# Tableaus for Logic Programming

Melvin Fitting
mlflc@cunyvm.cuny.edu
Dept. Mathematics and Computer Science
Lehman College (CUNY), Bronx, NY 10468
Depts. Computer Science, Philosophy, Mathematics
Graduate Center (CUNY), 33 West 42nd Street, NYC, NY 10036 *

May 19, 1993

**Abstract**

We present a logic programming language, which we call Proflog, with an operational semantics based on tableaus, and a denotational semantics based on supervaluations. We show the two agree. Negation is well-behaved, and semantic non-computability issues do not arise. This is accomplished essentially by dropping a domain closure requirement. The cost is that intuitions developed through the use of classical logic may need modification, though the system is still classical at a level once removed. Implementation problems are discussed very briefly — the thrust of the paper is primarily theoretical.

## 1 Introduction

Life would be almost perfect if we had a logic programming language that was efficient, treated negation naturally, and had a semantics that exactly fit its computational mechanism. As we all know, life is not almost perfect. But in the spirit of honest compromise, we propose a logic programming language that goes part way towards meeting these ideal goals.

Over the years many semantical approaches have been introduced for logic programming. Often these have been plagued with non-computability problems [2, 5, 12]. This is inevitable because we are dealing with inductive definability over a fixed, infinite domain [8]. One solution is to relax the domain closure condition — assume the universe is not just Herbrand. Kunen does this in [6, 7]. We do it also, but in a different way, using the notion of *supervaluations* [11]. This yields a semantics in which non-computability problems never arise. The cost is that negation is harder to understand than in classical logic.

We propose a computational mechanism (operational semantics) based on semantic tableaus as in [10, 4], enhanced by a rather natural notion of procedure call. This model of computation is easy to understand, and it fits the proposed semantics exactly. The cost is that inefficiency is virtually built in.

The basic ideas have appeared before [3], but it was presented as one of several candidates for a logic programming semantics, so it was easily overlooked. This time we are directing things towards a community with considerable experience with implementation. Is there the possibility of imposing not unreasonable restrictions to produce an efficient language — much as Prolog was

*Research partly supported by NSF Grant CCR-9104015.

produced from an earlier abstract logic programming paradigm? How useful would such a language be in practice? Also, it is clear that instead of classical tableaus, modal or many-valued ones could be used. Although there have been some modal logic programming languages, we seem to have here a uniform mechanism for introducing them. This is unexplored territory. We hope it does not remain so.

When the system presented below was first considered, it was given a name, 'Proflog,' though the name was never used in print. One could think of this as suggesting programming using first-order logic. However we prefer to think of it as suggesting programming as done by a theoretically oriented Professor: semantical correctness outweighs efficiency. But we encourage others not so theoretically oriented to look at the ideas from the opposite point of view. What restrictions on the language will lead to efficiency without loosing naturalness?

It should be noted that the use of full first-order logic as a programming language is not a new idea. Both [1] and [9] propose versions, with the second explicitly using tableaus. The difference here is that clauses are not thought of as simply first-order formulas, but as first-order formulas plus a recursion mechanism, in which the attempt to close one tableau can cause the creation of further tableaus. More details will have to wait until the formal presentation of the system below.

## 2 Syntax

Usually a program determines the content of the Herbrand universe. That is, the formal language considered to be in use is the smallest language in which the program could be written. This leads to problems when 'irrelevant' axioms are added to a program. We take an alternate route — we fix a language, $L$, and then write a program in the language, possibly not using all of it. This avoids some problems that might otherwise arise. In particular, the explicit specification of language plays a role whose importance might be overlooked: during the course of a tableau proof, typically, additional constant or function symbols are introduced. *These can not appear in query answers.* A program, written in a language $L$, can only answer questions about things that have names in $L$. This is a point that seems reasonable intuitively, but also it plays a role in ensuring the soundness of the system presented below.

A *language* $L$ is specified by giving its constant, function, and relation symbols. We assume equality is always among the relation symbols. Then *terms*, *atomic formulas*, and *formulas* are built up in the ways usual in first-order logic. We allow $\wedge$, $\vee$, $\neg$, and $\supset$ as propositional connectives, and both $\forall$ and $\exists$ as quantifiers. We also allow *true* and *false* as propositional constants. We use the word *sentence* for a formula with no free variables. We write $\varphi(x_1, \ldots, x_n)$ to indicate a formula with all its free variables among $x_1,\ldots,x_n$. If $t_1,\ldots,t_n$ are terms, we write $\varphi(t_1, \ldots, t_n)$ to indicate the result of substituting $t_1, \ldots, t_n$ for all free occurrences of $x_1, \ldots, x_n$ in $\varphi(x_1, \ldots, x_n)$.

We now set up a general but rather rigid notion of program. We leave it to others to consider more flexible versions.

**Definition 2.1** *By an $L$-*clause *we mean an expression of the form:*

$$R(x_1, \ldots, x_n) \leftarrow \varphi(x_1, \ldots, x_n)$$

*where $R(x_1, \ldots, x_n)$ is an atomic formula of $L$ and $\varphi(x_1, \ldots, x_n)$ is a formula of $L$. This $L$-clause is said to be* for *the relation $R$. An $L$-program is a finite set of $L$-clauses containing at most one for each relation symbol of $L$ (except equality).*

**Example** Suppose $L$ has 0 as a constant symbol, $s$ as a one-place function symbol, and *even* and *odd*, and of course $=$ as relation symbols. Then the following is a program, intended to recognize the even and the odd numbers, where the number $n$ is identified with $s^n(0)$.

**Program** $P_1$

$$\begin{aligned} \mathit{even}(x) &\leftarrow x = 0 \lor (\exists y)[x = s(y) \land \mathit{odd}(y)] \\ \mathit{odd}(x) &\leftarrow (\forall y)[\mathit{even}(y) \supset \neg(x = y)] \end{aligned}$$

**Example** This time suppose $L$ has 0 as a constant symbol, $s$ as a one-place function symbol, and *win* as a relation symbol (and equality; we will not continue saying this). The program is for a simple version of the game nim, in which there are two players, one starts with a positive integer and, taking turns, a player lowers the number by either 1 or 2. The first player unable to move looses.

**Program** $P_2$

$$\mathit{win}(x) \leftarrow (\exists y)[(x = s(y) \lor x = s(s(y))) \land \neg \mathit{win}(y)]$$

Finally, a *query* is any ground literal of $L$. In the next few sections we say what it means for a query to be a consequence of a program, semantically and operationally.

## 3 Semantics

Partly because of the problem of assigning meanings to programs with clauses like $p \leftarrow \neg p$, three-valued logic has played a role in the semantics of logic programming. One could say this is a natural idea; one could say it is not natural; and one could have no opinion. We say it is natural, and we make it the basis of our work here. In addition to *true* and *false*, there is a third truth value, denoted $\perp$, which is interpreted as *undefined* — if a sentence has $\perp$ as its value it indicates that when used as a query the program will not return a classical value because of infinite regress. But there are several three-valued logics. Kleene's strong three-valued logic has been the usual choice in logic programming. Here we follow a different tradition, and base things on *supervaluations* [11]. We present the central ideas after some background.

We want the underlying logic to be classical, so we use models in the standard sense. The following fixes the notation and terminology we will be using. A *model* for the language $L$ is a structure $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the *domain* of the model, and $\mathcal{I}$ is the *interpretation*. $\mathcal{I}$ associates with each constant symbol $c$ of $L$ a member $c^{\mathcal{I}} \in \mathcal{D}$; with each $n$-place function symbol $f$ of $L$ a function $f^{\mathcal{I}} : \mathcal{D}^n \to \mathcal{D}$; and with each $n$-ary relation symbol $R$ of $L$ an $n$-ary relation $R^{\mathcal{I}} \subseteq \mathcal{D}^n$. We assume the interpretation of the equality symbol, $=^{\mathcal{I}}$, is always the equality relation on the domain, that is, all models are *normal*. We write $\mathcal{M} \models X$ to indicate that the sentence $X$ is true in $\mathcal{M}$; similarly for $\mathcal{M} \not\models X$. We will sometimes work with extensions of the language $L$ — the notion of a model is extended too, by broadening the interpretation to assign meanings to the extra symbols of the language extension. All this is straightforward.

We are going to drop the domain closure condition: we will use models whose domains are not the Herbrand universe — they must, however, contain at least the Herbrand universe. In the Herbrand universe constant and function symbols of $L$ are interpreted freely in the sense that terms are not considered equal unless they are identical. We want our extensions of the Herbrand universe to retain this feature.

**Definition 3.1** *A* weak Herbrand model *for a language $L$ is a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ such that:*

1. *$f^{\mathcal{I}}$ is one-one in each of its arguments, for every function symbol $f$ of $L$;*

2. *If $f$ and $g$ are distinct function symbols of $L$, $f^{\mathcal{I}}$ and $g^{\mathcal{I}}$ have non-overlapping ranges.*

We think of constant symbols as 0-place function symbols. Thus item 2 includes, as special cases, that distinct constant symbols have distinct interpretations, and it never happens that $f^{\mathcal{I}}(d_1, \ldots, d_n)$ and $c^{\mathcal{I}}$ are the same, where $f$ is $n$-ary and $c$ is a constant symbol. It is easy to verify that every weak Herbrand model has a subdomain isomorphic to the usual Herbrand universe, but there may be other 'non-standard' items in it as well. It is this fact that allows us to avoid non-computability problems.

What a program must do is tell us which ground atoms of $L$ are to be taken as true. We are not interested in possible non-standard items — in effect we do not know about them.

**Definition 3.2** *An $L$-*valuation *is a mapping from ground atoms of $L$ (except those involving equality) to the space $\{false, true, \bot\}$. (We exclude ground atoms of the form $t = u$, since the interpretation of $=$ is fixed in all weak Herbrand models.)*

The problem facing us now is how to extend the action of a valuation from ground atoms to arbitrary sentences of $L$. It is here that weak Herbrand models play their role.

**Definition 3.3** *We say a weak Herbrand model $\mathcal{M}$ for $L$* extends *an $L$-valuation $v$ provided, for each ground atom $A$ of $L$:*

$$\begin{array}{rcl} v(A) = true & \Rightarrow & \mathcal{M} \models A \\ v(A) = false & \Rightarrow & \mathcal{M} \not\models A \end{array}$$

Thus a weak Herbrand model extends a valuation provided it agrees with the valuation whenever the valuation assigns a classical truth value. Now the action of valuations is extended to the entire language $L$ by the simple device of requiring a consensus.

**Definition 3.4** *Let $v$ be an $L$-valuation, and $X$ be a sentence of $L$, not necessarily atomic.*

$$v(X) = \begin{cases} true & \text{if } \mathcal{M} \models X \text{ for every } \mathcal{M} \text{ extending } v \\ false & \text{if } \mathcal{M} \not\models X \text{ for every } \mathcal{M} \text{ extending } v \\ \bot & \text{otherwise} \end{cases}$$

This is the supervaluation idea: if all models agree on a classical value, that is the value we use, otherwise the value is undefined, or $\bot$. It is important to note that the behavior of $v$ on non-atomic formulas is *not* truth-functional. That is, there are no truth tables that can be given to determine values for conjunctions, disjunctions, and the like, based on the values of their components. Nonetheless, the assignment of truth values is classical at heart. Now, the primary notion.

**Definition 3.5** *Let $P$ be a program in the language $L$, and let $v$ be an $L$-valuation. We say $v$ is a* supervaluation model *for $P$ if, for each $L$-clause $R(x_1, \ldots, x_n) \leftarrow \varphi(x_1, \ldots, x_n)$ of $P$, and for all ground terms $t_1, \ldots, t_n$ of $L$:*

$$v(R(t_1, \ldots, t_n)) = v(\varphi(t_1, \ldots, t_n)).$$

It is probably not clear that supervaluation models exist at all. In fact, not only is it the case, but there is a smallest one, and a proof is sketched along the following lines. First, an ordering is imposed on the truth values, denoted $\leq_k$. It is displayed in Figure 1.

Next, this ordering is extended to the family of $L$-valuations by setting $v_1 \leq_k v_2$ provided $v_1(A) \leq_k v_2(A)$ for each *ground atom* $A$ of $L$.

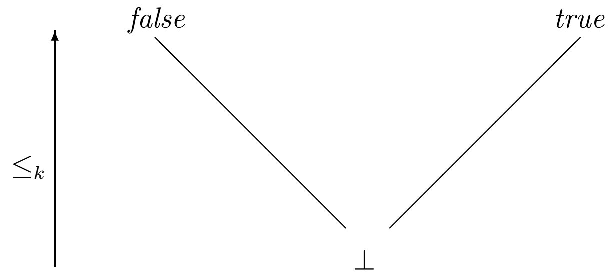

Figure 1: The truth values, ordered

**Lemma 3.6** *If $v_1 \leq_k v_2$ and $X$ is any sentence of $L$, $v_1(X) \leq_k v_2(X)$.*

**Proof** Assume $v_1 \leq_k v_2$. It follows from the definitions that any weak Herbrand model extending $v_2$ also extends $v_1$. The result follows immediately. ∎

Now we define an operator mapping valuations to valuations. It is our version of the 'single-step' operator, usually denoted $T_P$.

**Definition 3.7** *Let $P$ be a program in the language $L$; the operator $\Phi_P$ is defined as follows. For an $L$-valuation $v$, and ground atom $R(t_1, \ldots, t_n)$ of $L$, $\Phi_P(v)(R(t_1, \ldots, t_n)) = v(\varphi(t_1, \ldots, t_n))$ if there is a clause, $R(x_1, \ldots, x_n) \leftarrow \varphi(x_1, \ldots, x_n)$ in $P$, and $\Phi_P(v)(R(t_1, \ldots, t_n)) = \perp$ otherwise.*

**Lemma 3.8** *If $v_1 \leq_k v_2$ then $\Phi_P(v_1) \leq_k \Phi_P(v_2)$; that is, $\Phi_P$ is monotonic.*

**Proof** Immediate from Lemma 3.6. ∎

The space of $L$-valuations, under $\leq_k$, is a complete semi-lattice; in particular, sups of chains exist. It follows that monotonic mappings on this space always have smallest fixed points. It is also easy to see that any fixed point of $\Phi_P$ is a supervaluation model for $P$. Consequently, supervaluation models for $P$ exist, and among them there is a smallest. It is this smallest supervaluation model that we take as supplying the meaning for a program $P$.

**Definition 3.9** *For a program $P$, $s_P$ is the smallest supervaluation model for $P$.*

In [3] the smallest supervaluation model provided what was called the *weak model set semantics*, and it was contrasted with two other semantics which we do not consider here. One of the problems mentioned earlier with other versions of logic programming is that the semantics often turns out to be non-computable, and the operators associated with programs are non-continuous. This problem does not arise here. The single-step operator $\Phi_P$ is always continuous, and reaches its smallest fixed point in $\omega$ steps, through an approximation sequence beginning at the smallest $L$-valuation (mapping everything to $\perp$). Further, the smallest fixed point, $s_P$, is recursively enumerable. This can be shown by relating the semantic definition to a computational procedure, as we will do below.

**Example** Suppose $L$ is a language with function symbol $f$ and constant symbol $c$, and let $\emptyset$ be the program in this language with no clauses — the empty program. The smallest supervaluation model $s_\emptyset$ for the empty program is characterized quite simply. For ground terms $t$ and $u$ of $L$, $s_\emptyset(t = u) = \mathit{true}$ if $t$ and $u$ are identical, and $s_\emptyset(t = u) = \mathit{false}$ otherwise. In particular, $t = f(t)$ is *false* for every ground term $t$. But surprisingly, $s_\emptyset((\exists x)(x = f(x))) = \bot$, so the smallest supervaluation model remains uncommitted on the occurs check issue, even though it 'behaves correctly' on each instance.

The argument to establish this is quite simple. First, if $\mathcal{M}$ is any weak Herbrand model, and $t$ and $u$ are ground terms of $L$, it is easy to check that $\mathcal{M} \models t = u$ if and only if $t$ and $u$ are identical, so the characterization of $s_\emptyset$ is correct. Furthermore, if $\mathcal{M}_1$ is the weak Herbrand model whose domain is exactly the ground terms of $L$, $\mathcal{M}_1 \models \neg(\exists x)(x = f(x))$. Finally, let $\mathcal{M}_2$ be the weak Herbrand model with domain the regular trees — this model also extends $s_\emptyset$, but in it $\mathcal{M}_2 \models (\exists x)(x = f(x))$. Consequently in $s_\emptyset$, $(\exists x)(x = f(x))$ comes out $\bot$.

**Example** Consider Program $P_1$ from Section 2. In the smallest supervaluation model for this program $\mathit{even}(t)$ is true just of those closed terms of the form $s^n(0)$, where $n$ is even. Likewise $\mathit{odd}(t)$ is true just of closed terms designating odd numbers. It is easiest to show this after a computational mechanism has been introduced. The query $(\forall x)(\mathit{even}(x) \vee \neg\mathit{even}(x))$ is true in the semantics, for the simple reason that it is true in every weak Herbrand model. On the other hand, $(\forall x)(\mathit{even}(x) \vee \mathit{odd}(x))$ is $\bot$, essentially because there are weak Herbrand models in which there are 'non-standard' members.

## 4 Semantic tableaus

We present a brief sketch of the basic semantic tableau proof procedure. More detailed treatments can be found in [10, 4]. As originally developed, only sentences appeared in proofs, though they could come from a possibly larger language. It occurred independently to several people that free variables could be used instead, in a way that gave an important role to unification. Details of this approach can be found in [4]. In this section we sketch the simpler version, without free variables. It should be understood, however, that this is not suitable for automation as it stands.

For a given language $L$, by $L^{\mathbf{par}}$ we mean the language arising from the addition to $L$ of a countable collection of new constant symbols. These new constant symbols are called *parameters*. Tableau proofs are of sentences of $L$, but sentences of $L^{\mathbf{par}}$ can appear in them.

It is convenient to use Smullyan's device of grouping sentences into similarity classes. There are four such: the $\alpha$ or conjunctive sentences; the $\beta$ or disjunctive sentences; the $\gamma$ or universal sentences; and the $\delta$ or existential sentences. The $\alpha$ and $\beta$ sentences, and their *components*, and the $\gamma$ and $\delta$ sentences, and their *instances*, are defined in Table 1.

| Conjunctive | | | Disjunctive | | | Universal | | Existential | |
|---|---|---|---|---|---|---|---|---|---|
| $\alpha$ | $\alpha_1$ | $\alpha_2$ | $\beta$ | $\beta_1$ | $\beta_2$ | $\gamma$ | $\gamma(t)$ | $\delta$ | $\delta(t)$ |
| $(X \wedge Y)$ | $X$ | $Y$ | $\neg(X \wedge Y)$ | $\neg X$ | $\neg Y$ | $(\forall x)\Phi$ | $\Phi(t)$ | $(\exists x)\Phi$ | $\Phi(t)$ |
| $\neg(X \vee Y)$ | $\neg X$ | $\neg Y$ | $(X \vee Y)$ | $X$ | $Y$ | $\neg(\exists x)\Phi$ | $\neg\Phi(t)$ | $\neg(\forall x)\Phi$ | $\neg\Phi(t)$ |
| $\neg(X \supset Y)$ | $X$ | $\neg Y$ | $(X \supset Y)$ | $\neg X$ | $Y$ | | | | |

Table 1: Sentence Categories

Now, a *proof* of a sentence $X$ of $L$ is a closed tableau for $\neg X$. A *disproof* of $X$ is a closed tableau for $X$ itself. A tableau is a tree constructed using *branch extension rules* to be given in a moment. It is *for Z* if the sentence $Z$ labels the root. The tableau is *closed* if each branch is closed, and a branch is closed if it contains an explicit contradiction: *false*; or both $A$ and $\neg A$ for some sentence $A$. Finally, there are five branch extension rules. One of these is: a branch containing a sentence of type $\alpha$ can be extended by adding two extra nodes to the end, one labeled with $\alpha_1$, the other labeled with $\alpha_2$. Another is: a branch containing a sentence of type $\beta$ can be extended by adding a left and a right child for its end node, with one labeled $\beta_1$, the other labeled $\beta_2$. These, and the other three rules, are given schematically in Table 2.

| | | | | | | |
|---|---|---|---|---|---|---|
| $\dfrac{\neg\neg Z}{Z}$ | $\dfrac{\neg \mathit{true}}{\mathit{false}}$ | $\dfrac{\neg \mathit{false}}{\mathit{true}}$ | $\begin{array}{c}\alpha\\ \hline \alpha_1\\ \alpha_2\end{array}$ | $\dfrac{\beta}{\beta_1 \mid \beta_2}$ | $\dfrac{\gamma}{\gamma(t)}$ | $\dfrac{\delta}{\delta(p)}$ |
| | | | | | (for any closed term $t$ of $L^{\mathbf{par}}$) | (for a new parameter $p$) |

Table 2: Branch Extension Rules

In the $\delta$ rule, $p$ is to be a parameter that has not previously occurred on the branch. This is what is meant by calling it 'new.' We do not give examples of tableau proofs here. They can be found readily in [10, 4].

## 5 Rules for equality

The tableau rules for equality that we need fall into two categories. There are general rules for equality, and there are special rules arising from the requirements placed on the function symbols of $L$ in Definition 3.1. A fuller treatment of the general rules can be found in [4]. We begin with a brief statement of these. Once again we remind you that we are not presenting a version suitable for automation.

**Reflexivity Rule** For any closed term $t$ of $L^{\mathbf{par}}$, $t = t$ can be added to the end of any tableau branch.

**Substitutivity Rule** Let $X$ and $X'$ be sentences of $L^{\mathbf{par}}$, where $X'$ is like $X$ except that an occurrence of a closed term $t$ in $X$ has been replaced by an occurrence of the closed term $u$. Then, if $X$ and $t = u$ occur on a tableau branch, $X'$ can be added to the branch end.

The Substitutivity Rule allows left-right replacement. It is easy to show that right-left replacement is a derivable rule. More generally, if a branch contains $t = u$, a derived rule allows us to add $u = t$. We assume this when convenient. The tableau system of the previous section, together with these two rules, provides a sound and complete proof system for classical logic with equality. Now for the special equality rules arising from our restriction to weak Herbrand models.

**Free Closure Rule** A tableau branch is closed if it contains:

1. $c = d$ where $c$ and $d$ are distinct constant symbols of $L$;
2. $f(t_1, \ldots, t_n) = c$ where $f$ is a function symbol of $L$ and $c$ is a constant symbol of $L$;
3. $f(t_1, \ldots, t_n) = g(u_1, \ldots, u_k)$ where $f$ and $g$ are distinct function symbols of $L$.

**One-One Rule** If $f$ is a function symbol of $L$, a branch containing $f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n)$ may be extended by adding $t_i = u_i$ to the end, for any $i = 1,\ldots,n$.

We again postpone giving tableau examples until the next section.

## 6 Semantic tableaus with program calls

So far in our discussion of tableaus, programs have played no role. It is time for that to change. A program allows a tableau to call up other, subsidiary, tableaus in an attempt to close its branches. For this section, $P$ is a program in the language $L$.

**Procedure Call Rule** A tableau branch is closed if:

1. it contains a ground atom $R(t_1,\ldots,t_n)$ of $L$, there is a clause $R(x_1,\ldots,x_n) \leftarrow \varphi(x_1,\ldots,x_n)$ in $P$, and there exists a closed tableau for $\varphi(t_1,\ldots,t_n)$;
2. it contains a negated ground atom $\neg R(t_1,\ldots,t_n)$ of $L$, there is a clause $R(x_1,\ldots,x_n) \leftarrow \varphi(x_1,\ldots,x_n)$ in $P$, and there exists a closed tableau for $\neg\varphi(t_1,\ldots,t_n)$.

Note that the ground atom $R(t_1,\ldots,t_n)$ or the negated ground atom $\neg R(t_1,\ldots,t_n)$, must be of $L$, not of $L^{\mathbf{par}}$. The tableau construction may have introduced parameters to deal with existential sentences, but the program $P$ does not 'know' about them; it only 'knows' about the language $L$.

**Definition 6.1** *A* $P$-tableau *is a tableau constructed using the Branch Extension Rules of Section 4, the Equality Rules of Section 5, and the Procedure Call Rule above. A query* $A$ succeeds *with program* $P$ *if there is a closed* $P$*-tableau for* $\neg A$*. A query* $A$ fails *with program* $P$ *if there is a closed* $P$*-tableau for* $A$*.*

Notice that it is possible for a query $A$ to neither succeed nor fail with program $P$ — it just means that $P$-tableaus for $A$ and for $\neg A$ do not close.

**Example** Consider the program $P_2$ from Section 2. We show that for this choice of program, there is a $P_2$-closed tableau for $win(s(s(s(0))))$, and so $win(s(s(s(0))))$ fails.

We begin with a tableau having $win(s(s(s(0))))$ at its root. By the Procedure Call Rule, this tableau closes provided there is a closed $P_2$-tableau for $(\exists y)[(s(s(s(0))) = s(y) \vee s(s(s(0))) = s(s(y))) \wedge \neg win(y)]$, which leads to the following tableau (the numbers are for reference only).

(1) $(\exists y)[(s(s(s(0))) = s(y) \vee s(s(s(0))) = s(s(y))) \wedge \neg win(y)]$
(2) $(s(s(s(0))) = s(c) \vee s(s(s(0))) = s(s(c))) \wedge \neg win(c)$
(3) $(s(s(s(0))) = s(c) \vee s(s(s(0))) = s(s(c)))$
(4) $\neg win(c)$

| | |
|---|---|
| (5) $s(s(s(0))) = s(c)$ | (8) $s(s(s(0))) = s(s(c))$ |
| (6) $s(s(0)) = c$ | (9) $s(s(0)) = s(c)$ |
| (7) $\neg win(s(s(0)))$ | (10) $s(0) = c$ |
| | (11) $\neg win(s(0))$ |

Line (2) is from (1) by the $\delta$ rule, introducing the new parameter $c$. Lines (3) and (4) are from (2) by an $\alpha$ rule, and lines (5) and (8) are from (3) by $\beta$. Now (6) is from (5) by the One-One Rule, as is (9) from (8), and (10) from (9). Finally (7) is from (4) and (6) using Substitutivity, and (11) is from (4) and (10) for the same reason. Now the Procedure Call Rule can be applied to each of (7) and (11). If closed tableaus result, the one above is closed and we are done. We begin with line (7), which calls for the creation of a $P_2$-tableau for $\neg(\exists y)[(s(s(0)) = s(y) \vee s(s(0)) = s(s(y))) \wedge \neg win(y)]$.

$$\begin{array}{c}
(1)\ \neg(\exists y)[(s(s(0)) = s(y) \vee s(s(0)) = s(s(y))) \wedge \neg win(y)] \\
(2)\ \neg[(s(s(0)) = s(0) \vee s(s(0)) = s(s(0))) \wedge \neg win(0)]
\end{array}$$

$$\begin{array}{ll}
(3)\ \neg(s(s(0)) = s(0) \vee s(s(0)) = s(s(0))) & (7)\ \neg\neg win(0) \\
(4)\ \neg s(s(0)) = s(0) & (8)\ win(0) \\
(5)\ \neg s(s(0)) = s(s(0)) & \\
(6)\ s(s(0)) = s(s(0)) &
\end{array}$$

Here line (2) is from (1) using the $\gamma$ rule, with the term 0. Lines (3) and (7) are from (2) by $\beta$, and (4) and (5) are from (3) by $\alpha$. Line (6) is by the Reflexivity Rule, and this branch is closed. Line (8) is from line (7) by the double negation rule. The Procedure Call Rule now is invoked on line (8), starting the following $P_2$-tableau.

$$\begin{array}{l}
(1)\ (\exists y)[(0 = s(y) \vee 0 = s(s(y))) \wedge \neg win(y)] \\
(2)\ (0 = s(d) \vee 0 = s(s(d))) \wedge \neg win(d) \\
(3)\ 0 = s(d) \vee 0 = s(s(d)) \\
(4)\ \neg win(d)
\end{array}$$

$$\begin{array}{ll}
(5)\ 0 = s(d) & (6)\ 0 = s(s(d))
\end{array}$$

Line (2) is from (1) by $\delta$, introducing the parameter $d$. Lines (3) and (4) are from (2) by $\alpha$, and (5) and (6) are from (3) by $\beta$. Now both (5) and (6) close their branches, by the Free Closure Rule.

We have now verified that the left branch of our original extended tableau is closed by virtue of its line (7). The right branch remains, and we leave it to you to carry out a Procedure Call Rule application on line (11), $\neg win(s(0))$.

## 7 Soundness and completeness

We said the tableau rules presented in the last few sections exactly fit the supervaluation semantics. In this section we present a proof of this.

**Definition 7.1** *Let $P$ be a program in the language $L$. We define a valuation $t_P$ as follows. For a ground atom $A$ of $L$:*

$$t_P(A) = \begin{cases} \textit{true} & \textit{if there is a closed } P\textit{-tableau for } \neg A \\ \textit{false} & \textit{if there is a closed } P\textit{-tableau for } A \\ \perp & \textit{otherwise} \end{cases}$$

Now, the basic result of this section is easy to state.

**Theorem 7.2** *For a program $P$ in the language $L$, $t_P = s_P$, the smallest supervaluation model for $P$.*

The rest of the section is given over to a proof of this Theorem.

**Definition 7.3** *Let $v$ be a valuation. A set $S$ of sentences of $L^{\boldsymbol{par}}$ is $v$-*satisfiable *if there is a weak Herbrand model $\mathcal{M}$ extending $v$ in which all members of $S$ are true. A tableau is $v$-*satisfiable *if one of its branches is; and a branch is if the set of sentences on it is.*

The following is a modification of the usual preservation-of-satisfiability result that is at the heart of proving tableau soundness.

**Lemma 7.4** *Let $v$ be a valuation, and let $\mathcal{T}$ be a $v$-satisfiable $P$-tableau.*

1. *The result of applying any tableau rule except the Procedure Call Rule to $\mathcal{T}$ yields another $v$-satisfiable $P$-tableau.*
2. *Suppose also that $v$ is a supervaluation model for $P$ (not necessarily the smallest) and $R(t_1, \ldots, t_n) \leftarrow \varphi(t_1, \ldots, t_n)$ is a ground instance of a clause of $P$. Then:*
   - *(a) if $R(t_1, \ldots, t_n)$ occurs on a $v$-satisfiable branch of $\mathcal{T}$, $\{\varphi(t_1, \ldots, t_n)\}$ is $v$-satisfiable.*
   - *(b) if $\neg R(t_1, \ldots, t_n)$ occurs on a $v$-satisfiable branch of $\mathcal{T}$, $\{\neg\varphi(t_1, \ldots, t_n)\}$ is $v$-satisfiable.*

**Proof** Item 1 is essentially a standard argument (see [10, 4]) extended to handle the equality rules. This is straightforward, and we omit the argument. The two parts of Item 2 have similar proofs; we give the argument for the first part.

Suppose $R(t_1, \ldots, t_n)$ is a ground atom of $L$, and it occurs on branch $\theta$ of tableau $\mathcal{T}$, where $\theta$ is $v$-satisfiable in the weak Herbrand model $\mathcal{M}$. Then $\mathcal{M} \models R(t_1, \ldots, t_n)$, so $v(R(t_1, \ldots, t_n))$ can not be *false*; it must be either *true* or $\perp$. Also suppose $R(t_1, \ldots, t_n) \leftarrow \varphi(t_1, \ldots, t_n)$ is a ground instance of a clause of $P$. Since $v$ is a supervaluation model, $v(R(t_1, \ldots, t_n)) = v(\varphi(t_1, \ldots, t_n))$ so $v(\varphi(t_1, \ldots, t_n))$ is not *false*. Then there must be a weak Herbrand model $\mathcal{M}'$ that extends $v$ such that $\mathcal{M}' \models \varphi(t_1, \ldots, t_n)$, so $\{\varphi(t_1, \ldots, t_n)\}$ is $v$-satisfiable. ■

**Lemma 7.5** *If $v$ is a supervaluation model for $P$, a $v$-satisfiable $P$-tableau is not closed.*

**Proof** This is again standard, but extended to handle the Free Closure Rule and the Procedure Call Rule. The Free Closure Rule is straightforward. The Procedure Call Rule uses Item 2 of the preceding Lemma. We omit details. ■

**Proposition 7.6** *If $v$ is any supervaluation model for $P$, $t_P \leq_k v$; in particular, $t_P \leq_k s_P$.*

**Proof** Suppose $t_P \not\leq_k v$. Then there is a ground atom $A$ of $L$ such that either 1) $t_P(A) = \mathit{true}$ but $v(A) \neq \mathit{true}$, or 2) $t_P(A) = \mathit{false}$ but $v(A) \neq \mathit{false}$. Say it is 1); situation 2) is treated similarly. Since $t_P(A) = \mathit{true}$, there is a closed $P$-tableau for $\neg A$. Since $v(A) \neq \mathit{true}$ there must be a weak Herbrand model $\mathcal{M}$ extending $v$ such that $\mathcal{M} \not\models A$, and so $\mathcal{M} \models \neg A$. This implies that the tableau construction for $\neg A$ begins with a $v$-satisfiable tableau. It follows from Lemma 7.4 that there must exist a $v$-satisfiable tableau that is closed, and this contradicts Lemma 7.4. ■

Given this Proposition, to complete the proof of Theorem 7.2 it is enough to show $t_P$ is itself a supervaluation model.

**Proposition 7.7** *If* $R(t_1, \ldots, t_n) \leftarrow \varphi(t_1, \ldots, t_n)$ *is a ground instance of a clause of* $P$,

$$t_P(R(t_1, \ldots, t_n)) = t_P(\varphi(t_1, \ldots, t_n)).$$

**Proof** Suppose first that $t_P(R(t_1, \ldots, t_n)) = \mathit{true}$ but $t_P(\varphi(t_1, \ldots, t_n)) \neq \mathit{true}$; we derive a contradiction.

By definition of $t_P$, there is a closed $P$-tableau for $\neg R(t_1, \ldots, t_n)$. Since $R(t_1, \ldots, t_n)$ is atomic, the only applicable tableau rule is the Procedure Call Rule, so it must be that there is a closed $P$-tableau for $\neg\varphi(t_1, \ldots, t_n)$. Also, since $t_P(\varphi(t_1, \ldots, t_n)) \neq \mathit{true}$, there must be a weak Herbrand model $\mathcal{M}$ that extends $t_P$ such that $\mathcal{M} \not\models \varphi(t_1, \ldots, t_n)$, so we start the tableau construction for $\neg\varphi(t_1, \ldots, t_n)$ with a $t_P$-satisfiable $P$-tableau. The branch extension rules only produce $t_P$-satisfiable tableaus, so the final, closed one, must be $t_P$-satisfiable, say branch $\theta$ is $t_P$-satisfiable. It takes a small argument to show this is impossible (since we don't know $t_P$ is a supervaluation model, Lemma 7.5 is not applicable).

There are two basic reasons why tableau branch $\theta$ could be closed. One is that it contains an 'obvious' contradiction, that is, it contains an atomic sentence and its negation, or something violating one of the equality conditions. If $\theta$ is closed for this reason it cannot be satisfiable in *any* weak Herbrand model, so it is not $t_P$-satisfiable. The other basic reason $\theta$ could be closed is because of the Procedure Call Rule. Say $\theta$ contains $A$, there is an $L$-instance $A \leftarrow B$ of a clause of $P$, and there is a closed $P$-tableau for $B$. But if there is a closed $P$-tableau for $B$, there will also be a closed $P$-tableau for $A$ (via the Procedure Call Rule), so $t_P(A) = \mathit{false}$. But then $\theta$ cannot be satisfiable in a weak Herbrand model that extends $t_P$, so again it is not $t_P$-satisfiable. (The argument is similar, of course, if $\neg A$ occurs on $\theta$.)

This contradiction establishes that

$$t_P(R(t_1, \ldots, t_n)) = \mathit{true} \Longrightarrow t_P(\varphi(t_1, \ldots, t_n)) = \mathit{true}.$$

Now suppose that $t_P(\varphi(t_1, \ldots, t_n)) = \mathit{true}$ but $t_P(R(t_1, \ldots, t_n)) \neq \mathit{true}$; again we derive a contradiction. This time the argument is like the usual proof of tableau completeness.

Since $t_P(R(t_1, \ldots, t_n)) \neq \mathit{true}$, there is no closed $P$-tableau for $\neg R(t_1, \ldots, t_n)$, and by the Procedure Call Rule, this implies there is no closed $P$-tableau for $\neg\varphi(t_1, \ldots, t_n)$. Suppose we adapt a *systematic* tableau construction method, from [10] or [4], modifying it to take into account the rules of Section 5 and Section 6. This is straightforward, and we skip the details. If we apply this systematic construction to $\neg\varphi(t_1, \ldots, t_n)$ in the usual way, a (possibly) infinite tableau is generated, with an open branch. The set of sentences on that branch, call the set $S$, will be a first-order Hintikka set with equality, as defined in [4].

Suppose $t_P(A) = \mathit{true}$, where $A$ is a ground atom of $L$. Then there must be a closed tableau for $\neg A$, so by the Procedure Call rule, $\neg A$ cannot occur in $S$. Similarly, if $t_P(A) = \mathit{false}$, $A$ cannot occur in $S$. By Hintikka's Lemma, $S$ is satisfiable, say in $\mathcal{M}$. Moreover, we can arrange things so that if the atomic $L$-sentence $A$ is not in $S$, $A$ will be false in $\mathcal{M}$, and if $\neg A$ is not in $S$, $A$ will be true in $\mathcal{M}$. Now, as usual, factor the domain of $\mathcal{M}$ using the equivalence relation that calls terms $t$ and $u$ equivalent if $t = u$ occurs in $S$. This converts $\mathcal{M}$ into a normal model, $\mathcal{M}'$. Further, because of the Free Closure and One-One Rules, $\mathcal{M}'$ will be a weak Herbrand model for the language $L$. Finally, by construction, if $t_P(A) = \mathit{true}$, $\neg A$ is not in $S$, so $A$ is true in $\mathcal{M}$ and hence in $\mathcal{M}'$. Similarly if $t_P(A) = \mathit{false}$. It follows that $\mathcal{M}'$ extends $t_P$. Since $\neg\varphi(t_1, \ldots, t_n) \in S$, $\mathcal{M}' \not\models \varphi(t_1, \ldots, t_n)$, and so $t_P(\varphi(t_1, \ldots, t_n)) \neq \mathit{true}$, which contradicts our original assumption.

This time we have shown that

$$t_P(\varphi(t_1, \ldots, t_n)) = \mathit{true} \Longrightarrow t_P(R(t_1, \ldots, t_n)) = \mathit{true}$$

and so

$$t_P(R(t_1, \ldots, t_n)) = \mathit{true} \Longleftrightarrow t_P(\varphi(t_1, \ldots, t_n)) = \mathit{true}.$$

In a similar way it can be established that

$$t_P(R(t_1, \ldots, t_n)) = \mathit{false} \Longleftrightarrow t_P(\varphi(t_1, \ldots, t_n)) = \mathit{false}.$$

It follows from these equivalences that

$$t_P(R(t_1, \ldots, t_n)) = t_P(\varphi(t_1, \ldots, t_n)).$$

■

## 8 Implementation and Other Issues

Consider program $P_2$ from Section 2. A more usual way of presenting this might be:

$$\begin{array}{rcl} \mathit{win}(x) & \leftarrow & (\exists y)[\mathit{move}(x, y) \wedge \neg \mathit{win}(y)] \\ \mathit{move}(x, y) & \leftarrow & x = s(y) \vee x = s(s(y)) \end{array}$$

It is not hard to check that this does not work as expected. The reason is that, while the interpretation of equality is fixed, the interpretation of *move* in a weak Herbrand model is not — there could be non-standard moves. This means considerable care must be exercised in transporting familiar Prolog programs to Proflog. We are at the point that logic programmers were years ago with respect to Prolog: we need experience in order to develop a feeling for the characteristics of the language.

As outlined above, the tableau-based logic programming mechanism is not suitable for implementation. Some of the reasons are well-known. We used the classical tableau rules as given in [10]. In these, the gamma rule allows the introduction of any closed term, and there is the problem of deciding what ones would be best to choose. This can be replaced by the introduction of a free variable, followed by later unification to close branches, though this causes problems with the delta rule. These problems in turn can be solved by Skolemizing ahead of time, or by using so-called run-time Skolemization. A discussion of these problems and the solutions can be found in [4].

The introduction of free variables into the tableau mechanism is useful for another reason: it makes it possible to ask queries with free variables in familiar Prolog style. But this carries a cost with it. Suppose a branch $\theta$ of a $P$-tableau contains $R(x)$, and $R(x) \leftarrow \varphi(x)$ is a clause of $P$, a program in the language $L$. Suppose further that a closed $P$-tableau can be constructed for $\varphi(x)$, during which $x$ gets instantiated, via unification, to $t$. Does this mean the branch $\theta$ of the original tableau can be closed, after instantiating $x$ to $t$? It does, but only provided that $t$ is a term of $L$, and not of the enlargement of $L$ by Skolem functions. A mechanism for ensuring this must be introduced, and it could prove to be a serious compliction.

Equality plays a significant role in the tableau system presented here, and equality rules add considerable complexity to theorem proving. The Free Closure Rule from Section 5 adds still another layer of complexity. When free variables are allowed, it becomes: a branch can be closed if it contains $t = u$ where $t$ and $u$ can be *disunified.* Disunification means there is a substitution that will make $t$ and $u$ differ on a function symbol of $L$ at corresponding points of the two terms. Now, while there is a single most general unifier for two terms, there can be infinitely many disunifiers. For instance, consider the terms $f(g(f(x)))$ and $f(y)$. Then $\{y \rightarrow f(y_1)\}$ is a disunifier, but so is $\{y \rightarrow g(g(y_1))\}$, and so is $\{y \rightarrow g(f(f(y_1))), x \rightarrow g(x_1)\}$, etcetera. What is needed is a systematic way of generating a sequence of disunifiers. But this sequence is, in fact, the source of multiple answers to a query containing a free variable. The systematic generation of all disunifiers, and the merging of such generation arising from more than one disunification problem, may turn out to be the most intractable implementation issue, but it is clearly a central one.

Naturally, as with all real programming languages, some compromises must be made. Following the lead of Prolog, an incomplete but efficient mechanism will serve, provided the sources of the incompleteness are sufficiently well understood so that a programmer can still write satisfactory programs. It is an interesting question: what are the most natural compromises to make in implementing the system we have described. We hope others will explore this issue.

## References

[1] Bowen, K. A. Programming with full first-order logic. In *Machine Intelligence 10* (1982), Hayes, Michie, and Pao, Eds., Ellis Horwood and John Wiley, pp. 421–440.

[2] Fitting, M. C. A Kripke/Kleene semantics for logic programs. *Journal of Logic Programming 2* (1985), 295–312.

[3] Fitting, M. C. Partial models and logic programming. *Theoretical Computer Science 48* (1987), 229–255.

[4] Fitting, M. C. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[5] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proc. of the Fifth Logic Programming Symposium* (Cambridge, MA, 1988), R. Kowalski and K. Bowen, Eds., MIT Press, pp. 1070–1080.

[6] Kunen, K. Negation in logic programming. *Journal of Logic Programming 4* (1987), 289–308.

[7] Kunen, K. Signed data dependencies in logic programming. *Journal of Logic Programming 7* (1989), 231–245.

[8] Moschovakis, Y. N. *Elementary Induction on Abstract Structures.* North-Holland, 1974.

[9] Schönfeld, W. Prolog extensions based on tableau calculus. In *Proc. of the Ninth Int. Joint Conf. on Artificial Intelligence* (1985), pp. 730–732.

[10] Smullyan, R. M. *First-Order Logic.* Springer-Verlag, 1968.

[11] Van Fraassen, B. Singular terms, truth-value gaps, and free logic. *Journal of Philosophy 63* (1966), 481–485.

[12] Van Gelder, A., Ross, K. A., and Schlipf, J. S. The well-founded semantics for general logic programs. *JACM 38* (1991), 620–650.