Oracle Semantics for Prolog *

Roberto Barbuti[1], Michael Codish[2],
Roberto Giacobazzi[1], Michael Maher[3]

[1] Dipartimento di Informatica
Università di Pisa, I-56125 Pisa
{barbuti,giaco}@di.unipi.it

[2] Department of Computer Science
Katholieke Universiteit Leuven, B-3001 Leuven
codish@cs.kuleuven.ac.be

[3] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
mjm@watson.ibm.com

Abstract. This paper proposes to specify semantic definitions for Prolog in terms of an oracle which provides information on which clauses are to be applied to resolve a given goal. The approach is quite general. It is applicable to pure Prolog to define both operational and declarative semantics; and can be viewed as a basis for both sequential and parallel implementations.
Previous semantic definitions for Prolog typically attempt to encode the sequential depth-first search of the language into various mathematical frameworks. A clause is applied in such semantics only if it is chosen under the search strategy. We prefer instead to specify in a more declarative way the condition upon which a clause is to be applied. The decision whether or not to apply a clause may be viewed as a query to an oracle which may be specified from within the semantics or reasoned about from outside. This approach results in simple and concise semantic definitions which are more useful for arguing the correctness of program transformations and providing the basis for abstract interpretations than previous proposals.

1 Introduction

Logic programming is attractive for its non-algorithmic style and for its clean logical and fixpoint semantics. This leads to a high-level approach to programming, provides for the application of semantic-based techniques, such as abstract interpretation, and eases the justification of program transformations. On the other hand, real logic languages, such as Prolog, have to sacrifice logical purity

* This work has been supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. under grant n. 9100880.PF69 and by the Esprit Basic Research Action 3012 - Compulog.

to achieve acceptable efficiency. As a consequence the semantic simplicity and its advantages are often lost. Due to this dichotomy, logic programming faces two different paradigms: *logic* programming, in which the accent is placed on declarative specification of problems, and logic *programming*, in which the central issue is to write efficient programs to solve problems.

In this paper we weaken this dichotomy by applying the *logic* programming style to define a semantics for the logic *programming* language Prolog. The resulting semantics is more declarative and simpler than previous proposals. As a consequence many of the advantages of *logic* programming are regained.

Declarative semantics for logic programs specify the set of goals which are implied by the program, often by means of a fixpoint construction. This construction abstracts away from operational aspects of the interpreter, for example search strategy. Although declarative semantics can be specified in different frameworks, they are typically defined in terms of an immediate consequence operators such as the classic T_P operator [20]. The least fixpoint of the standard T_P operator specifies the set of ground atoms which are consequences of the program.

On the other hand, operational semantics are much more closely linked to the interpreter. Such semantics generally follow a top-down and goal-dependent approach. Because in these semantics the control is explicit, they are suitable for languages, like Prolog, in which the success set for a program strongly depends on the control. For this reason the semantics of Prolog is usually given by encoding operational semantics in frameworks such as dynamic algebra [6], deterministic transition systems [8], and denotational semantics [4, 9, 16].

In our approach, the semantics of logic programs is parameterized by a specification of the set of clauses applicable at each step in the computation. We call such a specification an *oracle*. Pure Prolog results from the choice of a specific oracle. This enables us to cleanly separate the control and deductive components of the semantics. The oracle can be specified both in a top-down manner, by means of a transition system, and in a bottom-up manner, using "T_P-like" methods. Furthermore, it can be used in both transition system and T_P-like semantics. We claim that the oracle approach is general enough to extend other top-down semantics for sequential logic programs to deal with pure Prolog programs.

Our aim is to provide a semantics simple enough to be used in practice as the basis of semantics-based program analysis and transformation. Previous semantics for Prolog have not been used for this purpose, although they have existed for some time, and we believe this is largely due to their complexity. Instead, work on abstract interpretation, for example, has concentrated on abstracting semantics which ignore the depth-first nature of Prolog computation. Clearly the results of such analyses are inherently less accurate than the results of similar analyses based on truer semantics.

In top-down semantics it is relatively easy to incorporate the oracle and obtain a Prolog semantics. We do this for a transition system semantics in section 3. However we wish to obtain a T_P-like semantics, since such a semantics is preferable for abstract interpretation. Furthermore, we also want a declarative characterization of the Prolog oracle, again for abstract interpretation.

We perform both of these tasks in section 4. In a step-by-step manner, we enhance the usual bottom-up semantics of logic programs until it is powerful enough to characterize termination. (This involves extending the notion of interpretation to a set of clauses, as in [7].) From this we obtain our declarative characterization of the Prolog oracle. We then present a simple T_P-like semantics for Prolog, using the oracle, which is obtained by interchanging the roles played by the program and interpretation in usual T_P-like semantics. The previously mentioned extension of the notion of interpretation is vital here.

The techniques of this paper also apply to several extensions of pure Prolog. In section 6 we briefly outline how our work extends to constraint logic programming languages such as CLP(R), and to Prolog with cut, negation-as-failure, built-ins such as *is*, *var* and *nonvar* ... Extensions to handle dynamic atom selection rules and *assert* and *retract* are more difficult and less elegant, but our methods are still applicable. We close with a short conclusion.

Proofs, not included due to space limitations, are present in the full version of this paper.

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs [18].

Throughout, Σ, Π and Var will respectively denote a set of function symbols, a set of predicate symbols and a denumerable set of variables. The non-ground term algebra over Σ and Var is denoted $Term(\Sigma, Var)$ or $Term$ for short. The set of atoms constructed from predicate symbols in Π and terms from $Term$ is denoted $Atom(\Pi, \Sigma, Var)$ or $Atom$ for short.

We fix a partial function mgu which maps a pair of syntactic objects to an idempotent most general unifier of the objects, if such exists. A statement $\theta = mgu(s, t)$ implies that s and t are unifiable.

Goals are sequences of atoms. A goal $B \in Atom^*$ is typically denoted $\langle a_1, \ldots, a_n \rangle$ or simply by $a_1, \ldots, a_n$. The empty atom sequence is denoted by $true$. The concatenation of goals B_1 and B_2 is denoted $B_1 :: B_2$.

A logic program is a finite set of clauses of the form $h \leftarrow B$ where h is an atom, called the *head*, and B is a goal, called the *body*. The set of clauses constructed from elements of $Atom(\Pi, \Sigma, Var)$ is denoted $Clause(\Pi, \Sigma, Var)$ or $Clause$ for short.

A Prolog program is a logic program. However, it is viewed as a set of predicates and each predicate is viewed as a sequence of clauses. Let P be a Prolog program and a an atom; $pred_P(a)$ denotes the predicate in P for a.

We assume that Prolog evaluation proceeds until all answers are generated and that unification is sound and terminating (i.e. the occur check is performed).

A *substitution* is a mapping from Var to $Term$ which acts as the identity almost everywhere. It extends to apply to any syntactic object in the usual way. We will use the set theoretic notation $\left\{ \, t/_x \;\middle|\; x \in dom(\theta),\ \theta(x) = t \, \right\}$ to represent

θ. A *variable renaming* is a substitution which is a bijection on *Var*. Syntactic objects t_1 and t_2 are *equivalent up to renaming*, denoted $t_1 \sim t_2$, if, for some variable renaming α, $t_1\alpha = t_2$. The equivalence class of t under $\sim$ is denoted by $[t]_\sim$. The set of variables that occur in a syntactic object t is denoted $vars(t)$. We denote by $\langle A_1, \ldots, A_n \rangle \ll_C I$ $(n \geq 0)$ that $A_1, \ldots, A_n$ are elements of the set I renamed apart from C and from each other, note that $i \neq j$ do not imply $A_i \neq A_j$.

3 Top-down Semantics

Top-down semantics for logic programs are semantics for which evaluation of a program's meaning corresponds to a top-down construction of a proof tree for an initial goal. This section presents a standard top-down semantics for logic programs defined in terms of transition systems and shows how it can be extended to provide semantics for Prolog programs.

3.1 Transition systems

Definition 1. (transition system)
A transition system is a pair (X, R) consisting of a set X of states and a transition relation $R \subseteq X \times X$ on states.

In general, transition systems are applied to describe program meaning in terms of the transitive closure of the transition relation which specifies a basic computation step.

Definition 2. (derived states)
Let $\Psi = (X, R)$ be a transition system on states X. If $s\ R^*\ s'$ then we say that s' is derived from s. If $s\ R^k\ s'$ we say that s' is derived from s in k steps.

A transition system can be viewed as associating a *computation tree* with (initial) states.

Definition 3. (computation tree)
Let $\Psi = (X, R)$ be a transition system and $x \in X$ a state. The computation tree associated with x is the tree $T(\Psi, x)$ with root x inductively defined so that every state y derived from x is a vertex in the tree and has a child z iff $y\ R\ z$.

The notion of *universal termination* is formalized by the following:

Definition 4. (universal termination)
Let $\Psi = (X, R)$ be a transition system and $x \in X$ a state. We say that x terminates universally for Ψ iff $T(\Psi, x)$ is of finite height.

We will make use of the following:

Remark.
Let $\Psi = (X, R)$ be a transition system and $x \in X$. Then: (1) x terminates universally for Ψ iff the set $\{\, n \mid \exists y.\ y \textit{ is derived from } x \textit{ in } n \textit{ steps} \,\}$ is finite; and (2) if $T(\Psi, x)$ has finite branching then x terminates universally for Ψ iff $T(\Psi, x)$ is finite.

3.2 Transition systems for logic programs

The transition system approach is well suited and commonly applied to define semantics of logic languages. States correspond to resolvents, the transition relation specifies resolution and the transition system captures the computation tree for initial goals.

Definition 5. (states)
A *state* of the computation is an element in $Atom^* \times Sub$. The set of states is denoted $State$.

Definition 6. (transition system, Ψ_P^{lp})
Let P be a logic program. The transition system Ψ_P^{lp} is $(State, \xrightarrow{lp}_P)$ where $\xrightarrow{lp}_P \subseteq State \times State$ is the smallest relation such that $s \xrightarrow{lp}_P s'$ iff

1. $s = \langle A_1, \ldots, A_i, \ldots, A_k; \theta \rangle$;
2. $H \leftarrow B_1, \ldots, B_n \ll_s P$ such that $\varphi = mgu(A_i\theta, H)$; and
3. $s' = \langle A_1, \ldots, B_1, \ldots, B_n, \ldots, A_k; \theta\varphi \rangle$.

The standard notion of a computation tree for an inital goal G and a program P is captured by $T(\Psi_P^{lp}, \langle G; \epsilon \rangle)$. Definition 6 is often refined (e.g. [18]) by introducing a *selection rule*. In particular, Prolog assumes a left-to-right selection rule. Logic programs with a left-to-right selection rule are termed *sequential logic programs.*

Definition 7. (transition system, Ψ_P^{slp})
Let P be a logic program. The transition system Ψ_P^{slp} is $(State, \xrightarrow{slp}_P)$ where $\xrightarrow{slp}_P \subseteq State \times State$ is the smallest relation such that $s \xrightarrow{slp}_P s'$ iff

1. $s = \langle A_1, \ldots, A_k; \theta \rangle$;
2. $H \leftarrow B_1, \ldots, B_n \ll_s P$ such that $\varphi = mgu(A_1\theta, H)$; and
3. $s' = \langle B_1, \ldots, B_n, A_2, \ldots, A_k; \theta\varphi \rangle$.

Similar to the previous case (for logic programs) the left-to-right computation tree for a goal G and a (sequential) logic program P is captured by $T(\Psi_P^{slp}, \langle G; \epsilon \rangle)$. The notion of the *answers* for a goal G are defined as usual in transition system semantics. In Section 4 we will need also the notions of *partial answers* and *partial answers with depth.*

Definition 8. (answers and partial answers)
Let P be a logic program, G a goal and $L \in \{lp, slp\}$. We say that G' is an:

1. L-answer for G iff $\langle G; \epsilon\rangle \xrightarrow{L*}_P \langle true; \theta\rangle$ and $G' \sim G\theta$;
2. L-partial-answer for G iff $\langle G; \epsilon\rangle \xrightarrow{L*}_P \langle G''; \theta\rangle$ and $G' \sim G\theta$; and
3. L-partial answer for G at depth k iff $\langle G; \epsilon\rangle \xrightarrow{Lk}_P \langle G''; \theta\rangle$ and $G' \sim G\theta$.

While it is well known that the lp-answers and the slp-answers for a program P are identical, this property does not extend to partial answers or partial answers with depth. For example, consider the program $\{\mathtt{p(x)} \leftarrow \mathtt{q(x)}, \mathtt{r(x)}; \mathtt{r(s(x))} \leftarrow \mathtt{r(x)}\}$. The lp-partial answers for $p(x)$ are $\{p(s^n(x)) \mid n > 0\}$, whereas the slp-partial answers are $\{p(x)\}$ (up to variable renaming). Furthermore, observe that the tree $T(\Psi_P^{lp}, \langle p; \epsilon\rangle)$ has an infinite branch while $T(\Psi_P^{slp}, \langle p; \epsilon\rangle)$ is finite.

3.3 An oracle transition system for Prolog

Prolog is a logic programming language which in addition to left-to-right selection of atoms in a state applies also a depth-first search strategy on clauses with the textual ordering. In this case the children of a node in the computation tree are assumed to be ordered to reflect the ordering on clauses. The Prolog computation tree for program P and goal G then corresponds to a pruning of $T(\Psi_P^{slp}, \langle G; \epsilon\rangle)$ which excludes everything to the right of the left-most infinite path. Prolog evaluation corresponds to a depth-first, left-to-right traversal of the Prolog computation tree, and is assumed to find all of the alternative answers. This intuition can be applied to define a transition system for Prolog by imposing a condition on Ψ_P^{slp}, excluding pairs which are to the right of an infinite branch.

The following transition system enhances Definition 7, by introducing a generic *oracle*. A Prolog semantics is then derived as an instance of this definition.

Definition 9. (oracle)
An oracle is a function from logic programs and states to sets of clauses.

Definition 10. (oracle transition system, Ψ_P^f)
Let P be a Prolog program and f be an oracle. The transition system Ψ_P^f is defined as $(State, \xrightarrow{f}_P)$ where $\xrightarrow{f}_P \subseteq State \times State$ is the smallest relation such that $s \xrightarrow{f}_P s'$ iff

1. $s = \langle A_1, \ldots, A_k; \theta\rangle$;
2. $H \leftarrow B_1, \ldots, B_n \ll_s f(P, s)$ such that $\varphi = mgu(A_1\theta, H)$; and
3. $s' = \langle B_1, \ldots, B_n, A_2, \ldots, A_k; \theta\varphi\rangle$.

The difference between this definition and Definition 7 is that here in addition to the usual conditions, an oracle must agree that a clause be applied to resolve a state. Note that for a program P and the oracle $f = \lambda P.\lambda s.P$, Ψ_P^f is precisely Ψ_P^{slp}.

In particular, we are interested in an oracle to specify a transition relation for Prolog. Consider the oracle *prolog* which, for a given program P, associates to states those clauses in P which are eventually chosen by Prolog's search strategy. In this case $\xrightarrow{prolog}_P$ specifies (somewhat tautologously) the transition relation for *Prolog*. A similar argument can be applied to specify transition relations for Prolog with cut, for meta-logical predicates such as $var(x)$ or negation.

As an aside, we note that this approach can be applied to derive parallel implementations of Prolog. An implementation can be viewed as a strategy for constructing the corresponding computation tree. If information about the oracle's decisions is available then the respective subtrees can be constructed in parallel. Even if the implementation is performed without such information, an oracular semantics is a better basis for establishing the correctness of a parallel implementation than a sequential operational semantics.

A less tautologous description of the Prolog oracle can be provided by observing that for a program P and a state $s = \langle a_1, \ldots, a_m\rangle$ the i^{th} clause for a_1 should be chosen by the oracle for P and s iff each path in the Prolog computation tree for s starting with a transition which applies one of the $i-1$ preceding clauses for a_1 is finite.

It is straightforward to show that Prolog computation of a goal G and program P terminates iff the state $\langle G;\epsilon\rangle$ terminates universally for the sequential logic program P; namely iff $T(\Psi_P^{slp}, \langle G,\epsilon\rangle)$ is of finite height [21]. Furthermore, by Konig's Lemma, $T(\Psi_P^{slp}, \langle G,\epsilon\rangle)$ is of finite height iff it is finite because $\xrightarrow{slp}_P$ has finite branching. Hence,

Definition 11. (Prolog oracle)
Let P be a Prolog program and $\langle G;\theta\rangle$ a state. The Prolog oracle is given by

$$prolog(P, \langle G;\theta\rangle) = \left\{ C_i \;\middle|\; \begin{array}{l} G = \langle a_1, \ldots, a_n\rangle,\ n \geq 1,\ pred_P(a_1) = \{C_1, \ldots, C_n\}, \\ 1 \leq i \leq n,\ Q = \{C_1, \ldots, C_{i-1}\}, \\ \langle G;\theta\rangle \xrightarrow{slp}_Q s \;\Rightarrow\; T(\Psi_P^{slp}, s) \textit{ is finite} \end{array} \right\}$$

Note that the relation $\xrightarrow{f}_P$ in Definition 10 is not necessarily decidable, in contrast to $\xrightarrow{lp}_P$. In particular, the relation $\xrightarrow{prolog}_P$ defined in terms of Definition 11 is only semi-decidable. However, for any state s, the vertices in $T(\Psi_P^{prolog}, s)$ are recursively enumerable just as in $T(\Psi_P^{lp}, s)$.

The transition system Ψ_P^{prolog} provides a concise operational semantics for Prolog in the style which is standard for logic programs. It is straightforward to apply a similar approach to provide other types of top-down semantics for

Prolog. In particular the denotational semantics for logic programs with left-to-right selection specified in [19] is easily enhanced by an oracle to a Prolog semantics.

4 Declarative Semantics

The objective of this section is to provide a semantic basis for the analysis of Prolog programs which maintains the simplicity of the T_P-based approach. The basic idea is to introduce a general oracle to a "*T_P-like*" semantics following the approach described in the introduction. A specific Prolog oracle is then introduced to provide Prolog semantics. Both the general oracle semantics and the Prolog oracle itself are derived by enhancing standard bottom-up semantics. As already demonstrated in Section 3 the Prolog oracle must basically provide information on termination. Hence we first establish a declarative specification which captures termination. This specification is then enhanced to characterize termination with a left-to-right selection rule. Finally, we modify a T_P-like semantics in the usual style and add the Prolog oracle to provide a semantics for Prolog.

The approach considered in this section is based on the observation that both top-down and bottom-up semantics of logic programs can be expressed in terms of program *unfolding*. In particular, the non-ground T_P semantics of [11] can be viewed as deriving the facts from the unfoldings of P [17]. While the first two enhancements described below follow the usual bottom-up approach; the third is technically simpler when described by viewing unfolding as a top-down process. The resulting semantics, while top-down, maintains the structure and advantages of the bottom-up approach mentioned above.

4.1 Declarative semantics and termination for logic programs

The general idea is to describe universal termination using *partial answers*, which can be reasoned about using bottom-up semantics. However we will need to associate partial answers with the *depth* at which they are *derived*. Furthermore, as partial answers are dependent on the selection rule; the results of this subsection are refined below to consider left-to-right selection.

It is straightforward to derive partial answers in a bottom-up semantics. The standard approach is first demonstrated in [10]. It is based on the observation that the partial answers for goal G with logic program P are precisely the answers of G with the program P' derived by adding to P the facts $p(x_1, \ldots, x_n)$ for each predicate p/n in P. We can show that this approach extends naturally to provide partial answers at depth k and can be enhanced to consider a left-to-right selection rule. In this paper we follow a different approach which extends more naturally to capture in addition to sequential selection also depth-first search. This is facilitated by enhancing the semantic domain to include clauses, as proposed in [7].

The use of clauses as semantic objects is considered in [13, 7] to model program compositionality. In particular [7] defines a bottom-up semantics in which interpretations consist of clauses instead of atoms. In this approach an interpretation for a program is a set of clauses; each clause representing a partial computation from its head to its body. We apply this approach to capture partial answers in a bottom-up semantics. Notice that a similar technique was developed independently in [12].

We will apply the results of this section to the definition of the declarative semantics of Prolog.

Definition 12. (clause interpretations)
Let P be a logic program. A *clause interpretation* is a subset of $Clause(\Pi, \Sigma, Var)_{\sim}$.

The definition of the declarative clausal semantics is technically simplified by applying the clause interpretation denoted by $\Phi = \{\ [p(\bar{x}) \leftarrow p(\bar{x})]_{\sim} \mid p \in \Pi\ \}$ [7]. The role of Φ in the following definition is to capture empty partial computations and is exemplified below.

Definition 13. (clausal semantics for lp [7])
Let P be a logic program and I be a clausal interpretation. We define

$$\mathcal{T}_P^{lp}(I) = \left\{ [(h \leftarrow B)\sigma]_{\sim} \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, ..., b_n \in P, \\ \langle h_1 \leftarrow B_1, ..., h_n \leftarrow B_n \rangle \ll_C I \cup \Phi, \\ \sigma = mgu(\langle b_1, ..., b_n \rangle, \langle h_1, ..., h_n \rangle), \\ B = B_1 :: \ \ldots \ :: B_n \end{array} \right\}$$

Example 1.
Let P be the program: $\{\mathtt{p(s(x))} \leftarrow \mathtt{q(x), r(x)};\ \ \mathtt{q(0)};\ \ \mathtt{r(x)}\}$ and $G = p(s(x)), r(x)$ be a goal. Evaluation of $lfp(\mathcal{T}_P^{lp})$ proceeds as follows:

(1) $\mathcal{T}_P^{lp}(\emptyset) = P$;
(2) $(\mathcal{T}_P^{lp})^2(\emptyset) = P \cup \{p(s(x)) \leftarrow q(x);\ p(s(0)) \leftarrow r(0);\ p(s(0))\}$;
(3) $(\mathcal{T}_P^{lp})^3(\emptyset) = (\mathcal{T}_P^{lp})^2(\emptyset)$.

$p(s(x)), r(x)$ and$p(s(0)), r(0)$ are both partial answers for G.

The bottom-up semantics can be enhanced to provide partial answers together with the corresponding depths. Semantic objects are then pairs consisting of a clause and an integer. Intuitively the pair $(h \leftarrow B; d)$ represents a partial computation from h to B in d steps. As the elements of Φ correspond intuitively to empty partial computations, we denote:

$$\Phi^{\mathcal{D}} = \{\ ([p(\bar{x}) \leftarrow p(\bar{x})]_{\sim}, 0) \mid p \in \Pi\ \}.$$

Definition 14. (clause/depth interpretations)
Let P be a logic program. A *clause/depth interpretation* is any subset of

$$Clause(\Pi, \Sigma, Var)_{\sim} \times \mathcal{N}.$$

We can enhance $\mathcal{T}_P^{lp}$ to provide partial answers with depth. Given a program P and a goal G, it is then possible to determine from the declarative semantics based on $\mathcal{T}_P^{lp}$ with depth (namely $\mathcal{D}_P^{lp}$) if the computation tree $T(\Psi_P^{lp}, \langle G; \epsilon\rangle)$ for G with P is finite. The details are omitted. A declarative characterization of partial answers at depth d for sequential logic programs requires the introduction of the sequential (left-to-right) selection rule in the $\mathcal{D}_P^{lp}$ function.

Definition 15. (clausal semantics for slp with depth)
Let P be a logic program and I be a clausal interpretation with depth. We define

$$\mathcal{D}_P^{slp}(I) = \left\{ ([(h \leftarrow B)\sigma]_\sim, d) \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in P,\ k \in \{1..n\}, \\[1ex] \left\langle \begin{array}{c} (h_1 \leftarrow true, d_1), \\ \cdots \\ (h_{k-1} \leftarrow true, d_{k-1}), \\ (h_k \leftarrow B_k, d_k) \end{array} \right\rangle \ll_C I \cup \Phi^{\mathcal{D}}, \\[1ex] \sigma = mgu(\langle b_1, \ldots, b_k\rangle, \langle h_1, \ldots, h_k\rangle), \\ d = \sum_{i=1}^{k} d_i + 1,\ B = B_k :: \langle b_{k+1}, \ldots, b_n\rangle \end{array} \right\}$$

We can show that $\mathcal{D}_P^{slp}$ is continuous and consistent with the semantics of [11], namely that $lfp(T_P) = \left\{\ [h]_\sim \;\middle|\; ([h \leftarrow true]_\sim, d) \in lfp(\mathcal{D}_P^{slp})\ \right\}$. We also have

Theorem 16.
Let P be a sequential logic program. G' is a partial answer at depth d for a goal $G = \langle a_1, \ldots, a_n\rangle$ and program P iff $\exists\langle (h_1 \leftarrow true, d_1) \ldots, (h_{k-1} \leftarrow true, d_{k-1}), (h_k \leftarrow B_k, d_k)\rangle \ll_G lfp(\mathcal{D}_P^{slp}) \cup \Phi^{\mathcal{D}}$ such that $\theta = mgu(\langle a_1, \ldots, a_k\rangle, \langle h_1 \ldots, h_k\rangle)$, $G\theta \sim G'$ and $d = \sum_{i=1}^{k} d_i$.

The previous theorem declaratively characterizes partial answers with depth for sequential logic programs. In view of Remark 3.1, we have a declarative characterization of termination which can replace the corresponding operational condition in Definition 11. That is, we can express the Prolog oracle declaratively.

We now turn to the problem of expressing the Prolog semantics in a T_P-like manner.

4.2 An oracle declarative semantics for Prolog

In [5] we stressed the fact that at each step in a Prolog computation, the set of clauses applicable to the left-most atom in a goal depends on the whole goal under the current substitution. This behaves differently from “pure” logic program computations, where the set of applicable clauses depends only on the selected atom.

Example 2.
Consider the following Prolog program P with a state $\langle p(x), q(x); \epsilon\rangle$.

```
p(a).
p(b).
q(a) ← q(a).
```

It is easy to see that in the computation the second clause for p cannot be applied because the tree $T(\Psi_P^{lp}, \langle p(x), q(x); \{a/_x\}\rangle)$ is infinite due to the infinite rewriting of $q(a)$. If we consider only the selected atom then we are unable to detect that the second clause is not used.

For this reason, the oracle semantics can be applied to Prolog only if at each step the state and the set of clauses from which the oracle chooses are available.

In the previous definition of T_P, the clauses of interpretations correspond to partial computations, of arbitrary depth, from heads to bodies. A step in the T_P evaluation corresponds to the application of a partial computation to a program clause. On the other hand the oracle approach is based on the application of a (chosen) program clause to a partial computation. Thus, to apply the oracle in the T_P definition, we need to switch the role of P and I. This is possible because both programs and interpretations have the same structure.

Switching P and I in the T_P definition corresponds to switch the construction of proof trees from bottom-up to top-down. The resulting immediate consequence operator corresponds to the unfolding semantics of the program [17].

An alternative approach to solve this problem is in [5]. However that solution is rather complex and more difficult to abstract for the purposes of abstract interpretation.

Definition 17. (oracle fixpoint semantics)
Let P be a logic program, f an oracle for P and I a clausal interpretation.

$$T_P^f(I) = \left\{ [(h \leftarrow B)\sigma]_\sim \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in I \cup \Phi, \\ h' \leftarrow B' \ll_C f(P, \langle b_1, \ldots, b_n; \epsilon\rangle), \\ \sigma = mgu(b_1, h'), \\ B = B' :: \langle b_2, \ldots, b_n\rangle \end{array} \right\}$$

Let $s_i = \langle G_i; \theta_i\rangle$ for $i = 1, 2$ and P a program. Then we define $s_1 \approx s_2$ iff $G_1\theta_1 \sim G_2\theta_2$. An oracle f is a $\approx$-congruence if whenever $s_1 \approx s_2$ then $f(P, s_1) = f(P, s_2)$. Such oracles are insensitive to the way a goal is represented as a state. It is clear that the Prolog oracle is a $\approx$-congruence. We can link oracle transition system semantics for logic programs and least fixpoint semantics of T_P^f as follows.

Theorem 18.
For any oracle f which is a $\approx$-congruence

$$\Phi \cup lfp(T_P^f) = \left\{ [(p(x) \leftarrow G)\theta]_\sim \;\middle|\; \langle p(x); \epsilon\rangle \xrightarrow{f}{}_P^{*} \langle G; \theta\rangle \right\}.$$

In particular, for the Prolog oracle, the fixpoint semantics based on T_P^{prolog} provides the Prolog answers and partial answers. Note that the unfolding of a clause in Φ with any clause C is the clause C itself, and viceversa; thus

$$T_P^{prolog}(\emptyset) = \bigcup_{p \in \Pi} prolog(P, \langle p(\bar{x}); \epsilon \rangle).$$

Example 3.
Consider the Prolog program P:

```
q(x,y) ← p(x,y).
p(a,b).
p(x,y) ← p(x,y).
p(x,x).
```

$$lfp(\mathcal{D}_P^{slp}) = \left\{ \begin{array}{l} (q(x,y) \leftarrow p(x,y); n);\ (p(x,y) \leftarrow p(x,y); n); \\ (p(a,b); n);\ (p(x,x); n);\ (q(a,b); n+1);\ (q(x,x); n+1) \end{array} \;\middle|\; n \geq 0 \right\}.$$

The only state s such that $T(\Psi_P^{slp}, s)$ is finite is $s = \langle true; \theta \rangle$ for every substitution θ. Thus the evaluation of $lfp(T_P^{prolog})$ proceeds as follows:

(1) $T_P^{prolog}(\emptyset) = \{q(x,y) \leftarrow p(x,y);\ p(a,b);\ p(x,y) \leftarrow p(x,y)\}$;
(2) $(T_P^{prolog})^2(\emptyset) = T_P^{prolog}(\emptyset)$.

Since the only clause with an empty body in $lfp(T_P^{prolog})$ is $p(a,b)$, we can argue that the only (Prolog) answer for the goal $p(x,y)$ in P is $p(a,b)$; whereas $p(x,y)$ is a partial answer for the goal itself.

A bottom-up characterization of Prolog success set requires more complex semantics objects like the ones introduced in [5], where Prolog programs were encoded in an *ask/tell* constraint language by means of a transformation. While the corresponding immediate consequence operator can be used for bottom-up computations, it does not have the simplicity of T_P^{prolog} and would be considerably more difficult to abstract for abstract interpretation.

5 Approximations

We have shown how to "factor" the semantics of Prolog into two components: the oracle, which specifies information on termination, and the deductive part (which can be expressed operationally or declaratively). If we wish to obtain approximate semantics of Prolog, for program analysis, we can approximate each of these components independently. Thus oracle semantics provides a framework for combining work on termination [1, 2, 3, 22] with work on semantics-based analysis (e.g. abstract interpretation, type inference).

Usual analyses, which are based on the semantics of sequential logic programs, can be viewed as using the trivial termination approximation: the oracle

$f = \lambda P.\lambda s.P$. Any more powerful oracle for termination will enhance these existing analyses. In the program of Example 3, an analysis based on sequential logic programs for possible aliasing would detect an aliasing of the arguments of `p` which does not occur in any Prolog computation. The following is a second example showing the advantages of abstracting the meaning of a Prolog program rather than a sequential logic program.

Example 4.
Consider the following Prolog program fragment involving 2-3 trees, which contains a variable naming error in the first clause.

```
min_size(X,Y,S) ←
                 leaf_count(X1,S1),
                 leaf_count(Y,S2),
                 min(S1, S2, S).
leaf_count(leaf(X), 1).
leaf_count(binary(N,L,R), Count) ←
                                  leaf_count(L, LCount),
                                  leaf_count(R, RCount),
                                  Count = LCount + RCount + 1.
leaf_count(ternary(N,L,M,R), Count) ←
                             leaf_count(L, LCount),
                             leaf_count(M, MCount),
                             leaf_count(R, RCount),
                             Count = LCount + MCount + RCount + 1.
```

Prolog execution of `min_size` results in $X1$ being bound (on backtracking) to larger and larger trees composed only of *leaf* and *binary* functors. A type analysis of the program, considered as a sequential logic program, would find that the variable $X1$ has type *2-3tree*, whereas a type analysis based on a Prolog semantics could determine that $X1$ only takes values from the subtype *2tree*. In the context of some strong typing information the programming error can be detected by the Prolog analysis, but not by the sequential logic programming analysis. (Of course other analyses can also detect such a simple error.)

6 Towards "Full" Prolog

Although we have stated results only for pure Prolog, our approach is applicable to several extensions of this language. In this section we briefly outline how it can be applied to these extensions.

The results of this paper extend very straightforwardly to constraint logic programming languages [14], replacing substitutions by constraints and unification by constraint solving. This work applies to pure CLP(R) with depth-first search and left-to-right selection of atoms, that is, CLP(R) without meta-logical features and without delayed constraints. To handle delayed constraints – and thus be applicable to what is essentially the language described in [15] – some further changes must be made. States must distinguish between delayed and (the usual) awake constraints, and transitions and functions must express the (possible) awakening of constraints after an execution step. However the modifications are not difficult to make.

Our approach also applies to Prolog with negation-as-failure and the *once* control facility. To model Prolog execution inside a *not*( ) or *once*( ) call, the oracle needs to detect whether executions begun with previous clauses fail (instead of detecting termination as Definition 11 does). Thus two oracles seem to be needed to model Prolog with these features. However by adding to the state a flag which expresses that the state occurs (or not) in a *not* or *once* environment, and appropriately modifying transitions and functions, the two oracles can be combined. Thus our oracle-based method extends to Prolog with negation-as-failure and *once*. In modelling the extended Prolog oracle we again use the corresponding sequential logic program semantics, this time also involving *not* and *once*. Several further enhancements to $\mathcal{D}_P^{slp}$ are necessary to characterize termination. Since the cut, !, can be expressed ("implemented") in terms of *not* and *once* this work extends to Prolog with cut.

Finally we note that there are some programming features and constructs of Prolog and similar languages which are difficult to present with such a semantics. This is largely due to the non-monotonic behavior (with respect to the instantiation ordering) that they can induce in programs. That is, a goal can have a finite execution while an instance of it does not terminate. Examples are: selection rules which are dependent on the binding environment of the goal (such as those of MU-Prolog or languages based on the Andorra principle) and *assert* and *retract*. However we can still obtain a T_P-like semantics, based on partial answers of all possible atoms instead of only the most general atoms. Prolog "built-ins" such as *is*, *var* and *nonvar* also induce non-monotonic behavior, but can be handled by including such "predicates" in the bodies of the semantic objects.

7 Conclusions

We have shown how to factor the semantics of Prolog into two parts: a termination component, the oracle, and a deductive component. Such a factorization is useful for abstract interpretation of Prolog, since it allows the combination of existing techniques on termination and analysis of logic programs. Thus we can have more accurate analyses of Prolog.

We also gave T_P-like characterizations of each component. These characterizations were designed to facilitate abstract interpretation. We expect they can be usefully the basis of abstract interpretation, not only for the deductive semantics, but also for termination. Our semantics based on $\mathcal{D}_P^{slp}$ shows how to embed in a bottom-up approach the left-to-right selection rule of Prolog and can be useful for the analysis of selection-rule based languages.

It seems that the oracle method is applicable to other programming languages (functional, equational, logical, set-based, ...) which contain multiple "definitions" of a procedure that in theory can execute nondeterministically. Implementations which choose a first applicable definition bear a similar relationship to their theoretical underpinnings as Prolog bears to logic programming. Application of our technique can give a more accurate semantics to the implemented language than the nicer, but more abstract, theoretical model involving nondeterminism.

Acknowledgement

The stimulating discussions with G.Levi and D. Rosenzweig are gratefully acknowledged. This work was done while M. Codish and M. Maher were visiting the University of Pisa. They thank the Dipartimento di Informatica for its hospitality.

References

1. K. R. Apt, R. N. Bol, and J.W. Klop. On the safe termination of PROLOG programs. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 353–368. The MIT Press, Cambridge, Mass., 1989.
2. K. R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Computational Logic*, pages 150–176. Springer-Verlag, Berlin, 1990.
3. K. R. Apt and D. Pedreschi. Proving Termination of General Prolog Programs. In T. Ito and A.R. Meyer, editors, *Proc. of Int. Conf. on Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 265–289, Sendai, Japan, 1991.
4. B. Arbab and D.M. Berry. Operational and Denotational Semantics of Prolog. *Journal of Logic Programming*, 4:309–330, 1987.
5. R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog Control. In *Proc. Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 95–104. ACM Press, 1992.
6. E. Börger. A logical operational semantics of full prolog. In E.Börger, H. Kleine, H. Büning, and M. Richter, editors, *CSL 89. 3rd workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1990.
7. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, 1992. To appear.
8. A. de Bruin and E. de Vink. Continuation semantics for Prolog with cut. In J. Diaz and F. Orejas, editors, *Proc. CAAP 89*, volume 351 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, Berlin, 1989.
9. S. K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 245–269. North-Holland, Amsterdam, 1987.
10. M. Falaschi and G. Levi. Finite failures and partial computations in concurrent logic languages. *Theoretical Computer Science*, 75:45–66, 1990.

11. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
12. M. Gabbrielli and M. C. Meo. Fixpoint Semantics for Partial Computed Answer Substitutions and Call Patterns. In H. Kirchner and G. Levi, editors, *Proceedings of Third International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992. In this Proceedings.
13. H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.
14. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.
15. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 1991. To appear.
16. N. D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 281–288, 1984.
17. G. Levi. Models, Unfolding Rules and Fixpoint Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1649–1665. The MIT Press, Cambridge, Mass., 1988.
18. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.
19. N.D.Jones and H.Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.
20. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.
21. T. Vasak and J. Potter. Characterization of Terminating Logic Programs. In *Proc. Third IEEE Int'l Symp. on Logic Programming*, pages 140–147. IEEE Comp. Soc. Press, 1986.
22. K. Vershaetse and D. De Shreye. Deriving Termination Proofs for Logic Programs, Using Abstract Procedures. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 301–315. The MIT Press, Cambridge, Mass., 1991.

This article was processed using the LaTeX macro package with LLNCS style