A logical semantics for depth-first Prolog
with ground negation

James Andrews *

Department of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6

Received May 1994; revised January 1996
Communicated by G. Levi

Abstract

A sound and complete semantics is given for sequential, depth-first logic programming with
a version of negation as failure. The semantics is logical in the sense that it is built up only
from valuation functions (multi-valued logic interpretations in the style of Fitting and Kunen)
and logically-motivated equivalence relations between formulas. The notion of predicate folding
and unfolding with respect to a program (Tamaki, Sato, Levi et al.) and the universal notion of
"disjunctive unfolding" (Andrews) are important elements of this semantics.

The negation used is the version which returns an error indication whenever it is invoked
on a non-ground goal. It is theoretically interesting that this form of negation, along with the
left-to-right processing of depth-first logic programming, can be characterized logically with four-
valued interpretations over an extended alphabet of terms. The fourth truth value, N, can be read
operationally as "floundering on negation". The extension of the alphabet provides the semantics
with a logical analogue of free variables. This intriguing technique may open the door to the
characterization of other forms of practical negation, or of other language features involving
groundness conditions.

This material was published earlier as a technical report [4] and is the expanded version,
including proofs, of a paper presented at the 1993 International Logic Programming Symposium
(ILPS) [5].

1. Introduction

Sometimes it is possible to take a seemingly non-logical feature of a logic program-
mieing system, and give it a more logical interpretation by extending our traditional
notions of semantics. In [1, 2] it was shown how this can be done for the depth-first,
left-to-right strategy of sequential Prolog. In this paper, those results are made more
abstract, and extended to a system having a form of negation as failure. The main
new elements added are a fourth truth value (to go with true, false and undefined) and a set of special constants which act as a logical analogue of free variables.

This paper also contains the clearest description so far of a style of semantics which could be called "congruential", which was latent in the proof-theoretic descriptions of [1, 2] and was developed further in [3]. The general method behind this semantics is to characterize a restricted class of query/program pairs with a compositional, logical valuation function, and then to characterize wider classes of queries and programs with the use of congruences (syntactically compositional equivalence relations) between formulas.

A pleasant feature of congruential semantics is the economy of mathematical notions it uses: it is built up almost exclusively from notions of term, formula, truth value, valuation, and congruence between formulas.

1.1. Handling of negation

The ideal situation in logic programming would be to have a highly logical declarative semantics of programs, and a practical operational semantics, with a powerful negation, which is sound and complete with respect to it. However, it is generally agreed that we must compromise on some of these requirements to stay within the logic programming paradigm. The compromise taken here is to use a relatively weak negation, but to try to maintain the rest of the ideal.

The particular form of negation considered in this paper is what could be called "insist-on-ground negation as failure": it computes a goal \( \neg G \) by the regular negation-as-failure (NAF) method only if \( G \) is ground, and terminates query processing immediately with an error indication if \( G \) is not ground. This is preferable to the unsound negation of most Prologs for some applications, but Naish points out [39] that even the slightly more liberal form of NAF used in IC-Prolog gives the error indication frequently. More useful is the form of NAF in which the call to \( \neg G \) is delayed until \( G \) is ground [22,38].

However, it is theoretically interesting that this form of negation can be given a logical semantics based on a four-valued logic. The four truth values are \( T \) (true), \( F \) (false), \( U \) (undefined), and \( N \) ("floundering on negation", needed to help characterize the other truth values). They are arranged in the following total order to give a semantics to the existential quantifier:

\[
N \\
| \\
T \\
| \\
U \\
| \\
F
\]
The ordering is not on "degree of truth" but rather on "degree of computational priority"; so the four truth values do not have the same structure as in, for instance, Belnap's four-valued logic [15].

Assisting in the logical characterization of the negation are an infinite number of "special constants". These constants act as logical stand-ins for unbound variables in the truth-functional semantics, allowing a logical interpretation of the groundness condition. Essentially, if a query flounders, then the special constants allow for some "ground" instance of it to flounder as well. The operational semantics is extended to take account of the special constants for theoretical purposes, but they are not needed in the usual query evaluation process.

The operational semantics within which the negation is situated is sound and complete with respect to the declarative, and uses the usual top-down, depth-first search strategy of most Prolog interpreters. This makes it somewhat more practical for general-purpose logic programming than proposals for negation which need a bottom-up and/or breadth-first strategy for completeness. Floundering can be avoided in practice by, for instance, requiring that variables inside negations be input-mode. However, note that the semantics given here is complete for all queries and programs, and thus in particular any restricted subset of queries and programs.

1.2. Unfoldings and congruences

In some sense, all the big problems in logic programming semantics come from the presence of recursive predicate definitions. The approach of this paper to handling this aspect of the semantics is based on the following observation:

If goal $G$ succeeds (fails) with respect to a program $P$, then there is some unfolding of $G$ which is true (false) in the Clark equality theory CET [21].

Usually we speak of the unfolding of a program [45], but when we generalize the notion of goal to include equalities (interpreted consistent with CET) and connectives, we can unfold goals as well. The positive part of the Clark completion [21] of $P$ is read as the specification of a fold/unfold congruence $\equiv_p$ between formulas (rendering irrelevant its falsehood or inconsistency as a set of formulas). We can thus use $\equiv_p$ and the semantics of goals with respect to the empty program to define the semantics of goals with respect to $P$.

**Example.** Let $P$ consist of the two clauses "even(0)." and "even(s(s(x))) ← even(x)." Then the positive completion of $P$, compl($P$), is

$$\text{even}(y) ← (y = 0) ∨ \exists x(y = s(s(x)) \& \text{even}(x))$$

Therefore the first unfolding of $\neg\text{even}(s(0))$ is

$$\neg((s(0) = 0) ∨ \exists x(s(0) = s(s(x)) \& \text{even}(x)))$$

which is true even with respect to the empty program.
These ideas are closely related to those used by Levi et al. [34, 28], who define the "unfolding semantics" of a program via the limit of the process of unfolding the program. These ideas allow us to sidestep the problem of asymmetry of success and failure in the least-model and fixpoint semantics by viewing the process of unfolding as central.

1.3. Organization of this paper

This paper is organized as follows. Section 2 presents some basic definitions. Section 3 gives the operational semantics which will act as a reference point for the declarative semantics. Section 4 gives the declarative semantics, in three parts: the semantics of a restricted class of goal formulas with respect to the empty program, the semantics of general goals with respect to the empty program, and the semantics of general goals with respect to general programs. Section 4 also gives proofs of relative soundness and completeness of the two semantics. Section 5 discusses various issues, such as the logicalness of the semantics and the characterization of variant forms of negation. Section 6 discusses this work in the context of previous work, and Section 7 gives conclusions and suggestions for future work.

2. Basic definitions

Definition 2.1. A first-order language \( \mathcal{L} \) consists of a countably infinite set \( \mathcal{V}(\mathcal{L}) \) of variable names, a finite set \( \mathcal{F}(\mathcal{L}) \) of function symbols \( f_i \) each with an associated arity \( n_i \geq 0 \), and a set \( \mathcal{P}(\mathcal{L}) \) of predicate names \( p_j \) each with an associated arity \( m_j \geq 0 \).

In the sequel we will assume the existence of some fixed first-order language \( \mathcal{L} \) with equality = as the language of all programs. We will also assume the existence of a countably infinite set \( \mathcal{S} \) of "special constants". Terms in \( \mathcal{L} \) and \( \mathcal{S} \) are built up from the variables and special constants in the usual way with the function symbols. We will use \( x, y \) as meta-variables standing for variables, \( p \) for predicates, and \( s, t \) for terms, all possibly subscripted.

In order to express the logical connectives more clearly, we will work with a syntactic generalization of goals and programs, in the style of Miller et al. [37]. It should be clear that this is no loss or gain of power over the usual clausal form.

Definition 2.2. A goal formula \( G \) of a language \( \mathcal{L} \) is a formula built up from the following BNF syntax:

\[
G ::= (s = t) \mid p(t_1, \ldots, t_n) \mid G \& G \mid G \lor G \mid \exists xG \mid \neg G
\]

A formula is ordinary if it contains no special constants.

A clause (defining \( p \)) in a language \( \mathcal{L} \) is a formula of the form \( p(t_1, \ldots, t_n) \leftarrow G \), where \( G \) is ordinary, or of the form \( p(t_1, \ldots, t_n) \).
A *program* is a finite sequence of clauses.

We will use $G, A, B, C$ as meta-variables standing for goal formulas, and $P$ for programs, all possibly subscripted.

We will not be concerned so much with the original form of a program as with its *positive Clark completion* [21].

**Definition 2.3.** Let $C$ be a clause in program $P$. Let $cc(C, P)$ be defined as:

1. $\exists y_1 \ldots y_n(x_1 = t_1, \ldots, x_n = t_n) \land G$, if $C$ is of the form $p(t_1, \ldots, t_n) \leftarrow G$, and
2. $\exists y_1 \ldots y_n(x_1 = t_1, \ldots, x_n = t_n)$, if $C$ is of the form $p(t_1, \ldots, t_n)$

where $y_1, \ldots, y_n$ are all free variables of $C$, and $x_1, \ldots, x_n$ are the first $n$ variables not appearing in $P$.

Let $p$ be a predicate name and $P$ be a program containing clauses defining $p$. Let $cd(p, P)$ be the expression $p(x_1, \ldots, x_n) \leftrightarrow cc(C_1) \lor \cdots \lor cc(C_m)$, where $C_1, \ldots, C_m$ is the sequence of clauses in $P$ defining $p$, and $x_1, \ldots, x_n$ are the first $n$ variables not appearing in $P$.

Then the *positive Clark completion* of $P$, $compl(P)$, is the set $\{cd(p_1, P), \ldots, cd(p_k, P)\}$, where $p_1, \ldots, p_k$ are all the predicates defined in $P$.

Note that for simplicity we assume all predicates are defined with the same arity in $P$. Also, while the original Clark completion does not refer to the *sequence* of clauses in $P$, here we modify the definition slightly to insist that the sequence of clauses in $P$ be preserved in the sequence of $cc$ expressions in $cd(p, P)$.

**Definition 2.4.** A *equivalence relation* $\equiv$ is a binary relation which is reflexive ($A \equiv A$), symmetric (if $A \equiv B$ then $B \equiv A$), and transitive (if $A \equiv B$ and $B \equiv C$ then $A \equiv C$).

A *congruence* (between goal formulas) is an equivalence relation $\equiv$ such that

1. If $B \equiv B'$ and $C \equiv C'$, then $(B \land C) \equiv (B' \land C')$;
2. If $B \equiv B'$ and $C \equiv C'$, then $(B \lor C) \equiv (B' \lor C')$;
3. If $B \equiv B'$, then $\exists x B \equiv \exists x B'$; and
4. If $B \equiv B'$, then $\neg B \equiv \neg B'$.

One important class of congruences is the following. It makes use of the notion of predicate folding and unfolding introduced by Tamaki and Sato [45], based on the corresponding functional notion of Burstall and Darlington [20].

**Definition 2.5.** The *fold/unfold congruence* associated with a program $P$, $\equiv_P$, is the least congruence such that for every predicate definition $p(x_1, \ldots, x_n) \leftarrow G$ in $compl(P)$, we have that

$$p(t_1, \ldots, t_n) \equiv_P G[x_1 := t_1, \ldots, x_n := t_n]$$

Informally, $A \equiv_P A'$ holds if $A'$ can be obtained from $A$ by any number of fold or unfold transformations (including zero). The intersection of any collection of
congruences is a congruence, so the intersection of all such congruences is the least such congruence.

3. Operational semantics

To specify precisely what kind of logic programming system we are talking about, I give here a formal operational semantics SPN (Sequential Prolog with Negation) for the system. SPN is basically SLD-resolution for our general goals and programs, streamlined for the desired left-to-right and negation strategies.

SPN is actually an abstract interpreter, since the only observable it characterizes is which of four general outcomes results from the computation of a goal:

- \( T \), meaning that the computation returns at least one answer substitution;
- \( F \), meaning that the computation fails finitely;
- \( U \), meaning that the computation has floundered upon trying to call a non-existent predicate (which is closely related to divergence, as we will see); and
- \( N \), meaning that the computation has floundered on negation on a non-ground subgoal.

This form of operational semantics is the most convenient for the purposes of proving equivalence with the truth-functional semantics. It omits mention of the returned answer substitution, and does not characterize the finding of more than one substitution; however, it could easily be modified to characterize these observables.

3.1. SPN derivations as proofs

To explain computations in SPN, it is perhaps more convenient to look on them first as proofs of statements, in analogy with formal mathematical proofs.

Like a formal proof, a "computation" of SPN is a tree of "judgements", written root-down, with only certain allowed configurations of child nodes (premisses) to parent node (conclusion). (The use of formal systems to express operational semantics was first studied extensively by Plotkin [41].) The allowed configurations of premisses and conclusion are given by proof rules; SPN's are given in Fig. 1. A particular judgement \( J \) is considered to be proven if there is a tree, built according to the rules, with \( J \) at the bottom and only axioms (zero-premiss rules) at the top.

In SPN, judgements are expressions of the form \((\theta : \alpha) \overset{P}{\rightarrow} \sigma\), where \( \theta \) is a substitution (the current variable binding environment), \( \alpha \) is a sequence of goal formulas (the goal stack, or sequence of goals yet to be processed), \( P \) is a program (the current program under consideration), and \( \sigma \) is a truth value (the result of the computation). \((\theta : \alpha)\) is called the closure, in analogy with the corresponding notion in functional language interpreters. The current program \( P \) is often omitted, where it can be assumed.

For example, let \( P \) be the program consisting of the single clause \( p(2) \). The positive Clark completion of this program is

\[
p(x) \leftrightarrow x = 2
\]
\((\theta : B, C, \alpha) \Rightarrow \sigma\) \\
\(\vdash (\theta : B \land C, \alpha) \Rightarrow \sigma\)

\(\forall 1:\) \\
\(\vdash (\theta : B, \alpha) \Rightarrow \sigma\) \\
\(\vdash (\theta : B \lor C, \alpha) \Rightarrow \sigma\)

where \(\sigma\) is \(T, U,\) or \(N\)

\(=1:\) \\
\(\vdash (\theta : B, \alpha) \Rightarrow \sigma\) \\
\(\vdash (\theta : s = t, \alpha) \Rightarrow \sigma\)

where \(\theta'\) is an mgu of \(s\theta\) and \(t\theta\)

\(=2:\) \\
\(\vdash (\theta : s = t, \alpha) \Rightarrow F\)

where \(s\) and \(t\) have no unifiers

\(\forall 2:\) \\
\(\vdash (\theta : B, \alpha) \Rightarrow F\) \\
\(\vdash (\theta : B \lor C, \alpha) \Rightarrow \sigma\)

\(\forall 3:\) \\
\(\vdash (\theta : B, \alpha) \Rightarrow \sigma\) \\
\(\vdash (\theta : 3x(B, \alpha) \Rightarrow \sigma)\)

where \(x'\) does not appear below the line

Fig. 1. Operational semantics SPN. Special constants are mentioned to aid proofs of theorems, but do not appear in ordinary computations.

\(\vdash (\theta : e) \Rightarrow T\)

\(\vdash (y := 2) : e \Rightarrow T\)

\(\vdash (y := 2) : y = 2 \Rightarrow T\)

\(\vdash (y := 2) : p(y) \Rightarrow T\)

\(\vdash (y := 3) : p(y) \Rightarrow F\) \\
\(\vdash (y := 3) : -p(y) \Rightarrow T\)

\(\vdash (y := 3) : y = 2 \lor y = 3 \Rightarrow T\)

\(\vdash (y := 3) : y = 2 \Rightarrow F\)

\(\vdash (x := 2 \lor x = 3) : \neg p(x) \Rightarrow T\)

\(\vdash (x := 2 \lor x = 3) : \neg p(x) \Rightarrow T\)

Fig. 2. A simple derivation in SPN.

A proof of the judgement \(((\_): \exists x((x = 2 \lor x = 3) \& \neg p(x))) \Rightarrow T\) is contained in Fig. 2. The computation shows that the query \(\exists x((x = 2 \lor x = 3) \& \neg p(x))\), when computed in the context of the empty substitution \((\_))\), results in the truth value \(T\); that is, the operational semantics considers the query to be true. The left and right-hand columns show the rules which are being applied to form each step of the derivation.
3.2. SPN derivations as computations

Judgements in SPN can also be seen as describing the computation steps followed by an interpreter computing a list of subgoals. To compute a query \( G \) relative to program \( P \), we let \( \theta \) be the empty substitution and \( \alpha \) be the sequence consisting of the single formula \( G \), and we try to find a \( \sigma \) such that \( ((\cdot) : G) \overset{P}{\Rightarrow} \sigma \) is provable.

In our example, if we were trying to find a \( \sigma \) such that \( ((\cdot) : \exists x((x = 2 \lor x = 3) \land \neg p(x))) \overset{P}{\Rightarrow} \sigma \), we could tell that the rule at the bottom of the computation would have to be \( \exists \). The form of that rule completely determines the form of the substitution and goal stack in its premiss (modulo the unimportant choice of variable name), so we can reduce the problem of finding \( \sigma \) to that of finding the outcome of \( ((\cdot) : y = 2 \lor y = 3, \neg p(y)) \). At this point, we have a choice of rules (\( \lor_1 \) or \( \lor_2 \)); but the choice depends only on the outcome of \( ((\cdot) : y = 2, \neg p(y)) \). Therefore we can reduce the problem to finding that outcome, knowing that we may have to do another subsidiary computation afterwards.

This general pattern of finding the outcomes to subsidiary computations is followed throughout the computation. In essence, we proceed around the entire outline of the proof, going up the left side and then down the right side of branches, until we get back to the root (see Fig. 3). Of course, not all computations terminate; for those that do not, no complete proofs exist, only fragmentary ones whose topmost judgements are not axioms. This corresponds to the situation of keeping doing subsidiary computations forever, without ever finding their outcomes.

4. Truth-functional semantics

In this Section, I describe the semantics which is the main contribution of this paper. The semantics uses the two logical notions of valuation and congruence. These interact to produce a precise characterization that neither notion alone can achieve.

Consider the notion of valuation (function mapping formulas to truth values). A valuation \( v \) is a highly logical construct if it can be defined compositionally – for...
instance, by defining $v(B \& C) = T$ only when $v(B) = v(C) = T$. But with the left-to-right behaviour of our operational semantics, it is possible to give a compositional valuation only with subsets of the set of formulas. For example, assume that loop is a predicate which goes into infinite recursion on any argument. If we wanted a $v$ such that $v(G)$ accurately reflects the operational outcome of the goal $G$, $v$ would have to have the behaviour shown in Table 1.

The formulas $2 = 2$ and $(2 = 2 \lor \text{loop}(3))$ have exactly the same truth value under this valuation; but when they appear as the left-hand half of a conjunction whose right-hand half is $2 = 3$, they yield different truth values. Thus a faithful $v$ cannot be truly compositional. One way to make $v$ more compositional would be to map goals onto denotations that were more complex than just truth values; but that would make the semantics less logical and more functional than if we used valuations.

It is also possible to identify certain logical and operational equivalences between formulas: for instance, whenever we replace any subformula $(B_1 \lor B_2) \& C$ by $(B_1 \& C) \lor (B_2 \& C)$, we get the same result, even in left-to-right search. But we cannot build up a semantics based on such congruences alone, since we would like to define the behaviour of $\exists x B$ in the traditional logical fashion, in terms of the behaviour of all its instances $B[x := t]$.

As this paper shows, however, if we use an interacting collection of valuations and congruences, we can preserve much of the compositionality and logical flavour that each construct gives. The approach to the present problem is in three stages. In the first, we give a valuation on a subclass of goal formulas which characterizes their behaviour with respect to the empty program $\emptyset$. In the second, we define a congruence which relates all other goal formulas to this subclass, still with respect to $\emptyset$. Finally, we show that the fold/unfold congruence $\congruence P$ associated with a program $P$ allows us to characterize the behaviour of all goals with respect to $P$.

4.1. First stage: outer-disjunction queries, empty program

In this section, we define a subset of the goal formulas called "outer-disjunctive" or "O" formulas. We then give a valuation function $v$ from O formulas to truth values, and prove that this characterizes precisely the behaviour of O formulas with respect to the empty program $\emptyset$.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$C$</th>
<th>$v(B)$</th>
<th>$v(C)$</th>
<th>$v(B &amp; C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 = 2$</td>
<td>$2 = 3$</td>
<td>$T$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$2 = 2 \lor \text{loop}(3)$</td>
<td>$2 = 3$</td>
<td>$T$</td>
<td>$F$</td>
<td>$U$</td>
</tr>
</tbody>
</table>
4.1.1. Definitions

Definition 4.1 (N and O formulas). The sets of N (negated-disjunction) formulas and O (outer-disjunction) formulas are defined mutually recursively as follows:

\[
N ::= (t_1 = f_2) \mid p(t_1, \ldots, t_n) \mid N_1 \& N_2 \mid \exists x N \mid \neg O
\]

\[
O ::= N \mid O_1 \lor O_2
\]

Note that the O formulas are the goal formulas in which all interior disjunctions are the immediate subformulas of either negations or other disjunctions. The N formulas are the O formulas whose top-level connective is not a disjunction.

For the purposes of defining the truth valuation on O formulas, it is convenient to define an ordering on truth values.

Definition 4.2. The ordering \( \leq_t \) on truth values is defined by \( F \leq_t U \leq_t T \leq_t N \).

As in previous work on truth-functional semantics [26, 33], the truth value of an existentially quantified formula \( \exists x B \) will be the maximum, under this ordering, of the values of its instances \( B[x := t] \). This is reflected in the definition of \( v \), the valuation on O formulas.

Definition 4.3. \( v \), a valuation function mapping ground, outer-disjunction (O) formulas to truth values, is defined as follows:

- \( v(t = t) = T \);
- \( v(s = t) = F \), where \( s \) is not identical to \( t \);
- \( v(p(t_1, \ldots, t_n)) = U \);
- \( v(B \& C) = \begin{cases} v(C) & \text{if } v(B) = T, \\ v(B) & \text{otherwise}; \end{cases} \)
- \( v(B \lor C) = \begin{cases} v(C) & \text{if } v(B) = F, \\ v(B) & \text{otherwise}; \end{cases} \)
- \( v(\exists x B) = \max_{\exists}(\{B[x := t] | t \text{ ground}\}); \)
- \( v(\neg B) = \begin{cases} F & \text{if } B \text{ ordinary and } v(B) = T, \\ U & \text{if } B \text{ ordinary and } v(B) = U, \\ T & \text{if } B \text{ ordinary and } v(B) = F, \\ N & \text{otherwise}. \end{cases} \)

(Recall that a formula is ordinary if it contains no special constants.) Note the compositional, logical nature of \( v \), in particular the treatment of \( \exists x B \) in terms of the instances of \( B \).

4.1.2. Examples

The treatment of insist-on-ground negation is perhaps the most subtle aspect of the valuation \( v \). The key is the way in which an existential formula is assigned a truth value which is the maximum, under the \( \leq_t \) ordering, of the truth values of its instances. An
Table 2

<table>
<thead>
<tr>
<th>B</th>
<th>(\nu(B[x := 2]))</th>
<th>(\nu(B[x := 3]))</th>
<th>(\nu(B[x := k]))</th>
<th>(\nu(\exists x B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 2)</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>(\neg x = 2)</td>
<td>F</td>
<td>T</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(x = x)</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>(\neg x = x)</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(x = 2 &amp; \neg x = 3)</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>(x = 2 &amp; \neg x = 2)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>(x = x &amp; \neg x = 2)</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 3

<table>
<thead>
<tr>
<th>B</th>
<th>(\nu(B[x := 2, y := 2]))</th>
<th>(\nu(B[x := k, y := j]))</th>
<th>(\nu(B[x := k, y := k]))</th>
<th>(\nu(\exists x \exists y B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg x = y)</td>
<td>F</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(x = y &amp; \neg x = y)</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(x = y &amp; x = 2 &amp; \neg x = y)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>(x = y &amp; x = 2 &amp; y = 3)</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

eXistent formula receives an \(N\) truth value (that is, \(\nu\) considers the query to flounder on negation) if and only if at least one of its instances receives \(N\).

Consider Table 2. \(k\) is a special constant.

Consider \(\neg x = 2\) and \(\neg x = x\). For both of these formulas, one instance returns the \(N\) sign, which (because of its position at the top of the order of truth values) makes the existential closure return \(N\) as well.

Now consider the \(k\) (special constant) instance of the formula \((x = 2 \& \neg x = 3)\). We still have that \(\nu(\neg k = 3) = N\); but \(\nu(k = 2) = F\), and because of the behaviour of \(\nu\) on \(\&\), the truth value of the instance is \(F\). Essentially, the early, failing subformula "protects" the truth value of the entire formula from being pushed upwards by the \(N\), because of the asymmetric treatment of \(\&\) by \(\nu\).

The only time that the truth value of the entire \(k\) instance will not be protected from an \(N\) subformula is when either the variable in question does not participate in any equations, or else is unified only with itself or with other uninstantiated variables. This is because equations like \(k = k\) (a \(k\) instance of \(x = x\)) receive a \(T\) truth value, causing \(\nu\) to get its final value from the rest of the formula. Thus we have that \(\nu(\exists x (x = x \& \neg x = 2)) = N\).

Note that unification with another, uninstaiated variable also works as we would like. The uninstaiated variable still raises the error condition, as it would in the operational semantics; however, if the uninstaiated variable or anything it is unified with is later instantiated, the error condition is not raised. Here is another, similar table (Table 3), where \(j\) and \(k\) are special constants.

Readers might want to try writing a short Prolog program to confirm the truth values under \(\nu\) of the examples in this section.
4.1.3. Properties

The operational semantics SPN is sound and complete with respect to the valuation \( v \), when the goal is an \( \mathbf{O} \) formula and the program is empty. We will prove this by a technique which involves looking only at ground queries.

There are four technical lemmas we will need before we proceed. I will just state them here, and prove them in Appendix A. The statements of the lemmas all have to do with “logical” behaviour in SPN computations, such as the behaviour of substitutions and conjunctions. First is a lemma that states that we can add irrelevant information to the substitution without affecting the result.

**Lemma 4.4 (Substitution Lemma).** Let \( (\theta : \alpha) \xrightarrow{p} \sigma \). Then for every \( \theta' \) such that \( x\theta = x\theta' \) for every free variable \( x \) of \( \alpha \), \( (\theta' : \alpha) \xrightarrow{p} \sigma \).

Next is a lemma which is at the heart of the approach to the proofs used here. The basic result is this. Say a negated-disjunction formula \( \mathbf{B} \) has a free variable \( x \), and the computation of \( \mathbf{B} \) as a goal results in \( \sigma \). Then \( \sigma \) is exactly the maximum, under the \( \preceq \), ordering, of the outcomes of the closed instances of \( \mathbf{B} \). To prove this, we need to generalize to many variables so that we can do an inductive proof.

**Lemma 4.5 (Existential Lemma).** Let \( \alpha \) be a sequence of negated-disjunction (N) formulas, such that \( (\theta : \alpha) \xrightarrow{\emptyset} \sigma \). Let \( V \) be a subset of the free variables of \( \alpha \theta \). Define a substitution \( \rho \) as acceptable if \( \rho \) does not assign a value to any variable not in \( V \), but \( \theta \rho \) grounds all variables in \( V \). Define \( \sigma'(\rho) \) as the function such that \( (\theta \rho : \alpha) \xrightarrow{\emptyset} \sigma'(\rho) \).

**EX1.** For some acceptable \( \rho \), \( \sigma'(\rho) = \sigma \).

**EX2.** If \( \sigma = T \), then for all acceptable \( \rho \), \( \sigma'(\rho) \in \{T,U,F\} \).

**EX3.** If \( \sigma = U \), then for all acceptable \( \rho \), \( \sigma'(\rho) \in \{U,F\} \).

**EX4.** If \( \sigma = F \), then for all acceptable \( \rho \), \( \sigma'(\rho) = F \).

Next come two lemmas which will allow us to reason separately about the two conjunctions in a conjunction. The first concerns success, and the second, all other outcomes.

**Lemma 4.6 (Conjunction, Left Success).** Let \( \alpha' \) be a sequence of negated-disjunction (N) formulas, such that \( (\theta : \alpha') \xrightarrow{\emptyset} T \). Then \( (\theta : \alpha', \alpha) \xrightarrow{\emptyset} \sigma \) iff there is some \( \theta' \) such that:

1. At the top right corner of the derivation of \( (\theta : \alpha') \) is the judgement \( (\theta' : \varepsilon) \xrightarrow{\emptyset} T \); and

2. \( (\theta' : \alpha) \xrightarrow{\emptyset} \sigma \).

**Lemma 4.7 (Conjunction, Left Non-Success).** If \( (\theta : \alpha') \xrightarrow{\emptyset} \sigma \), where \( \sigma \neq T \), then for all \( \alpha \), \( (\theta : \alpha', \alpha) \xrightarrow{\emptyset} \sigma \).

Using these lemmas, we can proceed with the proof of characterization. We will prove the two directions separately.
Theorem 4.8 (Soundness of SPN wrt \( u \)). For all ground, outer-disjunction (O) formulas \( G \), if \( \vdash (\,\cdot\, : G) \DownRightarrow \sigma \), then \( u(G) = \sigma \).

**Proof.** By induction on the size of \( G \). Cases are on the form of \( G \).

- **Case B & C:** There are two subcases.
  
  If \( \vdash (\,\cdot\, : B) \Rightarrow T \), then by the Conjunction Left Success lemma, there is some \( \theta' \) such that \( (\theta' : C) \Rightarrow \sigma \). But then, since \( C \) is ground and the choice of substitution cannot affect its outcome, we also have that \( (\,\cdot\, : C) \Rightarrow \sigma \). By the induction hypothesis, therefore, \( u(B) = T \) and \( u(C) = \sigma \); but then by the definition of \( u \), \( u(B \& C) = \sigma \).

  If \( \vdash (\,\cdot\, : B) \Rightarrow \sigma \), where \( \sigma \neq T \), then by the Conjunction Left Non-Success lemma, \( (\,\cdot\, : B, C) \Rightarrow \sigma \) as well. By the induction hypothesis, \( u(B) = \sigma \); but by the definition of \( u \), \( u(B \& C) = \sigma \) as well.

- **Case \( \exists x B \):** Because we have assumed that \( G \) is outer-disjunctive, this \( G \) must be a negated-disjunction (N) formula. We can therefore apply the Existential lemma.

  We apply the Existential lemma to the computation \( \vdash (\,\cdot\, : B) \Rightarrow \sigma \), with the set \( V \) of variables set to just \( \{ x \} \). (For simplicity, we ignore the renaming of the variable, which we can see is not necessary in this case.) By clause (1) of the lemma, there must be a \( t \) such that \( (\,x : = t\, : B) \Rightarrow \sigma \), i.e. that \( (\,x : = t\, : B) \Rightarrow \sigma \).

  Furthermore, by clauses (2)-(4) of the lemma, all \( t \) are such that when \( (\,x : = t\, : B) \Rightarrow \sigma' \), \( \sigma' \leq \sigma \). Therefore, by the induction hypothesis, there is some \( t \) such that \( u(B[x : = t]) = \sigma \), and for all \( t \), \( u(B[x : = t]) \leq \sigma \). That is, \( \sigma \) is the maximum, under \( \leq \), of the truth values of all the instances of \( B \). By the definition of \( u \), \( u(\exists x B) = \sigma \).

- **Case \( B \lor C \):** If \( \vdash (\,\cdot\, : B) \Rightarrow F \) and \( (\,\cdot\, : C) \Rightarrow \sigma \), then by the induction hypothesis, \( u(B) = F \) and \( u(C) = \sigma \); but then by the definition of \( u \), \( u(B \lor C) = \sigma \).

  Otherwise, \( (\,\cdot\, : B) \Rightarrow \sigma \) and \( \sigma \neq F \); but then by the induction hypothesis, \( u(B) = \sigma \), and by the definition of \( u \), \( u(B \lor C) = \sigma \).

- **Case \( s = t \):** Since \( G \) is ground, \( (\,\cdot\, : G) \Rightarrow \sigma \) only when \( s \) is identical to \( t \) and \( \sigma = T \), or when \( s \) and \( t \) are non-identical and \( \sigma = F \). \( u(G) = \sigma \) in both these cases.

- **Case \( p(t_1, \ldots, t_n) \):** The computation of all predicate calls always results in \( U \) with respect to the empty program, and \( u(G) = U \).

- **Case \( \neg B \):** Subcases are on the form and behaviour of \( B \).
  
  - If \( B \) contains special constants, then \( (\,\cdot\, : \neg B) \Rightarrow N \) and \( u(\neg B) = N \).
  
  - If \( B \) is ordinary and \( (\,\cdot\, : B) \Rightarrow T \), then \( (\,\cdot\, : \neg B) \Rightarrow F \). But by the induction hypothesis, \( u(B) = T \), and therefore \( u(\neg B) = F \).
  
  - If \( B \) is ordinary and \( (\,\cdot\, : B) \Rightarrow F \), then \( (\,\cdot\, : \neg B) \Rightarrow T \), by one use of the \( \varepsilon \) rule; but by the induction hypothesis, \( u(B) = F \) and \( u(\neg B) = T \).
  
  - Finally, if \( B \) is ordinary and \( (\,\cdot\, : B) \Rightarrow \sigma \) where \( \sigma \in \{ U, N \} \), then \( (\,\cdot\, : \neg B) \Rightarrow \sigma \) as well; but by the induction hypothesis, \( u(B) = \sigma \), so by the definition of \( u \), \( u(\neg B) = \sigma \) as well.  

Theorem 4.9 (Completeness of SPN wrt \( u \)). For all ground, outer-disjunction \( G \), if \( u(G) = \sigma \), then \( \vdash (\,\cdot\, : G) \DownRightarrow \sigma \).
Proof. By induction on the size of G. Cases are on the form of G.

- **Case B & C:** There are two subcases. If \( v(B) = T \) and \( v(C) = \sigma \), then by the induction hypothesis, \( ((\cdot) : B) \Rightarrow T \) and \( ((\cdot) : C) \Rightarrow \sigma \). Applying the Conjunction Left Success lemma, we see that if \( ((\cdot) : B, C) \Rightarrow \sigma' \), then there must be a \( \theta' \) such that \( (\theta' : C) \Rightarrow \sigma' \); but since C is ground, the choice of substitution does not affect the outcome of its computation. Therefore \( \sigma' = \sigma \), and by an application of the & rule, \( ((\cdot) : B & C) \Rightarrow \sigma \).

  If \( n(B) = \sigma \neq T \), then by the induction hypothesis, \( ((\cdot) : B) \Rightarrow \sigma \). By the Conjunction Left Non-Success lemma, therefore, \( ((\cdot) : B, C) \Rightarrow \sigma \); but then by an application of the & rule, \( ((\cdot) : B & C) \Rightarrow \sigma \).

- **Case \( \exists x B \):** By the definition of \( v \), we know that there must be at least one \( t \) such that \( v(B[x := t]) = \sigma \); we also know that all \( t \) are such that \( v(B[x := t]) \leq_1 \sigma \).

  So by the induction hypothesis, for some \( t \), \( ([x := t] : B) \Rightarrow \sigma \); and for all \( t \), when \( ([x := t] : B) \Rightarrow \sigma' \), \( \sigma' \leq_1 \sigma \).

  Now let \( ((\cdot) : \exists x B) \Rightarrow \sigma'' \), so that \( ((\cdot) : B) \Rightarrow \sigma'' \). If it were the case that \( \sigma'' > \sigma \), then by clause (1) of the Existential Lemma, there would be a \( t \) such that \( ([x := t] : B) \Rightarrow \sigma' > \sigma \), which is not the case. If it were the case that \( \sigma'' < \sigma \), then by clauses (2-4) of the Existential Lemma, all \( t \) would be such that \( ([x := t] : B) \Rightarrow \sigma' < \sigma \), which is not the case. Therefore \( \sigma'' = \sigma \); i.e., \( ((\cdot) : \exists x B) \Rightarrow \sigma \).

- **Case \( B \lor C \):** If \( v(B) = F \) and \( v(C) = \sigma \), then by the induction hypothesis, \( ((\cdot) : B) \Rightarrow F \) and \( ((\cdot) : C) \Rightarrow \sigma \). But then by rule \( \lor 2 \), \( ((\cdot) : B \lor C) \Rightarrow \sigma \). Otherwise, \( v(B) = \sigma \neq F \); by the induction hypothesis, \( ((\cdot) : B) \Rightarrow \sigma \), and by rule \( \lor 1 \), \( ((\cdot) : B \lor C) \Rightarrow \sigma \).

- **Case \( s = t \):** Recall that \( s \) and \( t \) must be closed. If \( s \) and \( t \) are identical, then \( \sigma = T \); but then they clearly unify, so \( ((\cdot) : s = t) \Rightarrow T \) by one application of rule \( =1 \) and one application of rule \( =e \). Otherwise, \( \sigma = F \), \( s \) and \( t \) do not unify, and \( ((\cdot) : s = t) \Rightarrow F \) by one application of rule \( =2 \).

- **Case \( p(t_1, \ldots, t_n) : \sigma = U \), and with respect to the empty program, \( ((\cdot) : p(t_1, \ldots, t_n)) \Rightarrow U \).**

- **Case \( \neg B \):** Subcases are on the form of \( B \) and value of \( v(\neg B) \).
  - If \( B \) contains special constants, \( ((\cdot) : \neg B) \Rightarrow N \) and \( v(\neg B) = N \).
  - If \( B \) is ordinary and \( v(\neg B) = N, T, U, \) or \( F \), then \( v(B) = N, F, U, \) or \( T \), respectively; therefore, by the induction hypothesis, \( ((\cdot) : B) \Rightarrow N, F, U, \) or \( T \) respectively; therefore, by the \( \neg \) rules, \( ((\cdot) : \neg B) \Rightarrow N, T, U, \) or \( F \), respectively.

The complete result is the following.

**Corollary 4.10 (Characterization, \( v \).)** For all ground, outer-disjunction \( G \), \( v(G) = \sigma \) iff \( ((\cdot) : G) \Rightarrow \sigma \).

**Proof.** From the previous two theorems. □
Interpreter suggested by SPN. Applying \( v \) directly to general goal formulas would not work, however, essentially because disjunctions have a compositional semantics only when they are at the top level of a query. Consider, as a counterexample, the query

\[
\exists x((x = x \lor p( )) \land 2 = 3)
\]

If we applied \( v \) to general goal formulas and not just \( \emptyset \) formulas, it would yield \( F \) on this query (every instance of the left-hand conjunct returns \( T \), and the right-hand conjunct returns \( F \)). However, in fact, this query returns \( U \) when computed with respect to the empty program. Essentially, \( v \) does not take account of backtracking.

4.2. Second stage: general queries, empty program

For the next stage, we define a congruence \( \equiv_{du} \) on general goal (G) formulas, and show that congruent goals have the same outcome with respect to \( \emptyset \). We then show that every goal formula is \( \equiv_{du} \)-congruent to an \( \emptyset \) formula, and define the semantics of G formulas with respect to \( \emptyset \) accordingly. \( \equiv_{du} \) is just the congruence induced by the disjunctive unfolding operation of [2].

4.2.1. The disjunctive unfolding congruence

The disjunctive unfolding congruence is an equivalence relation between formulas which is intended to relate pairs of formulas only if they have the same operational outcome.

**Definition 4.11.** \( \equiv_{du} \), the disjunctive unfolding congruence between formulas, is the least congruence such that

- \( (B_1 \lor B_2) \land C \equiv_{du} (B_1 \land C) \lor (B_2 \land C) \);
- \( B \land (C_1 \lor C_2) \equiv_{du} (B \land C_1) \lor (B \land C_2) \), when \( B \) is a negated-disjunction (N) formula;
- \( \exists x(B_1 \lor B_2) \equiv_{du} (\exists xB_1) \lor (\exists xB_2) \).

Note that some congruences that seem reasonable, such as the De Morgan law \( \neg(B \lor C) \equiv (\neg B) \land (\neg C) \), do not preserve operational outcome: for instance, if \( B \) is true and \( C \) has a free variable, then \( \neg(B \lor C) \) flounders but \( (\neg B) \land (\neg C) \) fails. The disjunctive unfolding congruence, however, does preserve operational outcome, as we will see later.

To prove that the disjunctive unfolding congruence preserves operational outcome, it is convenient to define a relation which holds between formulas which are only "one step apart" in the congruence.

**Definition 4.12.** \( G \triangleright_{du} G' \), in words "\( G' \) is a disjunctive \( 1 \)-unfolding of \( G \)"., if \( G' \) is just \( G \) with either:

- a subformula of the form \( (B_1 \lor B_2) \land C \) replaced by \( (B_1 \land C) \lor (B_2 \land C) \); or
a subformula of the form \( B \& (C_1 \lor C_2) \), where \( B \) is a negated-disjunction \((N)\) formula, replaced by \((B \& C_1) \lor (B \& C_2)\); or

- a subformula of the form \( \exists x (B_1 \lor B_2) \) replaced by \((\exists x B_1) \lor (\exists x B_2)\).

Lemma 4.13. \( \equiv_{du} \) is the reflexive-symmetric-transitive closure of the \( \triangleright_{du} \) relation. In other words, \( G \equiv_{du} G' \) iff \( G' \) can be obtained from \( G \) by \( 0 \) or more \( \triangleright_{du} \) steps or their inverses.

Proof. Straightforward from the definitions of \( \equiv_{du} \) and \( \triangleright_{du} \). \( \Box \)

Now for the result on operational equivalence. The proof is a long case analysis, but is straightforward.

Theorem 4.14 (Operational equivalence of \( \equiv_{du} \) formulae). \( \equiv_{du} \) preserves operational outcome on goal stacks; that is, if \((\theta : G_1, \ldots, G_n) \triangleright \sigma \) for some \( n \geq 1 \), and \((\theta : G'_1, \ldots, G'_n) \triangleright \sigma' \), and \( G_i \equiv_{du} G'_i \) for \( 1 \leq i \leq n \), then \( \sigma' = \sigma \).

Proof. The \( G'_i \) formulas can be obtained from the \( G_i \) formulas by zero or more \( \triangleright_{du} \) steps or their inverses. The proof is by induction on the total number of \( \triangleright_{du} \) steps used. The base case, 0, is trivial. For the inductive case, it suffices to prove the result when only one \( \triangleright_{du} \) step has been performed; the rest follows by the induction hypothesis.

Thus we will prove the result when the two goal stacks differ by only one \( \triangleright_{du} \) step. We will do this by induction on the size of the first computation.

Again the base case (1 judgement in the computation) is easy: the rule is either \( =2 \) or \( \neg 3 \), and neither is affected by the change from \( G_1 \) to \( G'_1 \). There are two main subcases to the inductive case. If the top-level connective of \( G_1 \) is unaffected by the change to \( G'_1 \), then the result follows directly from the induction hypothesis. Otherwise, we have sub-subcases on how the top-level connective of \( G_1 \) was transformed by \( \triangleright_{du} \).

Case 1. \( G_1 \) and \( G'_1 \) are \((B_1 \lor B_2) \& C \) and \((B_1 \& C) \lor (B_2 \& C) \), collectively. Schematically, we have one of the following two situations:

(a) if \((\theta : B_1, C, \alpha) \Rightarrow F:\)

\[
\begin{array}{c}
(\theta : B_1, C, \alpha) \Rightarrow F \\
\hline
(\theta : B_2, C, \alpha) \Rightarrow \sigma \\
\hline
(\theta : B_1 \lor B_2, C, \alpha) \Rightarrow \sigma \\
\hline
(\theta : (B_1 \lor B_2) \& C, \alpha) \Rightarrow \sigma
\end{array}
\]

(b) if \((\theta : B_1, C, \alpha) \Rightarrow \sigma:\)

\[
\begin{array}{c}
(\theta : B_1, C, \alpha) \Rightarrow \sigma' \\
\hline
(\theta : B_2, C, \alpha) \Rightarrow F \\
\hline
(\theta : B_1 \lor B_2, C, \alpha) \Rightarrow \sigma \\
\hline
(\theta : (B_1 \& C) \lor (B_2 \& C), \alpha) \Rightarrow \sigma'
\end{array}
\]
(b) if \((\theta : B_1, C, \alpha) \Rightarrow \sigma\), where \(\sigma \neq F\):

\[
\begin{align*}
(\theta : B_1, C, \alpha) &\Rightarrow \sigma \\
(\theta : B_1 \lor B_2, C, \alpha) &\Rightarrow \sigma \\
(\theta : (B_1 \lor B_2) \land C, \alpha) &\Rightarrow \sigma \\
(\theta : (B_1 \lor C) \vee (B_2 \land C), \alpha) &\Rightarrow \sigma
\end{align*}
\]

In either situation, the closures at the top of the first computations are exactly the same as those at the top of the second; therefore \(\sigma = \sigma'\).

Case 2. \(G_1\) and \(G_1'\) are collectively \(B \land (C_1 \lor C_2)\) and \((B \land C_1) \lor (B \land C_2)\), where \(B\) is an \(N\) formula. There are several subcases.

- \((\theta : B) \Rightarrow T\), and \((\theta : B, C_1, \alpha) \Rightarrow F\). In this case we know from the Conjunction Left Success lemma that there is a \(\theta'\) such that \((\theta' : C_1 \lor C_2, \alpha) \Rightarrow \sigma\), but also from another application of that lemma that \((\theta' : C_1, \alpha) \Rightarrow F\). Thus \((\theta' : C_2, \alpha) \Rightarrow \sigma'\), and we have the following two computations:

\[
\begin{align*}
(\theta' : C_1, \alpha) &\Rightarrow F \\
(\theta' : C_2, \alpha) &\Rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
(\theta' : C_1 \lor C_2, \alpha) &\Rightarrow \sigma \\
(\theta : B, C_1 \lor C_2, \alpha) &\Rightarrow \sigma
\end{align*}
\]

Because the form of the closures at the top of each computation is the same, \(\sigma = \sigma'\).

- \((\theta : B) \Rightarrow T\), and \((\theta : B, C_1, \alpha) \Rightarrow \sigma''\), where \(\sigma'' \neq F\). Again, we know from the Conjunction Left Success lemma that there is a \(\theta'\) such that \((\theta' : C_1 \lor C_2, \alpha) \Rightarrow \sigma\), but also that \((\theta' : C_1, \alpha) \Rightarrow \sigma''\). Thus \(\sigma'' = \sigma\), and we have the following two computations:

\[
\begin{align*}
(\theta' : C_1, \alpha) &\Rightarrow F \\
(\theta' : C_2, \alpha) &\Rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
(\theta' : B, C_1 \lor C_2, \alpha) &\Rightarrow \sigma' \\
(\theta : B, C_1 \lor C_2, \alpha) &\Rightarrow \sigma'
\end{align*}
\]

Because the form of the closures at the top of each computation is the same, \(\sigma = \sigma'\).
• $(\theta : B) \Rightarrow F$. By three applications of the Conjunction Left Non-Success lemma, we know that $(\theta : B, C_1 \lor C_2, x) \Rightarrow F$, $(\theta : B, C_1, x) \Rightarrow F$, and $(\theta : B, C_2, x) \Rightarrow F$. Thus we have the following two computations:

\[
\begin{array}{c}
\frac{(\theta : B, C_1, x) \Rightarrow F}{(\theta : B, C_1 \lor C_2, x) \Rightarrow F}
\end{array}
\]

Clearly $\sigma = \sigma' = F$.

• $(\theta : B) \Rightarrow \sigma''$, where $\sigma'' \notin \{T,F\}$. From two applications of the Conjunction Left Non-Success lemma, we know that $(\theta : B, C_1 \lor C_2, x) = \sigma''$ and $(\theta : B, C_1, x) = \sigma''$. Thus we have the following two computations:

\[
\begin{array}{c}
\frac{(\theta : B, C_1 \lor C_2, x) \Rightarrow \sigma''}{(\theta : B, C_1, x) \Rightarrow \sigma'}
\end{array}
\]

Clearly, $\sigma = \sigma' = \sigma''$.

Case 3. $G_1$ and $G'_1$ are $\exists x (B_1 \lor B_2)$ and $(\exists x B_1) \lor (\exists x B_2)$, collectively. If $(\theta : \exists x B_1, x) \Rightarrow F$, then we have the following two computations:

\[
\begin{array}{c}
\frac{(\theta : B_1[x := x'], x) \Rightarrow F}{(\theta : \exists x B_1) \lor (\exists x B_2, x) \Rightarrow \sigma'}
\end{array}
\]

Otherwise, $(\theta : \exists x B_1, x) \Rightarrow \sigma \neq F$ and we have the following two computations:

\[
\begin{array}{c}
\frac{(\theta : B_1[x := x'], x) \Rightarrow \sigma}{(\theta : \exists x B_1, x) \Rightarrow \sigma}
\end{array}
\]

In either case, because the form of the closures at the top of each computation is the same, $\sigma = \sigma'$. □
So $\equiv_{du}$ preserves operational outcome with respect to the empty program. Now we show that every $G$ formula is related to at least one $O$ formula. There is in fact a simple algorithm to produce a $\equiv_{du}$-congruent $O$ formula from any $G$ formula.

**Definition 4.15.** $G'$ is the key subformula [1] of a goal formula $G$ if it is the leftmost disjunction in $G$ which is an immediate subformula of a conjunction ($\&$) or existential ($\exists$) subformula.

$O(G)$, the $O$ form of $G$, is formed from $G$ by the following algorithm:
1. Set $G_0$ to $G$, and $i$ to 0.
2. While $G_i$ has a key subformula:
   a. Set $G_{i+1}$ to $G_i$ with a $\triangleright_{du}$ operation applied to the $\&$ or $\exists$ subformula enclosing its key subformula.
   b. Set $i$ to $i + 1$.
3. Return $G_i$.

**Theorem 4.16 (Termination of $O$ form).** The $O$ form algorithm always terminates, with $O(G)$ being an $O$ formula.

**Proof.** If $G$ has a key subformula $B \vee C$, we will always be able to apply a $\triangleright_{du}$ operation to its key subformula. (This is the case even if $B \vee C$ is the right-hand conjunct of a conjunction, since the left-hand conjunct must be an $N$ formula.) Each repetition of the while loop, however, increases the number of nodes in the tree of the formula, but not the height or branching factor of the tree. Since trees with fixed height and branching factor have bounded size, the process must terminate. The result will be a formula with no disjunctions which are immediate subformulas of $\&$ or $\exists$ subformulas; in other words, an $O$ formula. □

The equivalence class $[G]$ of a formula $G$ under $\equiv_{du}$ therefore consists of formulas which are closely related to each other syntactically, and which all have the same operational outcome. Moreover, the last theorem shows that at least one formula in $[G]$ is an $O$ formula. Since we know that $v$ is a faithful valuation on $O$ formulas, we know that we can get the operational outcome of any formula in $[G]$ by applying $v$ to the $O$ formula in $[G]$.

4.2.2. The valuation on $G$

In this section, we define the valuation $v'$ on the set $G$ of general goal formulas. This valuation is defined in terms of $v$ and $\equiv_{du}$, and is faithful to operational outcome for all goals with respect to the empty program.

**Definition 4.17.** $v'$, a valuation function mapping $G$ formulas to truth values, is the unique total function such that:
- $v'(G) = v(G)$ if $G$ is an $O$ formula; and
- $v'(G) = v'(G')$ if $G \equiv_{du} G'$. 

The fact that such a \( v' \) exists and is unique comes from the observations at the end of the last section.

Now we can show that \( v' \) precisely characterizes the outcome of general goals with respect to \( \emptyset \). This is the main result for \( v' \).

**Theorem 4.18** (Characterization, \( v' \)). For all ground goals \( G \), \( (\cdot : G) \models \emptyset \) iff \( v'(G) = \sigma \).

**Proof.** \( O(G) \) is disjunctive-unfolding-congruent to \( G \). By the fact that \( \cong_{du} \) preserves operational outcome, \( (\cdot : G) \models \sigma \) iff \( (\cdot : O(G)) \models \sigma \). By the Characterization Theorem for \( \nu \), this happens iff \( \nu(O(G)) = \sigma \); but by the definition of \( v' \), this is true iff \( v'(G) = \sigma \). \( \square \)

Could we have given a compositional definition for \( v' \) which did not involve the two-stage construction with \( v \) and \( \cong_{du} \)? It may be possible given a different set of truth values (for instance, one in which a distinction is made between successful formulas that strongly terminate and those that do not), but it seems clear that with the current set of truth values, there is no one-stage, compositional valuation faithful to the behaviour of queries.

**4.2.3. Examples**

**Example 1.** Consider a query of the form

\[
(B_1 \lor B_2) \land (C_1 \lor C_2)
\]

where \( B_1, B_2, C_1, C_2 \) are all \( N \) formulas. This query is \( \cong_{du} \)-congruent to

\[
(B_1 \land (C_1 \lor C_2)) \lor (B_2 \land (C_1 \lor C_2))
\]

but not to

\[
((B_1 \lor B_2) \land C_1) \lor ((B_1 \lor B_2) \land C_2)
\]

because \( B_1 \lor B_2 \) is not an \( N \) formula. Now, since \( B_1 \) and \( B_2 \) are \( N \) formulas, \( A_2 \) is \( \cong_{du} \)-congruent to

\[
(B_1 \land C_1) \lor (B_1 \land C_2) \lor (B_2 \land C_1) \lor (B_2 \land C_2)
\]

which is an \( O \) formula; therefore \( v'(A_1) = v(A_4) \).

Note that the definition of \( \cong_{du} \) contains a side-condition that prevents \( A_3 \) from being congruent to \( A_1 \). This is necessary for the preservation of operational outcome. For example, \( A_3 \) is \( \cong_{du} \)-congruent to

\[
(B_1 \land C_1) \lor (B_2 \land C_1) \lor (B_1 \land C_2) \lor (B_2 \land C_2)
\]

Thus the outcomes of \( A_1 \), \( A_2 \) and \( A_4 \) are always the same (if \( B_1 \) and \( B_2 \) are \( N \) formulas), and the outcomes of \( A_3 \) and \( A_5 \) are the same. But \( A_4 \) and \( A_5 \)
cannot always have the same outcome, because the middle two disjuncts of \((A_4)\) are the reverse of those of \((A_5)\). For instance, the query

\[
(\text{true} \& \text{false}) \lor (\text{true} \& \text{true}) \lor (\text{loop}() \& \text{false}) \lor (\text{loop}() \& \text{true})
\]

succeeds; but if \(\text{loop}()\) diverges, the query

\[
(\text{true} \& \text{false}) \lor (\text{loop}() \& \text{false}) \lor (\text{true} \& \text{true}) \lor (\text{loop}() \& \text{true})
\]

diverges.

The side-condition is necessary, therefore, to ensure that the order of disjuncts in the congruent \(O\) formula reflects the order in which disjuncts will be processed in the computation. By the characterization theorem, we have \(v'(A_1) = v'(A_2) = v'(A_4) = v(A_4)\) and \(v'(A_3) = v'(A_5) = v(A_5)\), but since \(A_4 \not\equiv_{du} A_5\), it is not necessarily the case that \(v(A_4) = v(A_5)\).

**Example 2.** Consider the formula

\[
\exists x(p(x) \& \neg \exists y((gt(x,y) \lor gt(y,x)) \& p(y)))
\]

This formula is \(\equiv_{du}\)-congruent to both

\[
\exists x(p(x) \& \neg \exists y((gt(x,y) \& p(y)) \lor (gt(y,x) \& p(y))))
\]

and

\[
\exists x(p(x) \& \neg ((\exists y(gt(x,y) \& p(y)) \lor \exists y(gt(y,x) \& p(y))))
\]

The latter formula is an \(O\) formula, since its disjunction is directly within a negation. Therefore \(v'(A_6) = v'(A_7) = v'(A_8) = v(A_8)\).

4.3. Final stage: general queries, general programs

The last stage of the truth-functional semantics defines the fully general valuation function \(v_P\) associated with a program \(P\). \(v_P\) is defined in terms of \(v'\) and the fold/unfold congruence \(\equiv_P\) (Definition 2.5) associated with \(P\). \(v'\) was an extension of \(v\) via \(\equiv_{du}\); in a similar, but not identical, manner, \(v_P\) is an extension of \(v'\) via \(\equiv_P\).

4.3.1. The fold/unfold congruence

Recall the definition of \(\equiv_P\).

**Definition 4.19.** The fold/unfold congruence associated with a program \(P\), \(\equiv_P\), is the least congruence such that for every predicate definition \(p(x_1, \ldots, x_n) \leftrightarrow G\) in \(\text{compl}(P)\), we have that

\[
p(t_1, \ldots, t_n) \equiv_P G[x_1 := t_1, \ldots, x_n := t_n]
\]
\(\cong_P\) has the property of preserving operational outcome with respect to program \(P\), as \(\cong_P\) does with respect to the empty program. To prove this, it will again be convenient to define a relation which relates goals which are "one step apart" in the \(\cong_P\) congruence.

**Definition 4.20.** Let \(P\) be a program. \(G \triangleright_P G'\), in words "\(G'\) is a \(P\)-1-unfolding of \(G\)", if \(G'\) is just \(G\) with a subformula of the form \(p(t_1,\ldots,t_n)\) replaced by \(A[x_1 := t_1,\ldots,x_n := t_n]\), where the definition \(p(x_1,\ldots,x_n) \equiv A\) is in \(\text{compl}(P)\).

**Lemma 4.21.** \(\cong_P\) is the reflexive-symmetric-transitive closure of the \(\triangleright_P\) relation. In other words, \(G \cong_P G'\) iff \(G'\) can be obtained from \(G\) by 0 or more \(\triangleright_P\) steps or their inverses.

**Proof.** Straightforward from the definitions of \(\cong_P\) and \(\triangleright_P\). \(\square\)

**Theorem 4.22.** \(\cong_P\) preserves operational outcome with respect to \(P\); that is, if \((\theta : G_1,\ldots,G_n) \not\triangleright_P \sigma\) for \(n \geq 1\), and \((\theta : G_1',\ldots,G_n') \not\triangleright_P \sigma'\), and \(G_i \cong_P G_i'\) for \(1 \leq i \leq n\), then \(\sigma' = \sigma\).

**Proof.** The \(G_i'\) formulas can be obtained from the \(G_i\) formulas by zero or more \(\triangleright_P\) steps or their inverses. The proof is by induction on the total number of \(\triangleright_P\) steps used. The base case, 0, is trivial. For the inductive case, it suffices to prove the result when only one \(\triangleright_P\) step has been performed; the rest follows by the induction hypothesis.

Thus we will prove the result when the two goal stacks differ by only one \(\triangleright_P\) step. We will do this by induction on the size of the first computation.

Again, the base case (1 judgement in the computation) is easy: the rule is either \(\not\triangleright P_2\) or \(\not\triangleright P_3\), and neither is affected by the change. There are two main subcases to the inductive case. If neither \(G_i\) or \(G_i'\) is a predicate call, then the change does not affect the top-level connective, and the result follows directly from the induction hypothesis. Otherwise, \(G_i\) and \(G_i'\) are collectively \(p(t_1,\ldots,t_n)\) and \(A[x_1 := t_1,\ldots,x_n := t_n]\). Clearly, because of the computation step

\[
(\theta : A[x_1 := t_1,\ldots,x_n := t_n], x) \not\triangleright_P \sigma
\]

the two computations have identical results. \(\square\)

We must now prove that every goal \(G\) has a \(P\)-unfolding \(G'\) which has the same outcome with respect to the empty program that \(G\) does with respect to \(P\). We prove a generalization of this, for induction purposes.

**Theorem 4.23** (\(P\)-congruent form theorem). Let \((\theta : G_1,\ldots,G_n) \not\triangleright P \sigma\). Then there exist \(G_1',\ldots,G_n'\) such that \(G_i \cong_P G_i'\) for all \(1 \leq i \leq n\), and \((\theta : G_1',\ldots,G_n') \not\triangleright P \sigma\).
Proof. By induction on the size of the first computation. In the base case (1 judgement in computation), $G'_i = G_i$ for all $1 \leq i \leq n$, since the rule used must have been $\rightarrow 2$, $p2$, $\neg 3$, or $e$, and using the empty program instead of $P$ does not affect the result. For the induction case, subcases are on the form of the last rule used.

All subcases follow immediately from the induction hypothesis except for $p1$. In that subcase, $G_1$ is some $p(t_1, \ldots, t_n)$. Schematically, we have:

\[(b) \quad (\theta : A(t_1, \ldots, t_n), G_2, \ldots, G_n) \Rightarrow \sigma \xrightarrow{IH} (\theta : G'_1, G'_2, \ldots, G'_n) \Rightarrow \sigma \quad (c)\]

\[(a) \quad (\theta : p(t_1, \ldots, t_n), G_2, \ldots, G_n) \Rightarrow \sigma\]

By assumption, we have (a) and (b). By the induction hypothesis, there are $G'_1, \ldots, G'_n$ such that (c). But since $p(t_1, \ldots, t_n) \approx_P A(t_1, \ldots, t_n)$, and $A(t_1, \ldots, t_n) \approx_P G'_1$, certainly $p(t_1, \ldots, t_n) \approx_P G'_i$ (by the transitivity of $\approx_P$). So $G'_i \approx_P G_i$ for all $1 \leq i \leq n$, and the result holds. \(\square\)

The consequences here are slightly different than with $\approx_{du}$. The equivalence class $[G]$ of $G$ under $\approx_P$ contains formulas which all have the same outcome w.r.t. $P$. Also, it is always the case that there is one formula $G'$ in $[G]$ for which $v'(G')$ yields that outcome. However, it is not totally decidable what the formula is, because the search for it may be non-terminating.

4.3.2. The valuation $v_P$

Finally, we reach the valuation $v_P$ associated with $P$. This can be seen as the "denotation" of a program in this style of semantics.

First, we define another ordering on the truth values, analogous to the "information" or "definedness" ordering of $\leq_k$. It orders truth values based on how much information is contained in them.

Definition 4.24. The ordering $\leq_k$ between truth values is defined by: $U \leq_k N$, $U \leq_k T$, $U \leq_k F$ (see Fig. 4).

$v_P$ is defined in terms of $v'$, $\approx_P$, and $\leq_k$.

Definition 4.25. $v_P$, a valuation function for program $P$ mapping ground goal formulas to truth values, is defined as:

$$v_P(G) = \max_k(\{v'(G') \mid G \approx_P G'\})$$

where $\max_k(S)$ is the maximum over $\leq_k$ of the truth values in the set $S$. 
$v_p$ is everywhere defined because of the properties of $v'$ and $\equiv_p$. We have the following lemma:

**Lemma 4.26.** If $G' \equiv_p G''$, and $(()) : G' \not\to \sigma' \neq U$, and $(()) : G'' \not\to \sigma'' \neq U$, then $\sigma' = \sigma''$.

**Proof.** $(()) : G' \not\to \sigma'$ as well, since the definitions in $P$ can have no effect on the outcome. Similarly, $(()) : G'' \not\to \sigma''$. But then $\sigma' = \sigma''$, since $\equiv_p$ preserves operational outcome. □

$v_p$ precisely characterizes the outcome of all ground goals, not just with respect to the empty program now, but with respect to $P$. The following theorem is the culmination of all our work, and shows that $v_p$ can be taken as the denotation of $P$.

**Theorem 4.27** (Characterization, $v_p$). For all ground goals $G$ and programs $P$:

1. If $(()) : G \not\to^p \sigma$, then $v_p(G) = \sigma$.
2. If there is no $\sigma$ such that $(()) : G \not\to^p \sigma$, then $v_p(G) = U$.

**Proof.** (1) If $(()) : G \not\to^p \sigma$, then (by the $P$-congruent form theorem) there must be some $G' \equiv_p G$ such that $(()) : G' \not\to \sigma$. But then by the lemma, no other $G'' \equiv_p G$ can have any outcome other than $U$ or $\sigma$. Now, $v'(G')$ must be $\sigma$, by the characterization theorem for $v'$; so the definedness maximum, over all $G' \equiv_p G$, of $v'(G')$ must be $\sigma$. Thus $v_p(G) = \sigma$.

(2) If there is no such $\sigma$, then there can be no $G' \equiv_p G$ such that $(()) : G' \not\to^p \sigma \in \{N,T,F\}$; because otherwise we would certainly have that $(()) : G' \not\to^p \sigma$, and thus that $(()) : G \not\to^p \sigma$. Thus the definedness maximum, over all $G' \equiv_p G$, of $v'(G')$ must be $U$. □

So $v_p$ is faithful to the operational outcome of queries evaluated under $P$. Note that if we just know that $v_p(G) = U$, we cannot tell whether $G$ would diverge, or terminate returning $U$. However, if $G$ and $P$ contain no references to predicates not defined in $P$, the situation is more clear. Under that reasonable assumption, $v_p(G) = U$ iff $G$ diverges, and $v_p(G) \in \{N,T,F\}$ iff $G$ terminates returning that truth value.

Note that although the result refers only to ground goals, this is no real limitation. Whenever we give a goal to a Prolog system, we intend that any free variables in it be implicitly existentially quantified. It is a simple corollary that

For all goals $G$ and programs $P$:

1. If $(()) : G \not\to^p \sigma$, then $v_p(\exists[G]) = \sigma$.
2. If there is no $\sigma$ such that $(()) : G \not\to^p \sigma$, then $v_p(\exists[G]) = U$. 
4.3.3. Examples

Example 1. Consider the example from the Introduction:

\[
even(0).
\]

\[
even(s(s(x))) \leftarrow even(x).
\]

The positive Clark completion of \( P \), \( \text{compl}(P) \), is:

\[
even(y) \leftrightarrow (y = 0) \lor \exists x (y = s(s(x)) \& even(x))
\]

Now consider the query \( \neg even(s(0)) \). We have that \( v'(even(s(0))) = U \), since \( v'(G) = U \) for all predicate calls \( G \); thus \( v'(\neg even(s(0))) = U \) as well. However,

\[
\neg even(s(0)) \cong_p \neg (s(0) = 0 \lor \exists x(s(0) = s(s(x)) \& even(x))]
\]

which is an \( \mathcal{O} \) formula; and since \( v(s(0) = 0) = v(s(0) = s(s(t))) = F \) for all \( t \), we have that

\[
v'(\neg (s(0) = 0 \lor \exists x(s(0) = s(s(x)) \& even(x)))) = F
\]

Thus \( v_p(\neg even(s(0))) = F \).

Example 2. Consider the program

\[
\text{mem}(x, [x|x])).
\]

\[
\text{mem}(x, [y|x]) \leftarrow \text{mem}(x,xs).
\]

The positive Clark completion of \( P \) is

\[
\text{mem}(x, l) \leftrightarrow \exists xs(l = [x|xs]) \lor \exists y \exists xs(l = [y|xs] \& \text{mem}(x,xs))
\]

Now consider the query \( \text{mem}(b, [a|[b|[ ]]])\). We have that \( \text{mem}(b, [a|[b|[ ]]])\)\. We have that

\[
\cong_p \exists xs([a|[b|[ ]] = [b|xs]) \lor
\exists y \exists xs([a|[b|[ ]] = [y|xs] \& \text{mem}(b,xs))
\]

\[
\cong_p \exists xs([a|[b|[ ]] = [b|xs]) \lor
\exists y \exists xs([a|[b|[ ]] = [y|xs] \& (\exists xs'(xs = [b|xs']) \lor
\exists y \exists xs'(xs = [y|xs']) \& \text{mem}(b,xs')))
\]

The last formula above is an \( \mathcal{O} \) formula. But

\[
v(\exists xs([a|[b|[ ]] = [b|xs])) = F, \quad \text{and}
\]

\[
v(\exists y \exists xs([a|[b|[ ]] = [y|xs] \& \exists xs'(xs = [y|xs']))) = T
\]
by virtue of the fact that when we choose \([y := a, xs := [b]])

\[ v([a|[b[ ]]]) = [a|[b[ ]]]) \& \exists xs'([b[ ]]) = [b|xs']) = T. \]

Therefore \(v_p(mem(b, [a|[b[ ]])) = T. \)

**Example 3.** Consider the Clark-Andreka-Nemeti paradoxical program \([10]\)

\(\text{inflist}([y|ys]) \leftarrow \text{inflist}(ys). \)

Its positive Clark completion is

\(\text{inflist}(x) \leftrightarrow \exists y \exists ys(x = [y|ys] \& \text{inflist}(ys)) \)

Consider the query \(\exists z(\text{inflist}(z)). \) Let \(G_k\) be the result of doing \(k\) unfoldings of \(\text{inflist}(z). \) We have that \(v(G_k[z := t]) = U\) for all \(t\) which are lists of length > \(k\), but \(v(G_k[z := t]) = F\) for all other \(t. \) Now because \(F \leq U, \) we have that \(v(\exists z(G_k)) = U. \)

This holds for all \(k. \) But there is therefore no unfolding \(G\) of \(\exists z(\text{inflist}(z))\) such that \(v(G)\) is anything other than \(U, \) so \(v_p(\exists z(\text{inflist}(z))) = U, \) as desired.

5. Discussion

5.1. Recursive enumerability

The valuation \(v_p, \) the denotation of \(P, \) is the only valuation in this paper which is not a total recursive function. \(v\) is recursively defined on the structure of its argument, and \(v'\) can be characterized as just \(v\) applied to the (computable) \(O\) form of its argument. Another way of seeing that \(v\) and \(v'\) are total recursive is that they are simply the outcomes of the computation process defined by SPN w.r.t. the empty program, which always terminates because it does not expand predicate calls.

\(v_p, \) on the other hand, is a total function, but not recursive; to find whether there is a \(G'\) such that \(G \cong_p G'\) and \(v'(G') \neq U\) involves a potentially infinite search. This is to be expected, of course: finding a total recursive \(v_p\) which soundly and completely characterized SPN computations would be equivalent to solving the halting problem.

The fact that the earlier stages \(v\) and \(v'\) are total recursive is an appealing feature of this congruential semantics, since it suggests we have “factored out” the non-recursively enumerable elements into the last stage.

5.2. Logicalness of characterization

The three-stage characterization given here inherits its logicalness from the congruences and valuations used in its construction. It would be better if we could collapse some of the stages together, for instance by internalizing the predicate unfolding stage within the valuation; but this seems impossible to do cleanly, in part because of the Clark-Andreka-Nemeti anomaly.
The logical flavour of the congruences comes from the following fact:

**Proposition 5.1.** If \( G \cong_p G' \) or \( G \cong_{du} G' \), then \( \text{compl}(P) \models (G \leftrightarrow G') \).

The proposition is true because the congruences only ever replace subformulae by subformulae which are equivalent in the models of \( \text{compl}(P) \). Thus, looking at a goal for the moment as a first-order formula, by unfolding predicates and taking the disjunctive unfolding of the resulting formula, we are simply transforming the goal into an equivalent formula.

After unfolding predicates and disjunctions, we are applying a non-standard valuation function \( v \) to the result. If we replaced \( v \) by the standard first-order three-valued intuitionistic valuation (e.g. based on Kripke’s theory of truth [32]), we would get a \( v_p \) which simply reflected the truth of the formula in models of the positive completion of \( P \). However, although our \( v \) is non-standard, it is compositional, and the valuation of an existential formula is the truth maximum of the valuations of all its instances. Indeed, ignoring the left-to-right treatment of the binary connectives and the special treatment of special constants, \( v \) is identical to the standard valuation with respect to the least model of the Clark Equality Theory (CET [21]).

Finally, the definition of \( v_p(G) \) in terms of \( v'(G) \), \( \leq_3 \), and \( \cong_p \) is logical in the sense that we are finding the “most defined” truth value of a certain set of formulae equivalent to the original formula, where our notion of definedness follows standard treatments in multi-valued logic. It would be better if we could define \( v_p(G) \) in a way that was fully compositional with respect to the structure of \( G \); but several counterexamples (such as the one given at the start of Section 4) suggest that this is not possible.

In particular, let \( P \) once again be the Clark–Andreka–Nemeti paradoxical program

\[
\text{inflist}(f(x)) \leftarrow \text{inflist}(x)
\]

The least model of this program contains no instance of \( \text{inflist}(y) \), so \( \exists y(\text{inflist}(y)) \) is not entailed by it; and yet no top-down interpreter can deduce that the query \( \text{inflist}(y) \) should fail, although top-down interpreters are complete for success. (Analogous problems arise with the “gap” between \( T_P \downarrow \omega \) and the least fixpoint of \( T_P \downarrow \) in fixpoint semantics and Kripke–Kleene semantics [26].) So if we defined \( v_p(\exists y(\text{inflist}(y))) \) in a fully compositional way, as the truth maximum (\( \text{max}_v \)) of the set \( \{v_p(\text{inflist}(t))|t \text{ is a ground term}\} \), we would erroneously conclude that the query failed.

In contrast, \( \text{inflist}(y) \) is given the operationally correct “undefined” truth value in the congruential semantics in this paper, since there is no unfolding of it with a defined truth value. It seems that a correct valuation function for depth-first Prolog must go through some kind of unfolding stage before applying a compositional valuation function.
5.3. Other negation strategies

Many practical programs can be computed without floundering in the operational semantics described here, since negated atoms in clauses often correspond to tests done on ground terms. However, it would still be preferable to be able to characterize other, more complete negation strategies, such as delaying negation [22,38], and it may appear that we could do so with a simple modification of the semantics given here. Unfortunately, this appears not to be the case; only a simple variant, in which floundering closures are discarded and new closures allowed to succeed or fail, seems easily characterizable.

For instance, it may appear that we can characterize delaying negation by modifying the function \( u \) so that \( v(B \& C) = u(C \& B) \) if \( v(B) = N \) and \( v(C) \neq N \). This would appear to allow us to characterize a system which delays any subgoal \( G \) which has uninstantiated variables until the variables in \( G \) are instantiated, if ever. However, this semantics does not seem to correspond to the natural operational semantics. For instance, the query \( \exists x \exists y \forall G \), where \( G \) is \((x = 2 \& \neg(y = 3)) \& (x = 3 \& \text{loop}())\), would seem to fail in a natural delaying-negation system. \( v(\exists x \exists y \forall G) \) would be \( F \) only if all instances \( G \theta \) of \( G \) were such that \( v(G \theta) = F \); but in fact this is not the case with the new \( v \). Under the substitution \( \theta = [x := 3, y := k] \), where \( k \) is a special constant, the new \( v \) assigns \( N \) to the left conjunct; thus \( G \theta \) is given the same truth value as \((3 = 3 \& \text{loop}()) \& (3 = 2 \& \neg(k = 3))\), namely \( U \). It is not clear whether there is a \( v \) that will correctly characterize the natural behaviour, and it is not clear if there is an operational semantics which this \( v \) characterizes.

Of course, the operational semantics in this paper takes a somewhat hard line on floundering; if the right-hand disjunct of a disjunction flounders, so does the whole disjunction, as suggested by the rules \( \forall 1 \) and \( \forall 2 \). We could instead slightly liberalize the semantics by allowing the right-hand disjunct to return its result if the left-hand disjunct flounders:

\[
\forall 1: \quad (\theta: B, \alpha) \Rightarrow \sigma \quad \Rightarrow \quad (\theta: B \lor C, \alpha) \Rightarrow \sigma
\]

where \( \sigma \) is \( T \) or \( U \)

\[
\forall 2: \quad (\theta: B, \alpha) \Rightarrow \sigma(\theta: C, \alpha) \Rightarrow \sigma'
\]

where \( \sigma' \) is \( N \) or \( F \)

This operational semantics would cause floundering to behave as just another kind of failure. It should be characterizable by simply modifying \( v \) so that \( v(B \lor C) = v(B) \) if \( v(B) \in \{T,U\} \) and \( v(B \lor C) = v(C) \) otherwise. The operational semantics presented in the main body of this paper was used only because it seemed to embody the straightforward notion of floundering as a run-time error.

5.4. Absence-of-definition as failure

The approach to failure used in the operational semantics here is only one of several that could be taken. I have commented above on the idea of considering floundering to be a kind of failure; the absence of a definition for a predicate could also be taken as a kind of failure.
If we were to try to model this view of failure in the semantics, we would have to make some modifications. Here is one approach to modifying the semantics; others are possible. First, we would modify the operational semantics so that absence of definition is considered as failure (resulting in the $F$ truth value). Second, we would modify the definition of predicate unfolding with respect to a program so that an undefined predicate call is "unfolded" into false. Third, we would have a second operational semantics which considers all predicate definitions to be undefined (resulting in truth value $U$). Finally, all results phrased in terms of the empty program would have to be phrased in terms of the second operational semantics.

These modifications would somewhat weaken the homogeneity of the semantic structure. They are also not entirely necessary if our only goal is to model the behaviour of a Prolog system; many Prolog systems consider lack of definition to be a compile-time or run-time error, to be dealt with outside the execution model per se.

6. Comparison to previous work

Here I discuss the relationship of the semantics in this paper to previous work. The most relevant previous work to consider is in the areas of semantics of depth-first Prolog, semantics of negation, and unfolding semantics.

6.1. Depth-first Prolog

6.1.1. Metatheoretical and denotational approaches

Setting aside presentations of operational semantics of depth-first Prolog [35, 46, 25, 18, 24, 17, 29, 1, 12, 36] most of the previous attempts to characterize depth-first Prolog have been metatheatrical or denotational.

Francez et al. [27] build proof systems which reason about computations, and Barbuti et al. [13] build a transformational semantics relying on the encoding of the termination theory of a program. While these semantics have their good points (in particular, the Barbuti et al. paper models a wide range of control strategies), there is also a great deal of metatheoretic encoding going on in them. For instance, the unification algorithm, which is more properly a part of the operational semantics, is encoded and represented in the high-level semantics.

Many writers [31, 11, 23, 24, 40, 14] have constructed denotational semantics for depth-first Prolog. These attempts make valuable connections between logic programming semantics (even accounting for unsound negation) and general programming language semantics. But it is somewhat disappointing that denotations of logic programs are given as functions from inputs to outputs, when a major point of the logic programming paradigm is that programs do not have to be viewed in this way.

6.1.2. Logical approaches

In contrast to the metatheoretic and denotational approaches, the declarative semantics in this paper is built from three logical basic elements:
A valuation function $v$ which formalizes our intuitive notions of Clark equality, left-to-right conjunction and disjunction, and insist-on-ground negation, and which gives a semantics for free variables via the Tarskian "witness" interpretation of the existential quantifier;

- A congruence $\equiv_{du}$ which formalizes some intuitively clear logical and operational equivalences between goals; and

- A class of congruences $\equiv_p$ which formalizes our intuitive notion of predicate definition.

The technical results of soundness and completeness depend on the exact way in which these elements are defined and made to interact; the appeal of the semantics, like those of Fitting [26] and Kunen [33], lies in the logicalness of the elements.

Other researchers have also explored logical approaches to the semantics of depth-first logic programming. The work of Plümer [42, 43], Bronsard et al. [19], Bezem [16], and Apt et al. [7-9] deals with termination in Prolog programs. However, there is an important distinction to be made here. These researchers are working with the notion of *strong* or universal termination. A query is said to strongly terminate if all branches of its SLD-tree are finite; that is, if it either succeeds returning a finite number of solutions and then stopping, or else fails. If we consider only strong termination, we do not need to consider the left-to-right search strategy of Prolog, since a query strongly terminates iff it does so with respect to any search strategy. The notion of termination considered here is more general, in the sense that we consider a query to terminate successfully if it returns at least one solution. We can recapture strong termination by noting that a query $G$ strongly terminates iff the query $(G \land \text{false})$ fails.

Note also that these researchers are taking variable modes into consideration in their work, thus going beyond the scope of this paper. They are also exploring practical methods for proving termination, which I have not attempted to touch on.

Recently, Stärk [44] has published a semantics for depth-first Prolog with negation which covers some of the same ground as [5]. Stärk's method is to translate a Prolog program into a new program in which each predicate is translated into three predicates: one corresponding to success, one corresponding to failure, and one corresponding to left-termination of the original predicate. However, Stärk avoids the problem of characterizing floundering on negation by insisting that programs meet a mode criterion which guarantees that no queries flounder. The semantics of this paper is more general in the sense that even programs which do not meet this mode criterion and return non-floundering results can be characterized.

6.2. Negation

The corpus of literature on semantics of negation is massive, and I will not try to summarize it here; an excellent overview is given by Harland [30]. Briefly, the semantics given here differs from previous work in:

- not restricting the form of programs, as opposed to the stratified and locally stratified schemes, and related approaches;
- assuming the top-down processing which is common in practice, as opposed to the bottom-up processing assumed by the perfect, stable, well-founded or tight [47] semantics; and in
- considering the depth-first processing of queries which is common in practice, as opposed to most approaches.

It also varies in style from most approaches by adopting the congruential framework rather than treating programs as collections of formulas in a least-model- or fixpoint-like framework.

Naturally, the issue of using top-down and depth-first processing is irrelevant if we set aside the issue of completeness of the operational semantics, as for instance Fitting [26] and Kunen [33] do. But one of the main goals of this paper has been to give a logical and operational semantics which are sound and complete with respect to each other.

6.3. Unfolding semantics

I have already noted the similarity between this work and that of Levi's group on unfolding semantics ([34, 28] and many other papers). Whereas Levi et al. unfold a program in order to find all the atoms entailed by it, the approach of this paper takes the positive Clark completion of the program and unfolds a goal in order to characterize its truth value. Both lines of work have a basis in Tamaki and Sato's notion of unfolding of a logic program [45], which in turn is based on Burstall and Darlington's corresponding technique from functional programming [20].

In general, the benefit of the unfolding technique is the same: it factors out the problem of predicate expansion, allowing us to characterize the behaviour of recursive programs in terms of non-recursive programs. We thus avoid the problem of the Clark–Andreka–Nemeti anomaly (as pointed out earlier), and achieve a symmetric characterization of success and failure.

7. Conclusions and future directions

I have presented here a sound and complete semantics for depth-first Prolog with a weak but logically sound form of negation. The major elements of the semantics are valuations (functions from goals to truth values) and congruences (syntactically compositional equivalence relations). Since both the valuations and the congruences are logically motivated, the semantics has a highly logical flavour.

The techniques used in this paper are intriguing, and may be able to be applied to other commonly-used features of practical Prologs. Examples include other mode-dependent features such as predicate variables, var and nonvar, and dynamic type checking. An extension of the unfolding techniques used here may also result in a semantics for negation within extensions of Prolog such as hereditary Harrop
formulas [37]. I have recently extended the work in this paper to take account of the Prolog "cut"; a preliminary version of this work appears in [6].

In addition, other forms of negation within logic programming might be amenable to similar analysis. I am presently looking at negation in nondeterministic semantics, IC-Prolog-style negation, and (with Verónica Dahl) delaying negation.

Acknowledgements

Thanks to Verónica Dahl, Fred Popowich, Greg Sidebottom, François Bronsard and the referees for useful discussions and comments, and to Lee Naish and Uday Reddy for their help in the preparation of this paper. Special thanks to James Harland for extensive comments on an earlier draft.

Thanks also to the Logic and Functional Programming Group at SFU, in whose lab this work was developed; and to the School of Computing Science, the Centre for Systems Science, and the Lab for Computer and Communications Research at SFU for the use of their facilities. The preparation of this paper was financially supported by the Natural Sciences and Engineering Research Council of Canada, the SFU Centre for Systems Science, and the SFU President's Research Grant Committee, via Infrastructure Grants NSERC 06-4231, CSS 02-7960 and PRG 02-4028, Equipment Grants NSERC 06-4232, CSS 02-7961 and PRG 02-4029, Verónica Dahl's operating grant NSERC 06-4244, and the author's NSERC Postdoctoral Fellowship.

Appendix A. Proofs of technical lemmas

This appendix contains the proofs of the four technical lemmas stated in Section 4.1.

Lemma A.1 (Substitution lemma). Let \((\theta : x) \not\Rightarrow \sigma\). Then for every \(\theta'\) such that \(x\theta = x\theta'\) for every free variable \(x\) of \(x\), \((\theta' : x) \not\Rightarrow \sigma\).

Proof. By induction on the size of the computation. Cases are on the last rule used in the computation. All cases follow directly from the induction hypothesis except the following:

- Case \(\exists\): \(z\) is of the form \((\exists x B, \alpha')\). Let \(x'\) not appear in either \(B, x, \) or \(\theta\). Then \((\theta : B[x := x'], \alpha') \not\Rightarrow \sigma\). Now \(\theta'\) agrees with \(\theta\) on all the free variables of \(B[x := x'], \alpha\); by the induction hypothesis, therefore, \((\theta' : B[x := x'], \alpha') \not\Rightarrow \sigma\) and \((\theta' : B[x := x'], \alpha') \not\Rightarrow \sigma\).

- Case \(=1\): \(s\theta\) is identical to \(s\theta'\) and \(t\theta\) is identical to \(t\theta'\), so the set of mgu's of the two pairs of terms are identical. The result follows from the induction hypothesis.
Case 2: \( s\theta \) is identical to \( s\theta' \) and \( t\theta \) is identical to \( t\theta' \), so neither pair has an mgu.

Case 3: \( \theta \) and \( \theta' \) agree on the free variables (if any) of \( B \), so both queries return \( N \). \( \square \)

Lemma A.2 (Existential lemma). Let \( \alpha \) be a sequence of negated-disjunction (N) formulas, such that \( (\theta : \alpha) \overset{\emptyset}{\Rightarrow} \sigma \). Let \( V \) be a subset of the free variables of \( \alpha\theta \). Define a substitution \( \rho \) as acceptable if \( \rho \) does not assign a value to any variable not in \( V \), but \( \theta\rho \) grounds all variables in \( V \). Define \( \sigma'(\rho) \) as the function such that \( (\theta\rho : \alpha) \overset{\emptyset}{\Rightarrow} \sigma'(\rho) \). Then

EX1. For some acceptable \( \rho \), \( \sigma'(\rho) = \sigma \).

EX2. If \( \sigma = T \), then for all acceptable \( \rho \), \( \sigma'(\rho) \in \{T, U, F\} \).

EX3. If \( \sigma = U \), then for all acceptable \( \rho \), \( \sigma'(\rho) \in \{U, F\} \).

EX4. If \( \sigma = F \), then for all acceptable \( \rho \), \( \sigma'(\rho) = F \).

Proof. (EX1) Let \( V \) be as stated. The proof is by induction on the complexity (number of connectives and \( = \) symbols) of \( \alpha \). Cases are on the last rule used in the computation of \( \sigma \).

Case \&: Schematically, we have

\[
\begin{align*}
(b) & \quad (\theta : B, C, \alpha) \Rightarrow \sigma & \overset{\text{IH}}{\Rightarrow} & (\theta\rho : B, C, \alpha) \Rightarrow \sigma'(\rho) \\
(a) & \quad (\theta : B \& C, \alpha) \Rightarrow \sigma & & (\theta\rho : B \& C, \alpha) \Rightarrow \sigma'(\rho)
\end{align*}
\]

(a) and (b) are true by assumption. By the induction hypothesis, for some acceptable \( \rho \), \( \sigma'(\rho) = \sigma \); that is, (c) is true. (d) follows from (c) by one \& step.

Case \( \exists \): Schematically, we have

\[
\begin{align*}
(b) & \quad (\theta : B[x := x'], \alpha) \Rightarrow \sigma & \overset{\text{IH}}{\Rightarrow} & (\theta\rho : B[x := x'], \alpha) \Rightarrow \sigma'(\rho) \\
(a) & \quad (\theta : \exists x B, \alpha) \Rightarrow \sigma & & (\theta\rho : \exists x B, \alpha) \Rightarrow \sigma'(\rho)
\end{align*}
\]

(a) and (b) are true by assumption. By the induction hypothesis (since \( V \) is also a subset of the free variables in (b)), for some acceptable \( \rho \), \( \sigma'(\rho) = \sigma \); that is, (c) is true. (d) follows from (c) by one \( \exists \) step, since \( \rho \) does not assign a value to \( x' \).

Cases \( \lor 1, \lor 2 \): these cases cannot occur with negated-disjunction (N) formulas.

Case \( =1 \): Schematically, we have

\[
\begin{align*}
(b) & \quad (\theta\theta' : \alpha) \Rightarrow \sigma & \overset{\text{IH}}{\Rightarrow} & (\theta\theta'\rho' : \alpha) \Rightarrow \sigma \\
(a) & \quad (\theta : s = t, \alpha) \Rightarrow \sigma & & (\theta\rho : s = t, \alpha) \Rightarrow \sigma
\end{align*}
\]

(a) and (b) are true by assumption; but by the induction hypothesis, there is some acceptable \( \rho' \) such that (c) is true. But since \( \theta' \) is a unifier of \( s\theta \) and \( t\theta \), \( \theta'\rho' \) must also be. Define \( \rho \) to be the specialization of \( \theta'\rho' \) to the variables of \( V \). Then because of the properties of mgu, (d) follows from (c) in one \( =1 \) step.
• Case $=2$: $\sigma = F$, and because $s\theta$ and $t\theta$ do not unify, every substitution $\rho$ is such that $s\rho$ and $t\rho$ do not unify either. Thus, for all $\rho$, $\sigma'(\rho) = F$.

• Case $\rho_1$: This case cannot occur with respect to the empty program.

• Case $\rho_2$: The predicate call will return $U$ no matter what the substitution in the closure is; so all $\rho$ are such that $\sigma'(\rho) = U$. The result follows straightforwardly from the induction hypothesis.

• Case $\neg 1$: No $\rho$ can change the course of the computation of $B$ because $B\theta$ contains no free variables. Therefore $\sigma'(\rho) = F$ for all $\rho$. Note that in this and the other $\neg$ cases, the induction hypothesis does not need to be invoked on $B$ (which might not be a negated-disjunction formula).

• Case $\neg 2$: $B\theta$ contains no free variables, so no $\rho$ can affect the course of its computations. Schematically, therefore, we have

$$
\begin{align*}
&\quad \quad \quad (\theta : x) \Rightarrow \sigma \quad \overset{IH}{\Rightarrow} \quad (\theta \rho : x) \Rightarrow \sigma'(\rho) \\
&\quad \quad \quad (\theta : \neg B, x) \Rightarrow \sigma \quad \overset{IH}{\Rightarrow} \quad (\theta \rho : \neg B, x) \Rightarrow \sigma'(\rho)
\end{align*}
$$

(a) and (b) are true by assumption. By the induction hypothesis, there is an acceptable $\rho$ such that (c) is true, with $\sigma'(\rho) = \sigma$.

• Case $\neg 3$: For every free variable in $B\theta$ which is a member of $V$, we can substitute a special constant not found in $B$ and the effect will be the same.

• Case $\neg 4$: Since $B\theta$ contains no free variables, no $\rho$ can affect its computation. Therefore $\sigma'(\rho) = \sigma$ for all $\rho$.

• Case $\epsilon$. Clearly, $\sigma = T$ and $\sigma'(\rho) = T$ for any $\rho$.

(EX2-4) Let $V$ be as stated, and let $\rho$ be acceptable. The proof is by induction on the complexity (number of connectives and $=$ symbols) of $x$. Cases are on the last rule used in the computation of $\sigma$.

• Case $\&$: Schematically, we have

$$
\begin{align*}
&\quad \quad \quad (\theta : B, C, x) \Rightarrow \sigma \quad \overset{IH}{\Rightarrow} \quad (\theta \rho : B, C, x) \Rightarrow \sigma'(\rho) \\
&\quad \quad \quad (\theta : B \& C, x) \Rightarrow \sigma \quad \overset{IH}{\Rightarrow} \quad (\theta \rho : B \& C, x) \Rightarrow \sigma'(\rho)
\end{align*}
$$

(a) and (b) are true by assumption. (EX2). By the induction hypothesis, $(\theta \rho : B, C, x) \Rightarrow \sigma'$, where $\sigma' \in \{T, U, F\}$; that is, (c) is true. (d) follows from (c) by one $\&$ step. (EX3-4). Similar to (EX2).

• Case $\exists$: Schematically, we have

$$
\begin{align*}
&\quad \quad \quad (\theta : B[x := x'], x) \Rightarrow \sigma \quad \overset{IH}{\Rightarrow} \quad (\theta \rho : B[x := x'], x) \Rightarrow \sigma'(\rho) \\
&\quad \quad \quad (\theta : \exists x B, x) \Rightarrow \sigma \quad \overset{IH}{\Rightarrow} \quad (\theta \rho : \exists x B, x) \Rightarrow \sigma'(\rho)
\end{align*}
$$

(a) and (b) are true by assumption. (EX2). By the induction hypothesis (since $V$ is also a subset of the free variables in (b)), we have $(\theta \rho : B[x := x'], x) \Rightarrow \sigma'$ with $\sigma \in \{T, U, F\}$; that is, (c) is true. (d) follows from (c) by one $\exists$ step. (EX3-4). Similar to (EX2).

• Cases $\forall 1, \forall 2$: These cases cannot occur with negated-disjunction ($N$) formulas.
• Case \( l = 1 \), part (EX2). Let us consider the computation \( (\theta \rho : s = t, \alpha) \Rightarrow \sigma'(\rho) \). We have one of the following two situations.
  - The last step in this computation is \( = 2 \), and \( \sigma'(\rho) \) is \( F \). Clearly \( \sigma'(\rho) \in \{T, U, F\} \).
  - The last step is \( = 1 \), and has the form
    \[
    \begin{align*}
    (\theta \rho \theta'' : \alpha) & \Rightarrow \sigma'(
    \rho) \\
    (\theta \rho : s = t, \alpha) & \Rightarrow \sigma(
    \rho)
    \end{align*}
    \]
    But the last step of the computation of \( (\theta : s = t, \alpha) \) has the form
    \[
    \begin{align*}
    (\theta \rho : \alpha) & \Rightarrow \sigma \\
    (\theta : s = t, \alpha) & \Rightarrow \sigma
    \end{align*}
    \]
    Since \( \rho \theta'' \) unifies \( s \theta \) and \( t \theta \), and \( \theta' \) is an \( mgu \) of \( s \theta \) and \( t \theta \), there must be (by the properties of \( mgu \)) a \( \rho' \) such that \( \theta \theta' \rho' \equiv \rho \theta'' \). This \( \rho' \) is acceptable because it need only assign values to the variables in \( V \). Therefore, by the induction hypothesis part (EX2), \( \sigma'(\rho) \in \{T, U, F\} \).
• Case \( l = 1 \), parts (EX3-4). Similar to \( = 1 \), part (EX2).
• Case \( l = 2 \): \( \sigma = F \), so we need only prove (EX4). Because \( s \theta \) and \( t \theta \) do not unify, every substitution \( \rho \) is such that \( s \theta \rho \) and \( t \theta \rho \) do not unify either. Thus, for all \( \rho \), \( \sigma'(\rho) = F \).
• Case \( p_1 \): This case cannot occur with respect to the empty program.
• Case \( p_2 \): \( \sigma = U \), so we need only prove (EX3). The predicate call will return \( U \) no matter what the substitution in the closure is; so all \( \rho \) are such that \( \sigma'(\rho) = U \). The result follows straightforwardly from the induction hypothesis.
• Case \( ~1 \): \( \sigma = F \), so we need only prove (EX4). No \( \rho \) can change the course of the computation of \( B \) because \( B \theta \) contains no free variables. Therefore, \( \sigma'(\rho) = F \) for all \( \rho \). (Note that in this and the other \( ~ \) cases, the induction hypothesis does not need to be invoked on \( B \), which might not be a negated-disjunction formula.)
• Case \( ~2 \): \( B \theta \) contains no free variable, so no \( \rho \) can affect the course of its computations. Schematically, therefore, we have
  \[
  \begin{align*}
  (b) & \quad \ldots \quad (\theta : \alpha) \Rightarrow \sigma \xrightarrow{IH} \ldots \quad (\theta \rho : \alpha) \Rightarrow \sigma'(\rho) \\
  (a) & \quad (\theta : \neg B, \alpha) \Rightarrow \sigma \xrightarrow{IH} (\theta \rho : \neg B, \alpha) \Rightarrow \sigma'(\rho)
  \end{align*}
  \]
  (a) and (b) are true by assumption, (EX2). By the induction hypothesis, (c) is true, and \( \sigma'(\rho) \in \{T, U, F\} \). (d) follows from (c) with one \( ~2 \) step. (EX3) and (EX4): Similar to (EX2).
• Case \( ~3 \): \( \sigma = N \), so this case cannot occur with the assumptions of (EX2)–(EX4).
• Case \( ~4 \): Assuming \( \sigma \neq N \) as per (EX2)–(EX4), we must have \( \sigma = U \); that is, we only need prove (EX3). Since \( B \theta \) contains no free variables, no \( \rho \) can affect its computation. Therefore \( \sigma'(\rho) = \sigma \) for all \( \rho \), and (EX2)–(EX4) hold.
• Case \( e \): \( \sigma = T \), so we need only prove (EX2); but clearly \( \sigma'(\rho) = T \) for any \( \rho \).

Lemma A.3 (Conjunction, Left Success). Let \( \alpha' \) be a sequence of negated-disjunction (N) formulas, such that \( (\theta : \alpha') \Rightarrow T \). Then \( (\theta : \alpha', \alpha) \Rightarrow \sigma \) iff there is some \( \theta' \) such
that
(1) At the top right corner of the derivation of \((\theta : \alpha')\) is the judgement \((\theta' : \varepsilon) \rightarrow T\); and
(2) \((\theta' : \alpha) \Rightarrow \sigma\).

**Proof.** By induction on the complexity of \(\alpha\). Cases are on the last rule applied in the computation of \((\theta : \alpha')\).
- **Case \& : \alpha' is \(B \& C, \alpha''\).** Schematically, we have

\[
\begin{align*}
(c) \quad (\theta' : \varepsilon) & \Rightarrow T \\
(d) \quad (\theta' : \alpha) & \Rightarrow \sigma \\
(b) \quad (\theta : B, C, \alpha'') & \Rightarrow T \\
(e) \quad (\theta : B, C, \alpha'', \alpha) & \Rightarrow \sigma \\
(a) \quad (\theta : B \& C, \alpha'') & \Rightarrow T \\
(f) \quad (\theta : B \& C, \alpha'', \alpha) & \Rightarrow \sigma
\end{align*}
\]

\((\rightarrow)\) We assume (a) and (f) from the theorem statement. (b) and (e) clearly follow.
By the induction hypothesis on (b) and (e), (c) and (d) are true.

\((\leftarrow)\) We assume (a), (c), and (d) from the theorem statement. (b) follows from (a).
By the induction hypothesis on (b), (c), and (d), (e) is true; (f) follows from (e).

- **Cases \# = 1:** similar to \&.
- **Cases \# = 2, p1, p2, \#1, \#3, and \#4:** these cannot be the last rule applied in the computation of \((\theta : \alpha')\), because that computation is supposed to result in \(T\).
- **Case \# = 2: \alpha' is \neg B, \alpha''\).** Schematically, we have

\[
\begin{align*}
(c) \quad (\theta' : \varepsilon) & \Rightarrow T \\
(d) \quad (\theta' : \alpha) & \Rightarrow \sigma \\
(b) \quad (\theta : B) & \Rightarrow F \\
(e) \quad (\theta : \alpha'') & \Rightarrow T \\
(a) \quad (\theta : \neg B, \alpha'') & \Rightarrow T \\
(f) \quad (\theta : \neg B, \alpha'', \alpha) & \Rightarrow \sigma
\end{align*}
\]

\((\rightarrow)\) We assume (a) and (f) from the theorem statement; lines (b) and (e) follow. By
the induction hypothesis on lines (b) and (e), (c) and (d) are true. \((\leftarrow)\) We assume (a),
(c) and (d) from the theorem statement; line (b) follows from (a). By the induction
hypothesis on (b), (c), and (d), line (e) is true; (f) follows from line (e).

- **Case \varepsilon: \alpha' is empty,** and the result holds trivially. \(\square\)

**Lemma A.4** (Conjunction, Left Non-Success) **If** \((\theta : \alpha') \Rightarrow \sigma\), **where** \(\sigma \neq T\), **then for all \(\alpha\),** \((\theta : \alpha', \alpha) \Rightarrow \sigma\).

**Proof.** By induction on the sum of the complexities of all the formulas in \(\alpha'\). Cases
are on the last rule applied in the computation. We will do case \& and case \#2; the
other cases are very similar.
Case &: The last step is of the form

\[
\begin{align*}
(\theta : B, C, \alpha'') &\Rightarrow \sigma \\
(\theta : B \land C, \alpha'') &\Rightarrow \sigma
\end{align*}
\]

By the induction hypothesis, \((\theta : B, C, \alpha'', \alpha) \Rightarrow \sigma\); but then we have the computation

\[
\begin{align*}
(\theta : B, C, \alpha'', \alpha) &\Rightarrow \sigma \\
(\theta : B \land C, \alpha'', \alpha) &\Rightarrow \sigma
\end{align*}
\]

Case \(\lor 2\): the last step is of the form

\[
\begin{align*}
(\theta : B, \alpha'') &\Rightarrow F \\
(\theta : C, \alpha'') &\Rightarrow \sigma
\end{align*}
\]

By the induction hypothesis on the left premiss, \((\theta : B, \alpha'', \alpha') \Rightarrow F\); and by the induction hypothesis on the right premiss, \((\theta : C, \alpha'', \alpha') \Rightarrow \sigma\). But then we have the computation

\[
\begin{align*}
(\theta : B, \alpha'', \alpha') &\Rightarrow F \\
(\theta : C, \alpha'', \alpha') &\Rightarrow \sigma
\end{align*}
\]

\[\square\]

References


