf{x}$ appears in $\mathbf{E}$).

Case $n = 0$. Since every inequality in an inequality environment must have an uninstantiated variable on one side, $\mathbf{J}$ cannot have any inequality lists; so it is just the equality environment, $\mathbf{E}$. This has a satisfying substitution by the equality environment satisfiability theorem.

Case $n > 0$. If the equality environment in $\mathbf{J}$ contains some uninstantiated variables, then choose one such uninstantiated variable, $\mathbf{y}$. Let $\mathbf{g}$ be a function symbol in $\mathcal{L}$ of degree greater than 0 (recall that we have assumed there is at least one such function symbol), and let the term schema $\mathbf{s}(\mathbf{r})$ stand for the term $\mathbf{g}(\mathbf{r}, 0, \ldots, 0)$. Now let $\mathbf{t}$ be the first term in the sequence $0, \mathbf{s}(0), \mathbf{s}(\mathbf{s}(0)), \ldots$ such that the inequality $\mathbf{y} \neq \mathbf{t}$ does not appear in $\mathbf{J}$.

Now let $\mathbf{J}'$ be the result of Subsume($(\mathbf{y} = \mathbf{t})$, $\mathbf{J}$). (Subsume succeeds in this case because the unification succeeds, and after dereferencing no inequality is of the form $\mathbf{s} \neq \mathbf{s}$; so at least one inequality must remain in every inequality list at the end of the algorithm.) From the induction hypothesis, $\mathbf{J}'$ has a satisfying substitution; and from the subsumption correctness theorem, this substitution will also satisfy $\mathbf{J}$.

□

As I mentioned above, all the proofs of equivalence of LPi and LPndi now follow through in much the same manner as the proofs of equivalence of LP and LPnd. The subsumption correctness and inequality environment satisfiability theorems appear at the base of the inductions for LPi/LPndi, as the unification correctness and equality environment satisfiability theorems did for LP/LPnd. The central characterisation result for the system with inequality takes the following form (compare Corollary 4.7):

Corollary 6.5 (Characterisation) For all formulae $\mathbf{A}$, $\mathbf{E}_0$: $\exists[\mathbf{A}] \overset{\mathrm{LPi}}{\Rightarrow} \mathbf{F}$ for some $\mathbf{F}$ iff $\vdash_{\mathrm{LPndi}} \exists[\mathbf{A}]$.

□

The system LPi/LPndi thus has the same pleasing properties as LP/LPnd: a useful, descriptive operational semantics in the form of a formal system, and a logical, intuitive characterising proof system.

6.4 Related Work

I have already mentioned the relation this work has to artificial intelligence constraint satisfaction problems. Colmerauer's system Prolog II [CKvC83] appears to have been the first to implement some soundly-based and complete notion of inequality. Prolog II, however, handles inequality over infinite terms; here we deal with inequality over terms of finite size. Neither system seems superior to the other for all applications. For instance, infinite terms are convenient for simulating automata, whereas for programs for logical systems we would often want to *disallow* such things as infinite proofs.

The Constraint Logic Programming, or CLP, system [JL86] subsumes systems like LP and LPi, as well as other systems of logic programming over domains like Presburger arithmetic and linear programming. For instance, CLP(R) is a system in which linear programming problems can be expressed and completely decided.

The theory behind CLP is considerably more complex than that involved in equality logic programming. It would therefore be interesting to study whether the entire CLP framework can be simplified by being cast into proof-theoretic terms, or if its generality demands a more abstract, set-theoretic characterisation.

7 Discussion

I feel that the characterisation of logic programming given in this paper is less conceptually complex than the standard one. This reduction in complexity is accompanied by a loss of generality in some areas, and a gain of generality in others; but I feel that, on the whole, the present characterisation is at least a useful alternative to the standard one. One's judgment of such things obviously involves taste, philosophy, and academic background. I will present the facts in this matter, and some of my personal beliefs, but will leave the final judgment to the readers.

Here is an outline of the salient concepts involved in the standard characterisation of simple, negationless logic programming with first order equality. It is based on Lloyd's standard work [Llo84].

- Declarative semantics.
  - Logic. First order theories.
  - Horn clauses. Programs. Goal clauses.
  - Interpretations. Models. Herbrand universes, bases, models. Least Herbrand model.
  - Substitutions. Most general unifiers. Unification. Termination and correctness of unification algorithm.
- Fixpoint semantics.

- Lattices. Monotonic and continuous mappings. The Knaster-Tarski theorem. Ordinal powers of a mapping.
- The T_P mapping. Fixpoint characterisation of least Herbrand model.

- Procedural semantics.
  - Computation rules. SLD-resolution. Success sets. Soundness of SLD-resolution.
  - Completeness of SLD-resolution.
  - SLD-trees. Search rules and SLD-refutation procedures. Fair search rules.

Here is an outline, which I hope readers will agree has the same level of detail, of the salient concepts involved in the characterisation given in this paper. Note in particular the presence of environments, and the absence of general resolution and fixpoint semantics.

- Basic definitions.
  - Logic. First order theories.
  - Program formulae. Predicate definitions. Programs.
  - Substitutions. Environments. Unification. Termination of unification algorithm.
- Operational semantics: the formal system LP.
- Characterising proof system: the formal system LPnd.
  - Correctness of unification.
  - Soundness of LP with respect to LPnd. Completeness of LP. Characterisation.

The standard characterisation is more general than the one given here in that it gives a general description of logic programming interpreters. Individual interpreters (which may or may not be complete) can be described by specifying a computation rule and a search rule. In the present characterisation, a new operational semantics must be presented to specify a new interpreter.

The present characterisation is somewhat more general than the standard one in that the use of unification over environments, rather than substitutions, makes the structure of the systems and the theorems about them easily adaptable to other constraint logic programming theories.

My feeling is that the standard characterisation contains many elements which are present only for historical reasons. Logic programming arose as a special case of resolution theorem proving, but it has developed into a separate programming

paradigm which I feel can justify an independent characterisation. From the point of view of the present characterisation, resolution can be seen as a formal system halfway between an operational semantics and a proof system – with insufficient operational details to be the former, and insufficient intuitiveness to be the latter.

I also find it puzzling that fixpoint semantics (the standard semantic framework for *functional* programming) has been deemed necessary as a "bridge" to show the equivalence of the declarative and procedural semantics from the time of the first papers on logic programming semantics (such as [vEK76]). As this paper clearly shows, no such bridge is necessary.

I leave it to the readers to judge whether the present characterisation is truly less conceptually complex than the standard one; whether the accompanying loss of generality is justifiable; and whether they agree with my assessment of the historical nature of the standard characterisation.

8 Conclusions and Future Work

In this paper, I have presented a simple, intuitive proof-theoretic characterisation of a particular logic programming system with parallel or. I have then extended it to a sequent calculus proving properties of programs (including inductive properties); and I have demonstrated its potential for generalisation to constraint logic programming languages by extending it to a system which handles inequality as well as equality. I have also argued that such proof-theoretic characterisations are less conceptually complex than the standard ones, without being significantly less general.

In future work, I am particularly interested in developing a similar characterisation of sequential-or logic programming. This task is made difficult by the non-compositional nature of sequential or: the termination of a query depends not only on the termination of the subformulae of the query, but also on the interaction between the subformulae.

Because a characterisation of sequential or will involve a characterisation of failure and backtracking, I hope that this work will lead to a proof-theoretic characterisation of some form of negation, such as negation as failure or negation as inconsistency.

It may be interesting to see how the framework given here can be formally generalised to other constraint languages. One goal might be to develop a formal framework of which LP and LPi are just instances, as well as logic programming systems deciding Presburger arithmetic, linear arithmetic, real arithmetic and so on. This work should clearly try to achieve the generality of the CLP system [JL86], the standard semantic framework for constraint logic programming.

9 Acknowledgements

Thanks to my advisor, Don Sannella, for many useful discussions on this work. Thanks also for helpful suggestions to Ruth Davis, James Harland, Bob Harper, and the referees of the MFCS '89 version of this paper.

Special thanks to Paul Voda for my initial inspiration. Many of the ideas contained here are just formal elaborations of his thoughts.

References

[And89] James H. Andrews. Proof-theoretic characterisations of logic programming. In *Mathematical Foundations of Computer Science*, Rytro, Poland, August 1989.

[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.

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

[JL86] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. Technical report, Department of Computer Science, Monash University, June 1986.

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

[Mac85] Alan Mackworth. Constraint satisfaction. Technical Report 85-15, Department of Computer Science, University of British Columbia, September 1985.

[ML71] Per Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium.* North-Holland, 1971.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, September 1981.

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

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