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)@dipisa.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.

*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.

In proceedings of ACM POPL'92.

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 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. [9]). 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 [10] 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.[5, 10, 20]). 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 [6], deterministic transition systems [12] or denotational semantics [17, 3, 13]. However, (semantics-based) analyses of Prolog are usually achieved by abstracting the more simple but less precise semantics of pure logic programs.

Unfortunately, it is not straightforward to encode Prolog control directly in terms of pure logic programs. Instead, we propose to model Prolog control 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 an instance of the *cc* class of concurrent constraint languages [22]. Constraints are used to specify that a clause may be chosen for reduction only if 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 [11] 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 termination can be provided either by applying techniques of abstract interpretation or by applying proof procedures. It is interesting to note that a precise model for termination provides a result which reflects the standard Prolog control; 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 presents briefly some preliminary definitions and notations. Section 3 introduces the τ language together with operational and declarative semantics. Section 4 describes how to model depth first control of Prolog using the τ language. Section 5 discusses the interpretation of the termination conditions. Section 6 describes related work and concludes.

Full 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 constraint logic programming (CLP) [16] and logic programming theory [1, 19].

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 $p(t_1, \dots, t_n)$ 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. The set of all 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 $vars(t)$. 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. Let $\mathcal{H}$ denote the standard Herbrand constraint system of equations on terms. 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 renamings. 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 $A\epsilon = B\sigma$.

3 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 the basis for bottom-up abstract interpretations of Prolog programs.

A τ program consists of ask-tell constrained clauses as in the cc paradigm [22]. 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, $\mathcal{H}$ [16].
- 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*, $\mathcal{T}$.

3.1 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 3.1 (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".*

Definition 3.2 (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 3.3 (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,*
- $\mathcal{T} \models^\rho_\sigma ask$ *iff* $t(g :: \rho)\sigma \subseteq \mathcal{T}$,
- $\langle \mathcal{H}, \mathcal{T} \rangle \models^\rho_\sigma \langle ask : tell \rangle$ *iff* $\mathcal{H} \models_\sigma tell$ *and* $\mathcal{T} \models^\rho_\sigma ask$

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

Example 3.1
Consider a store $\sigma = \{y = s(z)\}$, a resolvent Λ and a termination theory $\mathcal{T} = \{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, Var)^*\}$ 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 $t(p(y))\sigma \subseteq \mathcal{T}$, therefore $\langle \mathcal{H}, \mathcal{T} \rangle \models^\Lambda_\sigma c$.

3.2 Operational Semantics

An operational semantics for τ is given 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 (e.g. $CLP(\mathcal{H})$ [16]). 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 3.4 (transition system)
Let P *be a* τ *program and let* $\mathcal{T}$ *be a termination theory. Define* $\xrightarrow{\mathcal{T}}_P \subseteq Goal \times Goal$ *to be the smallest relation such that:*

$$\langle true : tell \rangle \,\Box\, A \mid B \xrightarrow{\mathcal{T}}_P$$

$$\langle true : tell \cup \{H = A\} \cup tell' \rangle \,\Box\, B' :: B$$

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

The (non-ground) success set of a τ program is specified (similar to the case of logic programs, e.g. [15]) in terms of the atomic goals of the form $\langle true : true \rangle \,\Box\, p(\bar{x})$ which have *successful computations* as specified by the transitive closure of the transition relation (given a termination theory $\mathcal{T}$).

Definition 3.5 (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:*

$$\mathcal{S}_{\mathcal{T}}(P) =$$

$$\left\{ p(\bar{t})\sigma \;\middle|\; \langle true : true \rangle \,\Box\, p(\bar{t}) \xrightarrow{\mathcal{T}}{}^*_P \langle true : \sigma \rangle \,\Box\, \right\}.$$

The operational semantics of a τ program is defined in terms of the *successful computations* for (consistent) initial goals:

Definition 3.6 (operational semantics)
Let P *be a* τ *program and let* $\mathcal{T}$ *be a termination theory. Define* $\mathcal{O}_{\mathcal{T}}(P) : Goal \to 2^{Store}$ *by:*

$$\mathcal{O}_{\mathcal{T}}(P) = \lambda \langle ask : tell \rangle \,\Box\, G.$$

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

Example 3.2
Let $\mathcal{T}$ be a termination theory $\{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, \emptyset)^*\}$, $\Pi_{\mathcal{P}} = \{p\}$, $\Sigma = \{0, s\}$, and let P be the following τ program:

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

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

The goal $\langle true : true \rangle \;\square\; p(s(x))$ 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 goal

$$\langle true : true \rangle \;\square\; p(s(0))$$

has a successful computation:

$$\langle true : true \rangle \;\square\; p(s(0)) \xrightarrow{\mathcal{T}}_{P}$$

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

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

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

3.3 Declarative semantics

We now define a declarative semantics for τ programs which is *open* [8] with respect to the interpretation of ask constraints. It is defined in terms of unfoldings of clauses [18] 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 which we term *constrained atoms.*

Definition 3.7 (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 3.8 (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* $\mathcal{H} \models_{\sigma_0} tell(\pi)$,
- $\mathcal{T} \models^{\rho}_{\sigma_0} \pi$ *iff for* $i = 1..n$, $\mathcal{T} \models^{\rho}_{\sigma_{i-1}} ask_i$ *where* $\sigma_i = \sigma_{i-1} \cup tell_i$,
- $\langle \mathcal{H}, \mathcal{T} \rangle \models^{\rho}_{\sigma_0} \pi$ *iff* $\mathcal{H} \models_{\sigma_0} \pi$ *and* $\mathcal{T} \models^{\rho}_{\sigma_0} \pi$.

Example 3.3
Let $\pi = p(x) \leftarrow \langle t(p(x)) : x = s(y) \rangle \leftarrow \langle t(p(y)) : y = s(z) \rangle \leftarrow \langle t(p(z)) : z = s(w) \rangle$, $\quad \sigma = \{x = sss(0)\}$ and $\mathcal{T} = \{t(p(0)), t(p(s(0))), t(p(ss(0))), \ldots\}$. Then $\langle \mathcal{H}, \mathcal{T} \rangle \models^{\Lambda}_{\sigma} \pi$ because $\mathcal{H} \models_{\sigma} \{x = s(y), y = s(z), z = s(w)\}$ and

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

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

Example 3.4 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 $\langle \mathcal{H}, \mathcal{T} \rangle \models^{\Lambda}_{\sigma} \pi_1$ but $\langle \mathcal{H}, \mathcal{T} \rangle \not\models^{\Lambda}_{\sigma} \pi_2$.

The semantic objects will consist of (equivalence classes) of *consistent* constrained atoms. In the following we will often denote by η any sequence of constraints $\langle t_i(g_i) : tell_i \rangle_{i=1}^{n}$ and by $\eta[\rho]$ the sequence $\langle t_i(g_i :: \rho) : tell_i \rangle_{i=1}^{n}$.

Definition 3.9 (equiv. 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 is a renaming* π_2' *of* π_2 *such that*

$$\langle \mathcal{H}, \mathcal{T} \rangle \models^{\rho}_{\sigma} \pi_1 \Leftrightarrow \langle \mathcal{H}, \mathcal{T} \rangle \models^{\rho}_{\sigma} \pi_2'.$$

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 3.10 (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 3.11 (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 τ-consequence operator T^{τ}_{P}.

Definition 3.12
Let P *be a* τ*-program. The mapping*

$$T_P^\tau : 2^{B_P^\tau} \rightarrow 2^{B_P^\tau}$$

is defined by:

$T_P^\tau(I) =$

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

Proposition 3.1
T_P^τ is well defined and continuous.

Definition 3.13 (Fixpoint semantics)
The fixpoint semantics *for a τ-program P is defined by $\mathcal{F}(P) = lfp(T_P^\tau) = T_P^\tau \uparrow \omega$.*

Example 3.5 Consider the following τ programs P_1^τ and P_2^τ where $\Pi_\mathcal{T} = \{t\}$:

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

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

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

$\mathcal{F}(P_1^\tau) =$

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

$\mathcal{F}(P_2^\tau) =$

$$\left\{ \begin{array}{l} p(x) \leftarrow \langle t(p(x)) : true \rangle, \\ p(x) \leftarrow \langle true : x = s(y) \rangle \leftarrow \langle t(p(y)) : true \rangle, \\ p(x) \leftarrow \langle true : x = s(y) \rangle \leftarrow \langle true : y = s(z) \rangle \\ \qquad \leftarrow \langle t(p(z)) : true \rangle, \\ \ldots \end{array} \right\}$$

In Example 3.6 (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 4.3.

3.4 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 3.14 (composition)
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:

$$I[\mathcal{T}](\rho) = \left\{ p(\bar{x})\sigma \;\middle|\; \begin{array}{l} [\pi]_\sim = [p(\bar{x}) \leftarrow \eta]_\sim \in I, \\ \bar{t} \in T(\Sigma, Var)^*, \\ \mathcal{T} \models^\rho_{\bar{x}=\bar{t}} \pi, \\ \sigma = tell(\pi) \cup \{\bar{x} = \bar{t}\} \end{array} \right\}.$$

Proposition 3.2 (monotonicity of composition)

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

Example 3.6 Consider the programs P_1^τ and P_2^τ from Example 3.5 and let $\mathcal{T}_1 = \{t(g) \mid g \in \mathcal{A}(\Sigma, \Pi_\mathcal{P}, Var)^*\}$ 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 3.3 (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. [(comment)]
The proof of $(\subseteq)$ is by induction on the length of the refutation while the proof of $(\supseteq)$ is by induction on the power of T_P^τ. Both apply the sequential (left-to-right) composition of the operational semantics:

for a goal $\langle true : \sigma_0 \rangle \;\Box\; B_1, \ldots, B_k$

$$\langle true : \sigma_0 \rangle \;\Box\; B_1, \ldots, B_k \xrightarrow{\mathcal{T}}{}^*_P \langle true : \sigma_k \rangle \;\Box \qquad \text{iff}$$

$$\bigwedge_{i=1}^{k} \langle true : \sigma_{i-1} \rangle \;\Box\; B_i, \ldots, B_k \xrightarrow{\mathcal{T}}{}^*_P$$

$$\langle true : \sigma_i \rangle \;\Box B_{i+1}, \ldots, B_k$$

to the induction hypothesis. $\Box$

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

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

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

$$ans_I^{\mathcal{T}}(G) \;=$$

$$\left\{ \sigma \;\middle|\; \begin{array}{l} for\ i = 1..n \\ \quad p_i(\bar{s}_i) \in I[\mathcal{T}](p_{i+1}(\bar{t}_{i+1}), ..., p_n(\bar{t}_n)), \\ p_1(\bar{s}_1), \ldots, p_n(\bar{s}_n), G \ are\ renamed\ apart, \\ \sigma = \bigcup_{i=1}^{n} \{\bar{s}_i = \bar{t}_i\} \cup tell,\ \mathcal{H} \models_{\emptyset} \sigma \end{array} \right\}$$

Theorem 3.4 (strong equivalence)
Let P be τ program, $\mathcal{T}$ a termination theory and $G = \langle true : tell \rangle \,\Box\, B_1, ..., B_n$ be a goal. Then $\sigma' \in ans_{\mathcal{F}(P)}^{\mathcal{T}}(G)$ iff *$\sigma \in \mathcal{O}_{\mathcal{T}}(P)(G)$ and $G\sigma = G\sigma'$.*

4 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. The basic idea is demonstrated by an example:

Example 4.1
Consider the following Prolog program with the initial goal $G = p(\bar{t}), r(\bar{s})$:

$$p(\bar{t_1}) \leftarrow q(\bar{t_2}).$$

$$p(\bar{t_3}) \leftarrow r(\bar{t_4}).$$

The intended operational semantics can be described by:

1. The first clause is applied to $p(\bar{t})$ if $\bar{t}$ unifies with $\bar{t_1}$. The success of G then depends on the success of the goal $q(\bar{t_2}), r(\bar{s})$, under the unification substitution.

2. The second clause is applied if
   (a) starting the computation of G with the first clause for p, it terminates (e.g. finitely fails or succeeds); and
   (b) $\bar{t}$ unifies with $\bar{t_3}$.

   The success of the goal then depends on the success of $r(\bar{t_4}), r(\bar{s})$ under the unification substitution.

This behaviour is naturally captured by the following τ program:

$$p(\bar{x}) \leftarrow \langle true : \bar{x} = \bar{t_1} \rangle \;\Box\; q(\bar{t_2}).$$

$$p(\bar{x}) \leftarrow \langle term_p^2(p(\bar{x})) : \bar{x} = \bar{t_3} \rangle \;\Box\; r(\bar{t_4}).$$

The ask $term_p^2(p(\bar{x}))$ is intended to test if a Prolog goal can be resolved with the second clause for p. Namely, that the computation which applies the first clause to that goal terminates. In fact, the operational semantics of τ will test $term_p^2(p(\bar{x}) :: \rho)$ where ρ corresponds to the rest of the Prolog goal. The claim is that given the *intended termination theory* for $term_p^2$ (i.e. the one which corresponds to the Prolog program) the operational semantics of the above τ program captures the intuitive meaning of the original Prolog program. This argument is justified below.

The following definitions specify the transformation from Prolog to τ and what we mean by the *intended termination theory.*

Definition 4.1 (transformation)
Let P be a Prolog program. For each predicate $\{C_1, \ldots, C_m\}$ in P we generate from the clause $C_i \equiv p(\bar{t}) \leftarrow Body$ a corresponding τ clause:

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

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 [19]. An SLD reduction is captured by:

Definition 4.2 (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.

Definition 4.3 (intended termination theory)
Let P be a Prolog program. The intended termination theory for P is defined by:

$\mathcal{T}_{Prolog}(P) =$

$$\left\{ term_p^i(g) \;\middle|\; \begin{array}{l} \{C_1, \ldots, C_n\} \text{ defines p in } P \text{ and} \\ \text{for } 1 \leq i \leq n \\ \quad \text{if } g' \in \bigcup_{j<i} resolve(g, C_j) \text{ then} \\ \quad \text{the SLD tree for } P \text{ and } g' \text{ is finite} \end{array} \right\}.$$

Theorem 4.1 *The intended termination theory for a Prolog program P is a termination theory (i.e. closed under instantiation).*

PROOF. [(comment)]
The proof follows from properties of SLD trees. □

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 4.4 (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 4.5 (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 Prolog program P.

Definition 4.6 (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 4.7 (Prolog success set)
$\mathcal{S}_{prolog} : Prolog \rightarrow 2^{Atom}$

$$\mathcal{S}_{prolog}(P) = \left\{ A \in Atom \;\middle|\; A \xrightarrow{prolog}{}^*_P \; true \right\}$$

Theorem 4.2 *Let P be a Prolog program and P^τ the corresponding τ program. The Prolog success set $\mathcal{S}_{Prolog}(P) = \mathcal{F}(P^\tau)[\mathcal{T}_{Prolog}(P)](\Lambda)$.*

PROOF. [(comment)]
The proof argues the correspondence between the respective operational behaviours of P and of P^τ given the corresponding intended termination theory. The ask conditions for termination in a computation of P^τ correspond precisely to the conditions for a Prolog transition.

□

Note that:

Proposition 4.3 *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 $term_p^i(g)$ for every resolvent g. $\mathcal{F}(P^\tau)[\mathcal{T}_{lp}](\Lambda)$ is the success set of P interpreted as a logic program.*

In the following examples we write *true* instead of the corresponding termination conditions for the first clause. Furthermore we do not specify these atoms in the respective termination theories. This simplifies somewhat the presentation.

Example 4.2 Consider the following two Prolog programs with $\Sigma = \{0, s\}$ and $\Pi_{\mathcal{P}} = \{p\}$:

P_1 $\quad p(x).$
$\quad\quad p(s(y)) \leftarrow p(y).$

P_2 $\quad p(s(y)) \leftarrow p(y).$

$p(x)$.

The transformed programs are respectively

P_1^τ $\quad p(x) \leftarrow \langle true : true \rangle \,\Box.$

$\quad p(x) \leftarrow \langle term_p^2(p(x)) : x = s(y) \rangle \,\Box\, p(y).$

P_2^τ $\quad p(x) \leftarrow \langle true : x = s(y) \rangle \,\Box\, p(y).$

$\quad p(x) \leftarrow \langle term_p^2(p(x)) : true \rangle \,\Box.$

The intended termination theories for P_1 and P_2 consist respectively of the non-ground and ground sets of goals:

$\mathcal{T}_{Prolog}(P_1) = \{term_p^2(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, Var)^*\}$ and

$\mathcal{T}_{Prolog}(P_2) = \{term_p^2(g) \mid g \in \mathcal{A}(\Sigma, \Pi_{\mathcal{P}}, \emptyset)^*\}.$

The open declarative meanings of P_1^τ and P_2^τ correspond to those given in Example 3.5; the success sets correspond to those given in Example 3.6:

$\mathcal{S}_{Prolog}(P_1) =$

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

$\mathcal{S}_{Prolog}(P_2) =$

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

which express the computational behaviour of the original Prolog programs. In particular, $\mathcal{F}(P_2^\tau)[\mathcal{T}_{Prolog}(P_2)](\Lambda)$ does not include $p(x)$ which corresponds to the fact the the Prolog computation does not succeed for that goal (because it diverges).

Note that while the depth-first search strategy is captured by the transformation and the corresponding termination conditions, the left-to-right selection rule is modelled by the sequential composition of atoms in the bodies in τ-clauses.

In the following example we show how the order of the atoms in the body of a Prolog clause leads to different solutions and how the declarative semantics of the transformed program captures this fact.

Example 4.3 The Prolog program Q

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

$p(s(y)) \leftarrow p(y).$

$p(x).$

$r(0).$

is transformed in a τ-program Q^τ:

$q(x) \leftarrow \langle true : true \rangle \,\Box\, p(x), r(x).$

$p(x) \leftarrow \langle true : x = s(y) \rangle \,\Box\, p(y).$

$p(x) \leftarrow \langle term_p^2(p(x)) : true \rangle \,\Box.$

$r(x) \leftarrow \langle true : x = 0 \rangle \,\Box.$

The declarative meaning of Q^τ is

$$\mathcal{F}(Q^\tau) = \left\{ \begin{array}{l} r(x) \leftarrow \langle true : x = 0 \rangle \\ q(x) \leftarrow \langle term_p^2(p(x), r(x)) : true \rangle \\ \qquad \leftarrow \langle true : x = 0 \rangle \end{array} \right\} \bigcup \mathcal{F}(P_2^\tau)$$

where $\mathcal{F}(P_2^\tau)$ is defined in Example 3.5. Note that the solutions $x = s^n(0)$ are inconsistent with the only solution for r (i.e. $x = 0$).

The intended termination theory $\mathcal{T}_{Prolog}(Q)$ contains $\{term_p^2(p(t), r(t')) \mid t \in T(\Sigma), t' \in T(\Sigma, Var)\}$. By considering the intended termination theory, the goal $q(x)$ does not succeed because $\mathcal{T}_{Prolog}(Q) \not\models_\emptyset^\Lambda term_p^2(p(x), r(x))$, while $q(0)$ succeeds. Changing the order of the atoms in the body of the first Prolog clause (and in the corresponding τ one) leads to the model

$$\mathcal{F}(Q^\tau) = \left\{ \begin{array}{l} r(x) \leftarrow \langle true : x = 0 \rangle \\ q(x) \leftarrow \langle true(p(x)) : x = 0 \rangle \\ \qquad \leftarrow \langle term_p^2(p(x)) : true \rangle \end{array} \right\} \bigcup \mathcal{F}(P_2^\tau).$$

In this case $q(x)$ succeeds with answer $x = 0$.

5 Modelling Termination

In the previous section, we have shown how to capture declaratively the success set of a Prolog program P by composing an interpretation for a corresponding τ program P^τ with the intended termination theory $\mathcal{T}_{Prolog}(P)$. Of course $\mathcal{T}_{Prolog}(P)$ cannot be effectively computed. However, the goal of this paper is to provide a semantic basis for the analysis of Prolog programs and for this purpose approximations of $\mathcal{T}_{Prolog}(P)$ both from above and from below can be useful:

<u>from above:</u> $\mathcal{T}_a \supseteq \mathcal{T}_{Prolog}(P) \Rightarrow \mathcal{F}(P^\tau)[\mathcal{T}_a](\Lambda) \supseteq \mathcal{F}(P^\tau)[\mathcal{T}_{Prolog}(P)](\Lambda)$ hence providing an approximation from above of the (Prolog) success set for P. This may provide better approximations than can be achieved when abstracting the meaning of P as a logic program. In particular because by Proposition 4.3 the success set of the logic program P is $\mathcal{F}(P^\tau)[\mathcal{T}_{lp}](\Lambda)$ and taking $\mathcal{T}_a \subset \mathcal{T}_{lp}$, gives $\mathcal{F}(P^\tau)[\mathcal{T}_a](\Lambda) \subset \mathcal{F}(P^\tau)[\mathcal{T}_{lp}](\Lambda)$.

<u>from below:</u> $\mathcal{T}_b \subseteq \mathcal{T}_{Prolog}(P) \Rightarrow \mathcal{F}(P^\tau)[\mathcal{T}_b](\Lambda) \subseteq \mathcal{F}(P^\tau)[\mathcal{T}_{Prolog}(P)](\Lambda)$ hence providing an approximation from below of the (Prolog) success set for P which can be useful for example in the context of the complexity analysis described in [14].

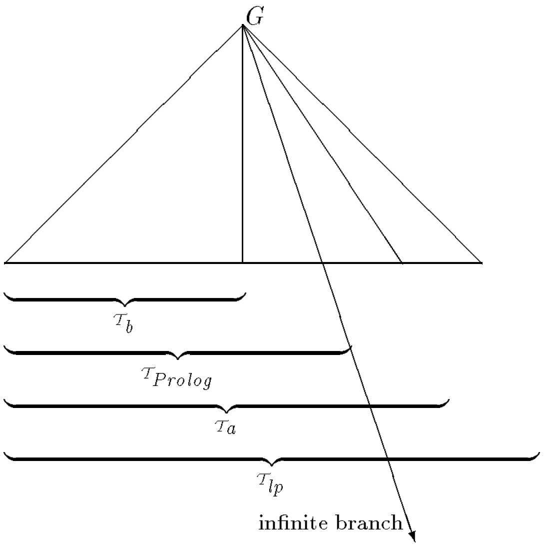

Figure 5 SLD-tree for G with P

Figure 5 illustrates the computation tree for a logic program P and a goal G for a left-to-right selection rule in which the left most infinite branch is indicated. Let P^τ denote the corresponding τ program. The solutions for G correspond to the success branches of the entire tree. These can be derived by composing $\mathcal{F}(P^\tau)$ with $\mathcal{T}_{lp}$. The solutions for G assuming Prolog's depth-first strategy are those which are to the left of the infinite branch. These can be derived by composing $\mathcal{F}(P^\tau)$ with the intended termination theory $\mathcal{T}_{Prolog}(P)$. Approximations of the Prolog solutions for G from above and below can be derived by composing $\mathcal{F}(P^\tau)$ respectively with $\mathcal{T}_a$ and $\mathcal{T}_b$ which approximate $\mathcal{T}$ from above and from below.

We would like to capture the intended termination theory for a Prolog program P as the declarative meaning of some corresponding τ program. This would provide a precise declarative specification of $\mathcal{T}_{Prolog}(P)$ by applying the τ semantics. Furthermore it would enable approximations to be derived by abstract interpretation. Consider the following example:

Example 5.1
Let $\Pi_{\mathcal{T}_{\neq}} = \{\neq\}$ (i.e. *"not unifiable"*) and consider the following τ program R:

$$t(x) \leftarrow \langle x \neq s(\_) : true \rangle \ \Box.$$
$$t(x) \leftarrow \langle true : x = s(x') \rangle \ \Box \ t(x').$$

The declarative meaning of R is

$$\mathcal{F}(R) = \left\{ \begin{array}{l} t(x) \leftarrow \langle x \neq s(\_) : true \rangle, \\ t(x) \leftarrow \langle true : x = s(x') \rangle \leftarrow \langle x' \neq s(\_) \rangle, \\ \ldots \end{array} \right\}$$

If $\mathcal{T}_{\neq}$ is the (downward closed) theory corresponding to "not-unifiable" and $\Sigma = \{0, s\}$ then $\mathcal{F}(R)[\mathcal{T}_{\neq}](\Lambda)$ gives the set $\{t(0), t(s(0)), t(ss(0)), \ldots\}$ which corresponds to the intended termination theory for the Prolog program P_2 in Example 3.5.

However, in general to model intended termination theories as τ programs we need to introduce a notion of user defined predicates in the ask parts of clauses and to interpret them as local tests which do not affect the global bindings of variables.

6 Conclusions

We have presented a simple constraint logic language τ together with its operational and declarative semantics. The operational semantics is used to justify a transformation which captures the control of a Prolog program given a suitable termination theory. The declarative semantics is parametric with respect to this theory enabling the specification of Prolog meaning by composition of a "logic part" with a "control part". Because the declarative semantics of τ is both simple and in the usual T_P style it provides a suitable basis for abstract interpretation of Prolog via the defined transformation.

The required termination theory can be approximated either by integrating results on Prolog termination (e.g. [2, 7, 21, 23]) or by extending the τ language so as to model termination from within and applying abstract interpretation.

The basic idea of defining semantics for Prolog without considering notions such as depth-first search and backtracking, which are more (sequential) implementation oriented is an important contribution of this paper. In particular, this idea can be applied to define a simple and concise operational semantics. We are currently in the process of applying this idea to other semantic definitions and extending our results to model full Prolog.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] K. R. Apt and D. Pedreschi. Proving Termination of General Prolog Programs. In *Proc. of Int. Conf. on Theoretical Aspects of Computer Software*, Sendai, Japan, 1991.

[3] B. Arbab and D.M. Berry. Operational and Denotational Semantics of Prolog. *Journal of Logic Programming*, 4:309–330, 1987.

[4] F. Bancilhon, D. Maier, Y. Sagiv, J. D, Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. *Proccedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, 1986.

[5] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. Technical Report TR 12/91, Dipartimento di Informatica, Università di Pisa, 1991. Submitted for publication.

[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, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky T.S.E. Maibaum, editor, *Proc. TAPSOFT'91*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, Berlin, 1991.

[8] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. Technical report, Dipartimento di Informatica, Università di Pisa, 1991.

[9] M. Bruynooghe, G. Janssens, B. Demoen, and A. Callebaut. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc Fourth IEEE Symp. Logic Programming*, pages 192–204. IEEE Comp. Soc. Press, 1987.

[10] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. Technical report, Dept. of Computer Science, The Weizmann Institute, Rehovot, 1990. Submitted for publication.

[11] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[12] 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.

[13] 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.

[14] S.K. Debray and N.W. Lin. Automatic Complexity Analysis of Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 599–613. The MIT Press, Cambridge, Mass., 1991.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[20] K. Marriott and H. Søndergaard. Bottom-up Abstract Interpretation of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 733–748. The MIT Press, Cambridge, Mass., 1988.

[21] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence - subseries of LNCS*. Springer-Verlag, Berlin, 1990. Ph.D. Thesis.

[22] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages*. ACM, 1990.

[23] 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.