Modelling Prolog Control *

Roberto Barbuti, Michael Codish,
Roberto Giacobazzi, Giorgio Levi

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
(barbuti,codish,giaco,levi)@di.unipi.it

Abstract

The goal of this paper is to construct a semantic basis for the abstract interpretation of Prolog programs. Prolog is a well-known logic programming language which applies a depth-first search strategy in order to provide a practical approximation of Horn clause logic. While pure logic programming has clean fixpoint, model-theoretic and operational semantics the situation for Prolog is different. Difficulties in capturing the declarative meaning of Prolog programs have led to various semantic definitions which attempt to encode the search strategy in different mathematical frameworks. However semantic based analyses of Prolog are typically achieved by abstracting the more simple but less precise declarative semantics of pure logic Programs.

We propose instead to model Prolog control in a simple constraint logic language which is presented together with its declarative and operational semantics. This enables us to maintain the usual approach to declarative semantics of logic programs while capturing control aspects such as search strategy and selection rule.

1 Introduction

Standard semantics typically associate programs with entities which capture the essence of "what they do" while abstracting away from details related to the text of the program as well as the control of the execution model. For semantics-based program analysis an enhanced or *"collecting"* semantics which recaptures some of these details is usually

---

*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.

required. After all, the purpose of program analysis is to analyse the *text* of the program with respect to the *control* of its execution model. In general, the fact that collecting semantics can be viewed as uncovering details which the standard semantics has hidden imposes a restriction on the choice of semantic models upon which program analyses can be based.

In the case of logic programs, standard semantics traditionally associate programs with the set of ground atoms which they imply. Program analyses, in contrast, are often required to capture: (1) answer substitutions for a query; and (2) call patterns – that is how particular clauses in the program are used in refutations of a query. It is no coincidence that in most cases practical abstract interpretations of logic programs approximate top-down semantics based on SLD resolution (e.g. [10]). The information concerning control and textual details of a program are more naturally recovered (and collected) from such semantics.

An alternative approach is taken in [11] and further pursued in this paper. Instead of enhancing the semantic basis for analysis to capture control — the program itself is enhanced so that its standard meaning reflects the required control. The key idea is to enhance a program P by a transformation M so that the standard meaning of $M(P)$ reflects the control information required for the analysis of P. So for example, the declarative meaning of a logic program P enhanced by a "*magic set*" transformation [4] is shown to reflect the call patterns of P enabling bottom-up analysis (e.g. [7, 11, 24]). The approach of modelling control has two main advantages: (1) the standard semantics is not required to be extended to a collecting semantics; and (2) any of the alternative semantic definitions for a language are potential choices as a basis for analysis.

In this paper we apply the idea of modelling control to define a declarative semantics for Prolog with the intention of providing a semantic basis for program analysis. The semantics of Prolog is usually specified by encoding its (sequential) control within different frameworks such as dynamic algebras [8], deterministic transition systems [14] or denotational semantics [3, 15, 20]. However, (semantics-based) analyses of Prolog are usually achieved by abstracting the more simple but less precise semantics of pure logic programs. This can lead to problems as illustrated by the following example.

Example 1.1 *Consider the parallel execution of the following Prolog program P where the predicate eq models an equality theory and the predicates q and r are defined elsewhere. A similar program is used by Lloyd (page 59 in [22]) to illustrate the incompleteness of Prolog's search strategy.*

$$p(x, y) \leftarrow eq(x, y), q(x), r(y).$$

$$eq(a, b).$$

$$eq(c, b).$$

$$eq(x, y) \leftarrow eq(y, x).$$

$eq(x, y) \leftarrow eq(x, z), eq(z, y).$

$eq(x, x).$

A compiler may designate the calls $q(x)$ *and* $r(y)$ *in the body of the first clause for parallel execution if it is able to detect (at compile-time) that any possible run-time environemnt does not bind* x *and* y *to terms which share a variable. Semantics-based program analysis is often able to provide such an information (see for example [25]).*

An analysis of P *which is based on a semantics that considers Prolog's left to right computation rule but ignores the depth first search strategy will not enable parallel execution of* $q(x)$ *and* $r(y)$ *due to the clause* $eq(x, x)$ *which introduces sharing to the runtime environment for* x *and* y*. However, Prolog execution of* P *would never consider the clause* $eq(x, x)$ *due to the infinite branch introduced by the depth first search strategy. Furthermore, an analysis of* P *which is based on a Prolog semantics could detect that:*

1. $q(x)$ *and* $r(y)$ *can be executed in parallel; and*
2. *the clause* $eq(x, x)$ *is not reachable, indicating a potential error in the program.*

We propose to model Prolog semantics in a constraint logic language τ which is expressive enough to capture Prolog control yet restricted enough to have a simple declarative semantics. A transformation from Prolog to τ is defined to describe the meaning of a Prolog program P in terms of the declarative semantics of the corresponding τ-program P^τ. The target language τ is strongly related to the *cc* class of concurrent constraint languages [27]. Constraints are used to specify that a clause may be chosen for reduction only if the computation with the preceding (Prolog) clauses terminates. We present a semantics for τ which is open with respect to the interpretation of these (termination) constraints. The meaning of a Prolog program is constructed by composing the meaning of the appropriate τ program (the "logic component") with a corresponding *termination theory* (the "control component"). This approach supports the analysis of Prolog programs based on the abstract interpretation [13] of τ programs consisting of two corresponding components: (1) a standard analysis for the logic component of the program; and (2) a termination analysis for the control component.

An approximate model for the control component (termination) can be provided either by applying techniques of abstract interpretation, as described in [6], or by applying proof procedures as proposed by [26]. It is interesting to note that a precise model for the control component provides a result which reflects the standard Prolog semantics; while the worst approximation (which specifies that all termination conditions are possibly true) reflects the meaning of P as a pure logic program.

The rest of the paper is organized as follows: Section 2 reviews the model of *constraint logic programs* and presents some preliminary definitions and notations. Section 3 demonstrates the basic intuition of our approach via an example. Section 4 introduces the τ language together with its operational and declarative semantics. Section 5 describes how to model depth-first control of Prolog using the τ language. Section

6 discusses the interpretation of the termination conditions. Section 7 describes related work and concludes. The Appendix contains the proofs of the equivalence results presented in Section 4. This paper is an extended version of [5].

2 Preliminaries

In the following we assume familiarity with logic programming theory [1, 22].

Let Σ and Var be denumerable sets of function symbols, and variables respectively; $T(\Sigma)$ and $T(\Sigma, Var)$ denote the corresponding ground and non-ground term algebras respectively. Likewise, if Π denotes a denumerable set of predicate symbols, $\mathcal{A}(\Sigma, \Pi, Var)$ denotes the set of atoms of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi$ and $t_i \in T(\Sigma, Var)$ $(1 \leq i \leq n)$. When clear from the context we will refer to elements of $\mathcal{A}(\Sigma, \Pi, Var)$ as Π-atoms or simply as atoms. Note that $\mathcal{A}(\Sigma, \Pi, \emptyset)$ is the set of ground Π-atoms. The set of finite sequences of atoms is denoted $Atom^*$. The sequence of atoms with first element A followed by the sequence B is denoted $A \mid B$; the concatenation of two sequences B and B' is denoted $B :: B'$. The empty sequence is denoted by Λ. The set of variables that occur in a syntactic object t is denoted $vars(t)$. Tuples of variables and terms are sometimes denoted $\bar{x}$, $\bar{y}$,... and $\bar{t}$, $\bar{s}$,... respectively.

Let $I \subseteq \mathcal{A}(\Sigma, \Pi, Var)$ and $V \subseteq Var$; $A \ll_V I$ denotes that A is a renaming of an element of I which contains no variables in V. Moreover, we extend this notation for any syntactic object. An equation is an expression of the form $t = s$ where $t, s \in T(\Sigma, Var)$. A set of equations E is *unifiable* iff there exists a mapping ϑ from Var into $T(\Sigma, Var)$ (called a *solution*) such that $\forall (t = s) \in E$, $(t\vartheta) \equiv (s\vartheta)$ where $\equiv$ denotes syntactic equivalence and $t\vartheta$ is the term obtained by replacing each variable x in t with $\vartheta(x)$. Solutions apply in the same way to any syntactic object. The empty set of equations is denoted ϵ. The application of a set of equations σ to a syntactic object A is defined by $A\sigma = \{A\vartheta \mid \vartheta \text{ is a solution of } \sigma\}$. Notice that the set $A\sigma$ is closed under instantiation and hence under renaming. The standard ordering on atoms (i.e. *less general* or *more instantiated*) is captured by defining $A \preceq B$ iff there is a unifiable set of equations σ such that $A\epsilon = B\sigma$.

Constraint logic programming (CLP) is a generalization of the pure logic programming paradigm, having similar model-theoretic, fixpoint and operational semantics [19]. One relevant feature is the distinction between testing for satisfiability and computing a solution of a given constraint formula. In CLP, the set of predicate symbols Π is partitioned into $\Pi_{\mathcal{C}}$ and $\Pi_{\mathcal{P}}$. The meaning of the predicates symbols in $\Pi_{\mathcal{P}}$ is given by a program in the usual way. Predicate symbols in $\Pi_{\mathcal{C}}$ are used to build constraints and their meaning is given by a separate *constraint theory* $\mathcal{X}$. $\Pi_{\mathcal{C}}$ contains at least the equality symbol "=". In the following we will denote as $CLP(\mathcal{X})$ a constraint logic programming paradigm based on the constraint theory $\mathcal{X}$. A constraint logic program is a finite set of clauses of the form $H \leftarrow c \,\Box\, \bar{B}$ where H is an atom, c is a constraint formula and $\bar{B}$ is a sequence of atoms in $\mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, Var)$. A constraint formula is a conjunction of atomic formulas in $\mathcal{A}(\Sigma, \Pi_{\mathcal{C}}, \mathit{Var})$. A goal G has the form $c \Box \bar{B}$. A *derivation* of a program P for an initial goal G_0 is a (finite or infinite) sequence of goals. Consecutive goals $G_i = c_i \Box \bar{B}_i$ and $G_{i+1} = c_{i+1} \Box \bar{B}_{i+1}$ in the sequence are related as follows: (1) $c_{i+1} = c_i \wedge \{B = H\} \wedge c$ where $H \leftarrow c \Box \bar{B} \ll_{vars(G_i)} P$, B is an element of $\bar{B}_i$ and $c_i \wedge \{B = H\} \wedge c$ is satisfiable; and (2) $\bar{B}_{i+1}$ is obtained from $\bar{B}_i$ by replacing the selected occurrence of B with $\bar{B}$. The replacement of B by $\bar{B}$ and the formation of c_{i+1} is called a *derivation step*. A computation rule determines (uniquely) for every goal in a derivation, which atom in that goal is selected. The computation of a solution of a constraint is left to a constraint solver, which does not affect the semantics definition of the language. A derivation for a goal G_0 is *successful* if it is finite and ends with a goal $c_n \Box \Lambda$. In this case, c_n is called an *answer constraint* for G_0. In the *CLP* paradigm, at each derivation step for a goal, the constraint part c is checked for satisfiability with respect to a constraint theory $\mathcal{X}$ [19], namely $\mathcal{X} \models (\exists)c$, where $(\exists)c$ denotes the existential closure of c.

The *Herbrand constraint theory*, $\mathcal{H}$, is obtained by choosing $\Pi_{\mathcal{C}} = \{=\}$ interpreted as Clark's equality theory. Herbrand constraints are (systems of) equations between terms in the non-ground term algebra. It is well known that logic programs can be characterised as corresponding *CLP*($\mathcal{H}$) programs (e.g., [12]). In this case the satisfiability of a Herbrand constraint is determined by computing a solution (a set of equations in solved form or *most general unifier*).

An extension of constraint logic programming is provided by the *ask-tell* paradigm [23, 27]. In addition to satisfiability, *validation* (or *entailment*) is introduced. A possibly existentially quantified constraint a is *entailed* by a constraint c iff $\mathcal{X} \models (\forall)(c \Rightarrow a)$ (denoted $\mathcal{X} \models (c \Rightarrow a)$) where $(\forall)c$ denotes the universal closure of c.

An ask-tell clause has the form

$$H \leftarrow \langle a : c \rangle \Box \bar{B}$$

where a and c are constraints (called *ask* and *tell* constraints respectively). In each derivation step ask constraints are checked for entailment, while tell constraints are checked for satisfiability.

Goals have the form $c_i \Box \bar{B}_i$ (as before). A derivation in the ask-tell paradigm is a sequence of goals G_i for $i = 0, 1...$ such that G_{i+1} is obtained from G_i by means of a renamed program clause $H \leftarrow \langle a : c \rangle \Box \bar{B}$ provided that $c \wedge \{H = B\} \wedge c_i$ is satisfiable and a is entailed by $c_i \wedge \{H = B\}$ where B is the selected atom in $\bar{B}_i$. Notice that if $\mathcal{X} \models (c \Rightarrow a)$ then $\mathcal{X} \models (c \wedge c' \Rightarrow a)$ holds for any constraint c'. Tell constraints are then monotonically refined during the derivation, at each step of the computation, the set of possible solutions of the constraint is contained in the set of possible solutions at any prior step. Thus, an ask constraint, entailed at a given step in a computation, will be entailed at all the subsequent steps. For this reason, ask constraints are not collected in the derivation.

This different view of constraint programming, based on ask and tell constraints, leads to a powerful paradigm for concurrent computations (cc languages) in a shared *store* [27]. A store is a constraint representing the global state of the computation. The evaluation of an ask constraint in a given store can have three possible results: (1) the ask constraint is entailed by the store; (2) the negation of the ask constraint is entailed by the store; or (3) neither the ask constraint nor its negation are entailed by the store. Synchronization is achieved through *suspending ask*, namely the process is suspended when the store entails neither the ask constraint nor its negation. A process remains suspended until the store becomes sufficiently updated to either entail the ask constraint or its negation.

In the following we will consider the ask-tell paradigm in a sequential framework. Namely, we do not allow suspension, i.e. if an ask constraint is not entailed by a store, the computation fails. Moreover we will consider a slighty different notion of ask constraints and entailment.

3 The Approach

This section demonstrates the basic idea of our approach via an intuitive example.

Example 3.1 *Consider a Prolog program consisting of two clauses written in a CLP style.*

$C_1 = \ p(\bar{x}) \leftarrow \bar{x} = \bar{t}_1 \ \Box \ q(\bar{t}_2).$

$C_2 = \ p(\bar{x}) \leftarrow \bar{x} = \bar{t}_3 \ \Box \ r(\bar{t}_4).$

and the goal

$G = \ \ \bar{y} = \bar{t} \ \Box \ p(\bar{y}), r(\bar{s}).$

The intended operational semantics can be described by:

1. *The first clause is applied to* $p(\bar{x})$ *if the constraint* $c \equiv \bar{y} = \bar{t} \wedge p(\bar{y}) = p(\bar{x}) \wedge \bar{x} = \bar{t}_1$ *is satisfiable. The success of* G *then depends on the success of the goal* $c \ \Box \ q(\bar{t}_2), r(\bar{s})$.
2. *The second clause is applied if*
   - *(a) the computation of* G *starting with a derivation applying the first clause for* p *terminates (e.g. finitely fails or succeeds); and*
   - *(b) the constraint* $\bar{y} = \bar{t} \wedge p(\bar{y}) = p(\bar{x}) \wedge \bar{x} = \bar{t}_3$ *is satisfiable.*

Informally, the above program could be translated to the following (constrained) one, where the search strategy is coded in an ask-like constraint.

$C_1^\tau = \ p(\bar{x}) \leftarrow \langle true : \bar{x} = \bar{t}_1 \rangle \ \Box \ q(\bar{t}_2).$

$C_2^\tau = \ \ p(\bar{x}) \leftarrow \langle term_p^2(p(\bar{x})) : \bar{x} = \bar{t}_3 \rangle \ \Box \ r(\bar{t}_4).$

The ask constraint true of C_1^τ denotes that C_1^τ is applicable to any goal, while the ask constraint $\mathit{term}_p^2(p(\bar{x}))$ of C_2^τ expresses the fact that the application of C_2^τ depends on termination of the computations for the goal $p(\bar{x})$ starting from clauses C_i^τ with $i < 2$.

Checking for the termination of the atom $p(\bar{x})$ is not sufficient if this atom is part of a non-atomic goal. In such a case, the applicability of a clause to a goal depends on the termination of the computation for the whole goal, starting from the preceding clauses. Thus, the applicability condition for clauses is not limited to the termination of one atom, but it must be extended to the whole goal. When dealing with non-atomic goals, the termination condition has to consider the whole computation, thus, the argument of term_p^2 must be extended at execution time. For instance, with the goal $\bar{y} = \bar{t} \;\Box\; p(\bar{y}), r(\bar{s})$, the ask constraint of clause C_2^τ must become $\mathit{term}_p^2(\bar{y} = \bar{t}, p(\bar{y}) = p(\bar{x}) \;\Box\; p(\bar{x}), r(\bar{s}))$, asking for the termination of the goal $p(\bar{x}), r(\bar{s})$ in a store containing the binding $\bar{x} = \bar{t}$.

To better understand the behaviour of applicability conditions, consider the following:

Example 3.2 *Consider the Prolog program:*

$p(0).$

$p(s(x)).$

$r(x) \leftarrow r(x).$

The second clause can be applied to the goal $p(x)$ because the computation for this goal terminates applying the first clause. However, the second clause is not applicable to the goal $p(x), r(x)$ as the computation starting with the first clause does not terminate for this goal (because of the divergence for $r(0)$).

The computational model of the constraint language introduced in the next section is suitabe to model such a behaviour.

4 The τ language

This section introduces a constraint logic language τ which is applied in the next section to model Prolog control. We present and prove equivalent two semantic definitions for τ. An operational semantics defined in terms of a simple transition system is later used to justify the proposed modelling of Prolog. A declarative semantics is given in terms of an immediate consequence operator and in combination with the transformation proposed in the next section provides a basis for bottom-up abstract interpretations of Prolog programs.

A τ program consists of ask-tell constrained clauses as in the cc paradigm. However in contrast with the cc framework, ask and tell constraints will be interpreted over different algebraic structures:

- Tell constraints are interpreted over the usual Herbrand system, $\mathcal{H}$ [19].
- Ask constraints correspond to semi-decidable tests which are intended to specify termination conditions (applied below to model termination of Prolog goals). These tests are interpreted by an associated *termination theory*, $\mathcal{T}$.

4.1 Syntax

Let Σ, Π and Var be given as above and let $\Pi_{\mathcal{H}} = \{=\}$, $\Pi_{\mathcal{T}}$ and $\Pi_{\mathcal{P}}$ be a partition of Π. We assume an element $true \in \Pi_{\mathcal{T}}$. A finite sequence of $\Pi_{\mathcal{P}}$-atoms is called a $\Pi_{\mathcal{P}}$-*resolvent* or a *resolvent* for short.

Definition 4.1 (goals and constraints)
A tell constraint *or a* store *is a finite set of* $\Pi_{\mathcal{H}}$*-atoms. The set of stores is denoted by Store. An* ask constraint *is an element of the form* $t(g)$ *where* $t \in \Pi_{\mathcal{T}}$ *and* g *is a* $\Pi_{\mathcal{P}}$*-resolvent. Ask constraints of the form* $t(\Lambda)$ *are denoted* t*. A* constraint *is a pair* $\langle ask : tell \rangle$ *consisting of an ask and a tell constraint.*

A goal *is a pair* $c \;\Box\; B$ *consisting of a constraint* c *and a resolvent* B*. The set of goals is denoted by Goal. The empty tell constraint is denoted by "true".*

In the following we will often omit Λ in empty goals.

Definition 4.2 (programs)
A τ program *consists of a finite set of clauses of the form* $H \leftarrow c \;\Box\; B$ *where* H *is a* $\Pi_{\mathcal{P}}$*-atom and* $c \;\Box\; B$ *is a goal.*

Ask and tell constraints are interpreted over corresponding $\mathcal{T}$ and $\mathcal{H}$ theories. $\mathcal{T}$ is a downward closed (i.e. closed under instantiation) set of ask constraints called a *termination theory*. We assume that $true(g) \in \mathcal{T}$ for every resolvent g.

Constraint consistency is defined for a given "environment" specified by a resolvent and a store.

Definition 4.3 (ask and tell consistency)
Let ρ *be a resolvent,* σ *a store and* $\langle ask : tell \rangle$ *a constraint with* $ask = t(g)$*, we write:*

- $\mathcal{H} \models_{\sigma} tell$ *iff* $\sigma \cup tell$ *has a solution,*
- $\mathcal{T} \models_{\sigma}^{\rho} ask$ *iff* $t(g :: \rho)\sigma \subseteq \mathcal{T}$,
- $\langle \mathcal{H}, \mathcal{T} \rangle \models_{\sigma}^{\rho} \langle ask : tell \rangle$ *iff* $\mathcal{H} \models_{\sigma} tell$ *and* $\mathcal{T} \models_{\sigma}^{\rho} ask$

and say that ask and tell are respectively $\mathcal{H}$ *and* $\mathcal{T}$ *consistent for* σ *and* ρ*.*

Example 4.1
Consider a store $\sigma = \{y = s(z)\}$, a resolvent $\rho = p(0)$ and a termination theory $\mathcal{T} = \{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, \mathit{Var})^*\}$ with $\Sigma = \{0, s\}$, $\Pi_{\mathcal{P}} = \{p\}$ and a constraint $c = \langle t(p(y)) : y = s(0)\rangle$. Since $\sigma \cup \{y = s(0)\}$ has a solution and $t(p(y), p(0))\sigma \subseteq \mathcal{T}$, therefore $\langle \mathcal{H}, \mathcal{T}\rangle \models^{\rho}_{\sigma} c$.

Intuitively, $\mathcal{T}$ consistency will be used to model the dynamic extension of applicability conditions for clauses. Given a store σ and a resolvent ρ, $\mathcal{T} \models^{\rho}_{\sigma} t(g)$ guarantees that the atomic goal g extended with the atoms in the resolvent ρ, terminates in the store σ.

4.2 Operational Semantics

An operational semantics for τ is formalised in terms of a transition relation on goals. A pair in the relation corresponds to the reduction of a goal with a renamed clause from the program. The conditions on the tell parts of the goal and clause are the standard ones reviewed in Section 2 (e.g. *CLP($\mathcal{H}$)* [19]). At each step the ask conditions of the chosen clause are tested with respect to a "current" store and resolvent. Ask constraints are not collected since downward closure of $\mathcal{T}$ implies that any test satisfied in a store will be continuously satisfied in subsequent (updated) stores.

Definition 4.4 (transition system)
Let P be a τ program and let $\mathcal{T}$ be a termination theory. Define $\xrightarrow{\mathcal{T}}_P \subseteq \mathit{Goal} \times \mathit{Goal}$ to be the smallest relation such that:

$$\langle \mathit{true} : \mathit{tell}\rangle \;\Box\; A \mid B \quad \xrightarrow{\mathcal{T}}_P \langle \mathit{true} : \mathit{tell} \cup \{H = A\} \cup \mathit{tell}'\rangle \;\Box\; B' :: B$$

if $C = H \leftarrow \langle \mathit{ask}' : \mathit{tell}'\rangle \;\Box\; B' \ll_V P$,
where $V = \mathit{vars}(\langle \mathit{true} : \mathit{tell}\rangle \;\Box\; A \mid B)$, $\mathcal{H} \models_{\mathit{tell}} \{H = A\} \cup \mathit{tell}'$ *and*
$\mathcal{T} \models^{B}_{\mathit{tell} \cup \{H = A\}} \mathit{ask}'$.

The (non-ground) success set of a τ-program is specified (similar to the case of logic programs, e.g. [17]) in terms of the atomic goals of the form $\langle \mathit{true} : \mathit{tell}\rangle \;\Box\; p(\bar{x})$ which have *successful computations* as specified by the transitive closure of the transition relation (given a termination theory $\mathcal{T}$).

Definition 4.5 (success set)
Let P be a τ program and let $\mathcal{T}$ be a termination theory. The success set of P with respect to $\mathcal{T}$ is:

$$\mathcal{S}_{\mathcal{T}}(P) = \left\{ \; p(\bar{x})\sigma \;\middle|\; \langle \mathit{true} : \mathit{tell}\rangle \;\Box\; p(\bar{x}) \xrightarrow{\mathcal{T}}{}^{*}_{P} \langle \mathit{true} : \sigma\rangle \;\Box\; \right\}.$$

Notice that the success set of a program P is a set of atoms closed under instantiation. The operational semantics of a τ program is defined in terms of the *successful computations* for (consistent) initial goals:

Definition 4.6 (operational semantics)
Let P be a τ program and let $\mathcal{T}$ be a termination theory. Define $\mathcal{O}_{\mathcal{T}}(P) : Goal \rightarrow 2^{Store}$ by:

$$\mathcal{O}_{\mathcal{T}}(P) = \lambda\langle ask : tell\rangle \;\Box\; G.\left\{ \sigma \;\middle|\; \begin{array}{l} \langle true : tell\rangle \;\Box\; G \xrightarrow{\mathcal{T}}{}^{*}_{P} \langle true : \sigma\rangle \;\Box \\ \mathcal{H} \models_{\emptyset} tell \;\; and \;\; \mathcal{T} \models^{\Lambda}_{\emptyset} ask \end{array} \right\}.$$

Note that the operational semantics is parametric with respect to a choice of termination theory $\mathcal{T}$. In the following we refer to such semantic definitions as *open with respect to $\mathcal{T}$*.

Example 4.2
Let $\mathcal{T} = \{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, \emptyset)^{*}\}$ be a termination theory $\Pi_{\mathcal{P}} = \{p\}$, $\Sigma = \{0, s\}$, and let P be the following τ-program:

$$p(x) \leftarrow \langle true : x = 0\rangle \;\Box.$$

$$p(x) \leftarrow \langle t(p(x)) : x = s(y)\rangle \;\Box\; p(y).$$

The goal $\langle true : true\rangle \;\Box\; p(s(x))$ cannot be reduced since reduction with the first clause would result in an inconsistent tell constraint while reduction with the second clause is not ask consistent (the termination theory contains only ground atoms). The goal $\langle true : true\rangle \;\Box\; p(s(0))$ has a successful computation:

$$\langle true : true\rangle \;\Box\; p(s(0)) \xrightarrow{\mathcal{T}}_{P} \langle true : y = 0\rangle \;\Box\; p(y) \xrightarrow{\mathcal{T}}_{P} \langle true : y = 0\rangle \;\Box.$$

The success set of P with respect to $\mathcal{T}$ is

$$\{p(0), p(s(0)), p(s(s(0))), \ldots\}.$$

4.3 Declarative Semantics

We now define a declarative semantics for τ programs which is *open* with respect to the interpretation of ask constraints. It is defined in terms of unfoldings of clauses [21] which contain uninterpreted ask conditions. This approach facilitates separation of termination from computation when applied to modelling Prolog control. Unfoldings of clauses with ask and tell constraints result in sequences of constraints [18] which we term *constrained atoms*.

Definition 4.7 (constrained atoms)
A constrained atom, π, *is a formula of the form*

$$H \leftarrow \langle ask_1 : tell_1 \rangle \leftarrow \cdots \leftarrow \langle ask_n : tell_n \rangle$$

where H *is an atom and* $\langle ask_1 : tell_1 \rangle, \ldots, \langle ask_n : tell_n \rangle$ *are constraints.*
We often denote $\pi = H \leftarrow \langle ask_i : tell_i \rangle_{i=1}^{n}$ *and* $tell(\pi) = \bigcup_{i=1}^{n} tell_i$.

The notion of consistency extends naturally to constrained atoms:

Definition 4.8 (consistent constrained atom)
Let σ_0 *be a store,* ρ *a resolvent,* $\mathcal{T}$ *a termination theory and* $\pi = p(\bar{x}) \leftarrow \langle ask_i : tell_i \rangle_{i=1}^{n}$ *a constrained atom. We write:*

- $\mathcal{H} \models_{\sigma_0} \pi$ *iff* $\mathcal{H} \models_{\sigma_0} tell(\pi)$,
- $\mathcal{T} \models_{\sigma_0}^{\rho} \pi$ *iff for* $i = 1..n$, $\mathcal{T} \models_{\sigma_{i-1}}^{\rho} ask_i$ *where* $\sigma_i = \sigma_{i-1} \cup tell_i$,
- $\langle \mathcal{H}, \mathcal{T} \rangle \models_{\sigma_0}^{\rho} \pi$ *iff* $\mathcal{H} \models_{\sigma_0} \pi$ *and* $\mathcal{T} \models_{\sigma_0}^{\rho} \pi$.

Example 4.3
Let $\pi = p(x) \leftarrow \langle t(p(x)) : x = s(y) \rangle \leftarrow \langle t(p(y)) : y = s(z) \rangle \leftarrow \langle t(p(z)) : z = s(w) \rangle$, $\sigma = \{x = sss(0)\}$ and $\mathcal{T} = \{t(p(0)), t(p(s(0))), t(p(ss(0))), \ldots\}$. Then $\langle \mathcal{H}, \mathcal{T} \rangle \models_{\sigma}^{\Lambda} \pi$ because $\mathcal{H} \models_{\sigma} \{x = s(y), y = s(z), z = s(w)\}$ and

$\mathcal{T} \models_{\sigma}^{\Lambda} t(p(x))$,
$\mathcal{T} \models_{\sigma \cup \{x = s(y)\}}^{\Lambda} t(p(y))$ and
$\mathcal{T} \models_{\sigma \cup \{x = s(y), y = s(z)\}}^{\Lambda} t(p(z))$.

The following example demonstrates that consistency of constrained atoms is order dependent:

Example 4.4
Let $\mathcal{T} = \{t(p(f(a)))\}$, $\sigma = \emptyset$ and

$$\pi_1 = p(x) \leftarrow \langle true : x = f(a) \rangle \leftarrow \langle t(p(x)) : true \rangle,$$

$$\pi_2 = p(x) \leftarrow \langle t(p(x)) : true \rangle \leftarrow \langle true : x = f(a) \rangle;$$

then $\langle \mathcal{H}, \mathcal{T} \rangle \models_{\sigma}^{\Lambda} \pi_1$ but $\langle \mathcal{H}, \mathcal{T} \rangle \not\models_{\sigma}^{\Lambda} \pi_2$.

The semantic objects will consist of (equivalence classes) of *consistent* constrained atoms. In the following we will often denote by η a sequence of constraints $\langle t_i(g_i) : tell_i \rangle_{i=1}^{n}$. Intuitively, a constrained atom, $p \leftarrow \eta$, represents a sequence of ask and tell constraints corresponding to a possible computation for p. A computed answer constraint is returned for that atom only if all the constraints in the sequence are consistent with respect to the current store. As the atom can be used in a more complex goal all the ask constraints in η must be augmented to be sensitive to the context in which they are used. This motivates the following notation: let η be the sequence $\langle t_i(g_i) : tell_i \rangle_{i=1}^{n}$, for any resolvent ρ, we denote by $\eta[\rho]$ the (augmented) sequence $\langle t_i(g_i :: \rho) : tell_i \rangle_{i=1}^{n}$.

Definition 4.9 (equivalence of constrained atoms)
Let π_1 *and* π_2 *be constrained atoms;* $\pi_1 \sim \pi_2$ *iff for every termination theory* $\mathcal{T}$, *store* σ *and resolvent* ρ *there are renamings* $\pi'_1 = p(\bar{t}_1) \leftarrow \eta_1$, $\pi'_2 = p(\bar{t}_2) \leftarrow \eta_2$ *of* π_1 *and* π_2 *such that* $vars(\sigma) \cap vars(\pi'_i) \subseteq vars(\bar{t}_i)$, *for* $i = 1, 2$ *and* $\langle \mathcal{H}, \mathcal{T} \rangle \models^{\rho}_{\sigma} \pi'_1 \Leftrightarrow \langle \mathcal{H}, \mathcal{T} \rangle \models^{\rho}_{\sigma} \pi'_2$.

The equivalence of constrained atoms, $\pi'_1 = p(\bar{t}_1) \leftarrow \eta_1$ and $\pi'_2 = p(\bar{t}_2) \leftarrow \eta_2$ is specified with respect to stores which do not affect the local variables of η_1 and η_2 (that is the local variables of possible computations for $p(\bar{t}_1)$ and $p(\bar{t}_2)$).

Note that for any constrained atom $\pi = p(\bar{t}) \leftarrow \eta$ there is an equivalent constrained atom of the form $\pi' = p(\bar{x}) \leftarrow \eta'$ (in which $\bar{x}$ are distinct variables not occurring in π) defined by taking new variables $\bar{x}$ and $\eta' = \langle true : \bar{x} = \bar{t} \rangle \leftarrow \eta$. So in the following, any constrained atom may be denoted as $p(\bar{x}) \leftarrow \eta$.

Definition 4.10 (B^{τ}_{P} base)
Let P *be a* τ*-program. The* base of interpretations *for* P, *denoted* B^{τ}_{P}, *is the set of equivalence classes of constrained atoms over the alphabet of* P.

Definition 4.11 (interpretation)
An interpretation *is any subset of* B^{τ}_{P}.

In the following we give a fixpoint characterization of the semantics in terms of a continuous immediate consequence operator T^{τ}_{P}.

Definition 4.12
Let P *be a* τ*-program. The mapping* $T^{\tau}_{P} : 2^{B^{\tau}_{P}} \rightarrow 2^{B^{\tau}_{P}}$ *is defined by:*

$$T^{\tau}_{P}(I) = \left\{ [H \leftarrow \bar{\eta}]_{\sim} \;\middle|\; \begin{array}{l} C = H \leftarrow c \,\Box\, B_1, \ldots, B_n \in P, \\ for\ i = 1..n \\ \quad \rho_i = B_{i+1}, \ldots, B_n, \\ \quad \pi_i = H_i \leftarrow \eta_i,\ [\pi_i]_{\sim} \in I, \\ \quad \bar{\eta}_i = \langle true : H_i = B_i \rangle \leftarrow \eta_i[\rho_i], \\ C, \pi_1, \ldots, \pi_n \text{ are renamed apart}, \\ \bar{\eta} = c \leftarrow \bar{\eta}_1 \leftarrow \ldots \leftarrow \bar{\eta}_n,\ \mathcal{H} \models_{\emptyset} H \leftarrow \bar{\eta} \end{array} \right\}.$$

Note that the unfolding of an atom B_i in the body of the clause C with respect to a constrained atom $H_i \leftarrow \eta_i$ keeps track of the current resolvent. Thus, the resulting constrained atom $H \leftarrow \bar{\eta}$ represents a computation for the atomic goal H obtained by applying to a clause the computations for the atoms in the body. To construct the new applicability tests, all the termination tests in each computation are enhanced with the resolvent corresponding to the remaining part of the goal to be evaluated. The ask constraints in the sequence η_i, in fact, are enhanced with the current resolvent $\rho_i = B_{i+1}, ..., B_n$ ($\eta_i[\rho_i]$).

Proposition 4.1
T_P^τ is well defined and continuous.

PROOF. The proof is standard [17, 22]. □

Definition 4.13 (fixpoint semantics)
The fixpoint semantics *for a τ-program P is defined by $\mathcal{F}(P) = \mathit{lfp}(T_P^\tau) = T_P^\tau \uparrow \omega$.*

Example 4.5
Consider the following τ programs P_1^τ and P_2^τ where $\Pi_{\mathcal{T}} = \{t, \mathit{true}\}$:

$$\begin{array}{ll} P_1^\tau: & p(x) \leftarrow \langle \mathit{true} : \mathit{true} \rangle \, \square. \\ & p(x) \leftarrow \langle t(p(x)) : x = s(y) \rangle \, \square \, p(y). \end{array}$$

$$\begin{array}{ll} P_2^\tau: & p(x) \leftarrow \langle t(p(x)) : \mathit{true} \rangle \, \square. \\ & p(x) \leftarrow \langle \mathit{true} : x = s(y) \rangle \, \square \, p(y). \end{array}$$

The fixpoint semantics of P_1^τ and P_2^τ are:

$$\mathcal{F}(P_1^\tau) = \left\{ \begin{array}{l} p(x) \leftarrow \langle \mathit{true} : \mathit{true} \rangle, \\ p(x) \leftarrow \langle t(p(x)) : x = s(y) \rangle, \\ p(x) \leftarrow \langle t(p(x)) : x = s(y) \rangle \leftarrow \langle t(p(y)) : y = s(z) \rangle, \\ p(x) \leftarrow \langle t(p(x)) : x = s(y) \rangle \leftarrow \langle t(p(y)) : y = s(z) \rangle \\ \qquad\qquad\qquad\qquad\qquad \leftarrow \langle t(p(z)) : z = s(w) \rangle, \\ \ldots \end{array} \right\}$$

$$\mathcal{F}(P_2^\tau) = \left\{ \begin{array}{l} p(x) \leftarrow \langle t(p(x)) : \mathit{true} \rangle, \\ p(x) \leftarrow \langle \mathit{true} : x = s(y) \rangle \leftarrow \langle t(p(y)) : \mathit{true} \rangle, \\ p(x) \leftarrow \langle \mathit{true} : x = s(y) \rangle \leftarrow \langle \mathit{true} : y = s(z) \rangle \leftarrow \langle t(p(z)) : \mathit{true} \rangle, \\ \ldots \end{array} \right\}$$

Observe that for the previous example, in the evaluation of the $\mathit{lfp}(T_P^\tau)$, the resolvents ρ_i are always empty as the clauses of P have unit or empty bodies. Example 5.3 illustrates the more general case.

In Example 4.6 (below) we show how to compose an interpretation for the ask predicate t with $\mathcal{F}(P_1^\tau)$ and $\mathcal{F}(P_2^\tau)$ to provide success sets for P_1^τ and P_2^τ. The choice of P_1^τ and P_2^τ is further motivated by Example 5.3.

4.4 Equivalence of Operational and Declarative Semantics

By composing the *open* declarative semantics $\mathcal{F}(P)$ of a program P with a termination theory $\mathcal{T}$ we can derive the corresponding success set $\mathcal{S}_{\mathcal{T}}(P)$.

Definition 4.14
Let I be an interpretation, ρ a resolvent and $\mathcal{T}$ a termination theory. The composition $I[\mathcal{T}]$ of I and $\mathcal{T}$ with respect to ρ is defined by:

$$I[\mathcal{T}](\rho) = \left\{ p(\bar{x})\sigma \;\middle|\; \begin{array}{l} [\pi]_{\sim} = [p(\bar{x}) \leftarrow \eta]_{\sim} \in I, \\ \exists \theta : \ vars(\theta) \cap vars(\pi) \subseteq \bar{x}, \\ \mathcal{T} \models^{\rho}_{\theta} \pi, \ \sigma = tell(\pi) \cup \theta \end{array} \right\}.$$

Note that $I[\mathcal{T}](\rho)$ contains the atoms obtained by the composition of the constrained atoms in I with a termination theory $\mathcal{T}$ and a resolvent ρ. Given $\pi = p(\bar{x}) \leftarrow \eta \in I$, if the sequence η is consistent in a store θ with respect to $\mathcal{T}$ and ρ and $\sigma = tell(\pi) \cup \theta$ then π contributes $p(\bar{x})\sigma$ to this composition.

Proposition 4.2 (monotonicity of composition)
Let I be an interpretation. If $\mathcal{T}_1 \subseteq \mathcal{T}_2$ then for each resolvent ρ, $I[\mathcal{T}_1](\rho) \subseteq I[\mathcal{T}_2](\rho)$.

PROOF. The proof is a straightforward application of the definitions. □

Example 4.6
Consider the programs P_1^{τ} and P_2^{τ} from Example 4.5 and let

$$\mathcal{T}_1 = \{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, Var)^*\} \text{ and } \mathcal{T}_2 = \{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, \emptyset)^*\}$$

where $\Sigma = \{0, s\}$ and $\Pi_{\mathcal{P}} = \{p\}$. Then

$$\mathcal{F}(P_1^{\tau})[\mathcal{T}_1](\Lambda) = \{p(t) \mid t \in T(\Sigma, Var)\} \text{ and}$$

$$\mathcal{F}(P_2^{\tau})[\mathcal{T}_2](\Lambda) = \{p(t) \mid t \in T(\Sigma)\}.$$

The following theorem states the soundness and completeness of the fixpoint semantics with respect to the operational notion of success set.

Theorem 4.3 (soundness and completeness)
Let P be τ-program and $\mathcal{T}$ be a termination theory. Then $\mathcal{S}_{\mathcal{T}}(P) = \mathcal{F}(P)[\mathcal{T}](\Lambda)$.

PROOF. See the Appendix. □

Strong equivalence shows the equivalence between the operational semantics of a program and the answers which can be computed from its fixpoint semantics for arbitrary goals:

Definition 4.15 (answers for generic goals)
Let I be an interpretation and let $\mathcal{T}$ be a termination theory. The set of answers *for a goal*

$$G = \langle ask : tell \rangle \, \Box p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$$

in I for $\mathcal{T}$ is defined by:

$$ans_I^{\mathcal{T}}(G) = \left\{ \sigma \;\middle|\; \begin{array}{l} \mathcal{T} \models_{\emptyset}^{\Lambda} ask \text{ and for } i = 1..n : \\ \quad [\pi_i]_{\sim} = [p_i(\bar{x}_i) \leftarrow \eta_i]_{\sim} \in I, \\ \quad \pi_1, ..., \pi_n, G \text{ are renamed apart}, \\ \quad \rho_i = p_{i+1}(\bar{t}_{i+1}), ..., p_n(\bar{t}_n), \\ \quad \sigma_0 = tell \text{ and } \sigma_i = \sigma_{i-1} \cup \{\bar{x}_i = \bar{t}_i\} \cup tell(\pi_i), \\ \quad \mathcal{T} \models_{\sigma_{i-1} \cup \{\bar{x}_i = \bar{t}_i\}}^{\rho_i} \pi_i, \ \mathcal{H} \models_{\emptyset} \sigma_n, \\ \sigma = \sigma_n \end{array} \right\}.$$

Informally, the answers for $G = \langle ask : tell \rangle \, \Box p_1(\bar{t}_1), ..., p_n(\bar{t}_n)$ with respect to I and $\mathcal{T}$ are obtained in the following way: for each atom $p_i(\bar{t}_i)$ in G a constrained atom, $\pi_i = p_i(\bar{x}_i) \leftarrow \eta_i$, is extracted from I. Each η_i is checked for consistency with respect to the termination theory $\mathcal{T}$, the store resulting from processing the atoms in positions $1, ..., i-1$ enhanced with the bindings $\bar{x}_i = \bar{t}_i$ and the resolvent given by the sequence of atoms in positions $i+1, ..., n$. If a η_i is found consistent then $\{\bar{x}_i = \bar{t}_i\} \cup tell(\pi_i)$ is added to the current store to continue the process until $p_n(\bar{t}_n)$.

The following theorem states the equivalence between the concept of answers for a goal and the oparational semantics.

Theorem 4.4 (strong equivalence)
Let P be τ-program, $\mathcal{T}$ a termination theory and $G = \langle ask : tell \rangle \, \Box \, B_1, ..., B_n$ be a goal. Then $\sigma' \in ans_{\mathcal{F}(P)}^{\mathcal{T}}(G)$ iff *$\sigma \in \mathcal{O}_{\mathcal{T}}(P)(G)$ and $G\sigma = G\sigma'$.*

PROOF. See the Appendix. □

For a given a termination theory, $\mathcal{F}(P)$ may contain possibly inconsistent sequences (i.e. constrained atoms which cannot be satisfied). The "pruning" is operated by interpreting $\mathcal{F}(P)$ in the termination theory.

5 Modelling Prolog in τ

The basic idea applied in this section is to specify in a "declarative" way the condition upon which a Prolog clause might eventually be chosen under a depth-first strategy. Namely, that there is no infinite path to the left in the corresponding SLD tree. This idea is first applied to model Prolog control in τ. A transformation which associates a Prolog program P with a corresponding τ program P^{τ} is introduced. The same idea is then applied to define an operational semantics for Prolog which is used to argue the correctness of this transformation.

We assume that Prolog evaluation proceeds until all answers are generated and that Prolog evaluation is sound. A Prolog program is viewed as a set of predicates, each predicate consisting of an ordered set of clauses. Prolog goals are sequences of atoms. Let us return to Example 3.1 where the operational semantics of the Prolog program P:

$$p(\bar{t}_1) \leftarrow q(\bar{t}_2).$$

$$p(\bar{t}_3) \leftarrow r(\bar{t}_4).$$

is informally captured by the τ program P^τ

$$p(\bar{x}) \leftarrow \langle true : \bar{x} = \bar{t}_1 \rangle \ \Box \ q(\bar{t}_2).$$

$$p(\bar{x}) \leftarrow \langle term_p^2(p(\bar{x})) : \bar{x} = \bar{t}_3 \rangle \ \Box \ r(\bar{t}_4).$$

The claim is that given the *intended termination theory* of P, the τ-program P^τ has the same meaning as the Prolog program P. The following definitions specify the transformation from Prolog to τ and what we mean by the *intended termination theory.*

Definition 5.1 (transformation)
Let P be a Prolog program. For each predicate $\{C_1, \ldots, C_m\}$ in P we generate from the clause $C_i = p(\bar{t}) \leftarrow Body$ a corresponding τ clause:

$$C_i^\tau = p(\bar{x}) \leftarrow \langle term_p^i(p(\bar{x})) : \bar{x} = \bar{t} \rangle \ \Box \ Body.$$

The corresponding τ program P^τ is then defined by collecting the respective C_i^τ clauses for all the predicates in P:

$$P^\tau = \left\{ C_i^\tau \;\middle|\; \begin{array}{l} \{C_1, \ldots, C_n\} \in P, \\ i \in \{1, \ldots, n\} \end{array} \right\}.$$

The $\Pi_{\mathcal{T}}$ predicates $term_p^i(p(\bar{x}))$ in the above definition correspond to tests which are intended to enable τ reduction of a goal with a clause C_i^τ only in case Prolog computation of the corresponding goal starting with the preceding clauses for that predicate (i.e. $\{C_1, \ldots, C_{i-1}\}$) terminates. Note that under this interpretation $term_p^1$ is vacuously true.

By SLD reduction we mean usual resolution with a left-to-right selection rule [22]. An SLD reduction is captured by:

Definition 5.2 (resolve)
Let Clause denote the set of Horn clauses. Define: $resolve : Atom^* \times Clause \longrightarrow 2^{Atom^*}$ by:

$$resolve(G, H \leftarrow Body) = \begin{cases} (Body :: B)\{H = A\} & \text{if } G = A \mid B, \\ \emptyset & \text{otherwise.} \end{cases}$$

The following definition is motivated by the observation (e.g. [2]) that when reasoning about termination of all solutions for a goal with a Prolog program, it is possible to ignore depth-first search and hence to consider the corresponding SLD tree (with a left-to-right selection rule). The proof of this observation is straightforward [28].

Definition 5.3 (intended termination theory)
Let P be a Prolog program. The intended termination theory for P is defined by:

$$\mathcal{T}_{Prolog}(P) \;=\; \left\{ term_p^i(g) \;\middle|\; \begin{array}{l} \{C_1, \ldots, C_n\} \text{ defines } p \text{ in } P \text{ and} \\ \text{for } 1 \leq i \leq n : \\ \quad \text{if } g' \in \bigcup\limits_{j<i} resolve(g, C_j) \text{ then} \\ \quad \text{the SLD tree for } P \text{ and } g' \text{ is finite} \end{array} \right\}.$$

Theorem 5.1
The intended termination theory for a Prolog program P is a termination theory (i.e. closed under instantiation).

Proof. The proof follows straightforwardly from the results in [28]. □

In the next section we discuss how to approximate the intended termination theory for a Prolog program. Meanwhile, we claim that given the intended termination theory for a Prolog program, the corresponding τ program captures the operational meaning of P. To formalise this we introduce an operational semantics for Prolog. We will need the following:

Definition 5.4 (finite-transitive)
A relation $R \subseteq X \times X$ on a set X is finite-transitive *for $x \in X$ iff the multi-set $\{x' \mid xR^*x'\}$ is finite.*

The following definition specifies a transition relation for logic programs with a left-to-right selection rule (but ignores the depth-first search of Prolog). This definition is then applied to specify the transition relation for Prolog. The implicit use of stores instead of substitutions is not essential.

Definition 5.5 (transition relation for SLD)
Let P be a logic program; $\xrightarrow{sld}_P \subseteq Atom^* \times Atom^*$ is the relation defined by:

$$g \xrightarrow{sld}_P g' \quad \Leftrightarrow \quad g' \in \bigcup_{C \in P} resolve(g, C).$$

We denote by $pred_P(A)$ the predicate (i.e. ordered set of clauses) for atom A in a Prolog program P.

Definition 5.6 (transition relation for Prolog)
Let P *be a Prolog program;* $\xrightarrow{prolog}_P \subseteq Atom^* \times Atom^*$ *is the smallest relation such that*

$$A \mid B \xrightarrow{prolog}_P G$$

if $pred_P(A) = \{C_1, \ldots, C_n\}$*;* $1 \leq i \leq n$*;* $G \in resolve(A \mid B, C_i)$*; and*
$\forall G' \in \bigcup_{j<i} resolve(A \mid B, C_j)$ *the relation* $\xrightarrow{sld}_P$ *is finite-transitive for* G'*.*

The success set of a Prolog program P is captured in terms of the transitive closure of $\xrightarrow{prolog}_P$.

Definition 5.7 (Prolog success set)
The Prolog success set is $\mathcal{S}_{Prolog} : Prolog \to 2^{Atom}$ *such that:*

$$\mathcal{S}_{Prolog}(P) = \left\{ A \in Atom \,\middle|\, A \xrightarrow{prolog}{}^{*}_{P} \Lambda \right\}.$$

Theorem 5.2
Let P *be a Prolog program and* P^τ *the corresponding* τ *program. Then:*

$$\mathcal{S}_{Prolog}(P) = \mathcal{F}(P^\tau)[\mathcal{T}_{Prolog}(P)](\Lambda).$$

Proof. By Theorem 4.3, $\mathcal{F}(P^\tau)[\mathcal{T}_{Prolog}(P)](\Lambda) = \mathcal{S}_{\mathcal{T}_{Prolog}(P)}(P^\tau)$. We have to prove that:

$$\mathcal{S}_{Prolog}(P) = \mathcal{S}_{\mathcal{T}_{Prolog}(P)}(P^\tau).$$

The proof argues the correspondence between the respective operational behaviours of P and of P^τgiven the corresponding intended termination theory. Let $G = p(\bar{t}) \mid \bar{B}$ be a Prolog goal.

$$G \xrightarrow{prolog}_P G'; \qquad \text{iff}$$

$$\begin{array}{l} \exists C_i \in pred_P(p(\bar{t})) = \{C_1, ..., C_m\} \text{ such that } G' \in resolve(G, C_i) \\ \text{and } \forall G'' \in \bigcup_{j<i} resolve(G, C_j),\ G'' \text{ has a finite SLD tree;} \end{array} \qquad \text{iff}$$

$$\begin{array}{l} \exists C_i \in pred_P(p(\bar{t})) = \{C_1, ..., C_m\},\ C_i = p(\bar{s}) \ :-\ Body, \\ \text{such that } G' \in resolve(G, C_i) \text{ and } term_p^i(G) \in \mathcal{T}_{Prolog}(P); \end{array} \qquad \text{iff}$$

$$\begin{array}{l} \exists C_i^\tau = p(\bar{x}) \ :-\ \langle term_p^i(p(\bar{x})) : \bar{x} = \bar{s} \rangle \ \Box\ Body \in P^\tau, \\ term_p^i(p(\bar{t}) \mid \bar{B}) \in \mathcal{T}_{Prolog}(P); \end{array} \qquad \text{iff}$$

$$\begin{array}{l} \exists C_i^\tau = p(\bar{x}) \ :-\ \langle term_p^i(p(\bar{x})) : \bar{x} = \bar{s} \rangle \ \Box\ Body \in P^\tau, \\ \mathcal{T}_{Prolog}(P) \models^{\bar{B}}_{\bar{x}=\bar{t}} term_p^i(p(\bar{x})); \end{array} \qquad \text{iff}$$

$$\langle \mathit{true} : \mathit{true} \rangle \; \Box \; p(\bar{t}) \mid \bar{B} \xrightarrow{\mathcal{T}}_{P^{\tau}} \langle \mathit{true} : \{\bar{x} = \bar{t}, \bar{x} = \bar{s}\} \rangle \; \Box \; \mathit{Body} :: \bar{B}.$$

Notice that $\langle \mathit{true} : \sigma \rangle \; \Box \; G \xrightarrow{\mathcal{T}}_{P^{\tau}} \langle \mathit{true} : \sigma' \rangle \; \Box \; G'$ iff for each $G'' \in G\sigma$: $\langle \mathit{true} : \mathit{true} \rangle \; \Box \; G'' \xrightarrow{\mathcal{T}}_{P^{\tau}} \langle \mathit{true} : \sigma'' \rangle \; \Box \; G'''$ and $G'\sigma' = G'''\sigma''$. The theorem follows by a straightforward inductive argument. $\Box$

In the following we consider the success set of pure logic programs as defined in [17], where the standard approach is enhanced by introducing variables into the semantics objects.

Proposition 5.3
Let P be a Prolog program and $\mathcal{T}_{lp}$ denote a termination theory which for every predicate p and corresponding i^{th} clause contains $\mathit{term}_p^i(g)$ for every resolvent g. $\mathcal{F}(P^{\tau})[\mathcal{T}_{lp}](\Lambda)$ is the success set of P interpreted as a logic program, closed under instantiation.

PROOF. By the definition

$$\mathcal{S}_{\mathcal{T}_{lp}}(P^{\tau}) = \left\{ \; p(\bar{x})\sigma \;\middle|\; \langle \mathit{true} : \theta \rangle \; \Box \; p(\bar{x}) \overset{\mathcal{T}_{lp}}{\rightarrow}{}^{*}_{P^{\tau}} \; \langle \mathit{true} : \sigma \rangle \; \Box . \; \right\}.$$

Let

$$P'^{\tau} = \left\{ \; p(\bar{x}) \leftarrow \langle \mathit{true} : \sigma' \rangle \; \Box \; \mathit{Body} \;\middle|\; \mathit{pred}_P(p(\bar{x} \right.$$