A Semantic Basis for the Termination Analysis of Logic Programs *

Michael Codish Cohavit Taboch

Department of Mathematics and Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
{mcodish,taboch}@cs.bgu.ac.il

Abstract

This paper presents a formal semantic basis for the termination analysis of logic programs. The semantics exhibits the termination properties of a logic program through its *binary unfoldings* — a possibly infinite set of binary clauses. Termination of a program P and goal G is determined by the absence of an infinite chain in the binary unfoldings of P starting with G. The result is of practical use as basing termination analysis on a formal semantics facilitates both the design and implementation of analyzers. A simple Prolog interpreter for binary unfoldings coupled with an abstract domain based on symbolic norm constraints is proposed as an implementation vehicle. We illustrate its application using two recently proposed abstract domains. Both techniques are implemented using a standard CLP(R) library. The combination of an interpreter for binary unfoldings and a constraint solver simplifies the design of the analyzer and improves its efficiency significantly.

1 Introduction

This paper provides a declarative (fixed-point) semantics which captures termination properties of logic programs. Several semantic definitions for logic programs have been proposed to capture various notions of observables: the standard minimal model semantics models logical consequences, the c-semantics and the s-semantics [20] model correct and computed answers respectively, the semantic definition of [22] models the notion of call patterns, etc. In this paper we show that the definition of [22] is suitable to model also the termination properties of programs. This provides a formal semantic basis for the analysis of termination of logic programs based on abstract interpretation [12].

We study *left termination*, i.e. the universal termination of logic programs executed by means of *LD-resolution*, which consists of SLD-resolution combined

*This work was supported by a grant from the Israel Science Foundation

with Prolog's leftmost selection rule. By universal termination we refer to the termination of all computations of a given atomic initial goal. This corresponds to the finiteness of the corresponding SLD tree. The results can easily be generalized to consider any local selection rule and non-atomic initial goals.

In general, to prove that a logic program terminates for a given goal it is sufficient to identify a strict decrease in some measure over a well founded domain on the consecutive calls in its computations. The majority of termination analyzes for logic programs apply this common approach but focus on different aspects of proving termination of programs. Several papers ([19, 6, 37]) handle the problem of inferring norms and well founded orders. Others ([16, 37, 7, 25, 4]) present techniques for computing inter—argument relations, an essential component in any termination analysis. The analysis presented in [35] shows how to infer classes of terminating queries using approximations in Natural and Boolean constraint domains. The work of [18] describes an approach which integrates all the components of the termination analysis and produces termination proofs by solving linear constraints. Several implemented systems are described in [38, 31]. An extensive survey of the most common techniques is given in [15].

This paper focuses on the semantics basis for termination analysis. Our semantics, similar to the definition of [22] is based on the notion of binary unfoldings. The idea to use (binary) clauses as semantic objects which capture call patterns first appeared in [24, 23]. This semantics is shown to correspond to the transitive closure of a binary relation which relates consecutive calls selected in a computation. Non-termination for a specific goal implies the existence of a corresponding infinite chain in this relation. Consequently, the semantics of binary unfoldings provides a basis for the analysis of termination using the techniques of abstract interpretation: it captures the termination properties while abstracting away from other details present in an operational semantics such as those based on SLD trees. This is the main contribution of the paper.

Our approach facilitates also the implementation of termination analyzes which are obtained directly by applying abstract interpretation and widening [13] techniques to the binary unfolding semantics. In contrast, previous works apply abstract interpretation to standard semantic definitions to derive properties of a program which are then used in a second phase to reason about its termination behavior. As an implementation vehicle, we introduce a simple Prolog interpreter which computes the binary unfoldings of a given program. Although, in general impractical, as such a computation is non-terminating, when coupled with a suitable technique of abstraction, a termination analyzer is obtained.

For abstraction, we adopt the approach described in [40, 16, 31] where termination analysis is obtained by first abstracting a given program by replacing each term in the program by its size as determined by a suitable norm function. The resulting abstract program expresses relations on the sizes of the program's argument positions. In [40, 31, 16, 37], the authors present techniques to compute finite approximations of these relations and illustrate their use in the analysis of

termination properties. In our approach, the abstract binary unfoldings relate the sizes of arguments in subsequent calls of a computation. Termination is guaranteed by checking that the size of some argument positions decreases in subsequent calls to the same predicate. We describe two different techniques to provide finite approximations of the binary unfoldings of the program, and illustrate their use in proving termination. One is based on linear size relations [25, 4, 14], and the other on monotonicity and equality constraints [7, 31]. Both techniques are implemented using readily available constraint logic programming techniques. A working implementation is accessible for experimentation at `http://www.cs.bgu.ac.il/~taboch/TerminWeb`. A preliminary version of this paper appeared as [10].

2 Preliminaries

In the following we assume a familiarity with the standard definitions and notation for logic programs [33] and abstract interpretation [12]. We assume a first order language with a fixed vocabulary of predicate symbols, function symbols and variables denoted Π, Σ and $\mathcal{V}$. We let $T(\Sigma, \mathcal{V})$ denote the set of terms constructed using symbols from Σ and variables from $\mathcal{V}$; $Atom$ denotes the set of atoms constructed using predicate symbols from Π and terms from $T(\Sigma, \mathcal{V})$. A goal is a finite sequence of atoms. The set of goals is denoted by $Atom^*$. The empty goal is denoted by $true$. Substitutions and their operations are defined as usual. The most general unifier of syntactic objects s_1 and s_2 is denoted $mgu(s_1, s_2)$. A syntactic object s_1 is more general than a syntactic object s_2 denoted $s_2 \leq s_1$ if there exists a substitution θ such that $s_2 = s_1\theta$.

A clause is an object of the form $head \leftarrow body$ where $head$ is an atom and $body$ is a goal. If $body$ consists of at most one atom then the clause is said to be binary. The set of binary clauses is denoted $\Im$. A binary clause can be viewed as a relation "$\leftarrow$" on $Atom \times (Atom \cup \{true\})$. An *identity clause* is a binary clause of the form $p(\bar{x}) \leftarrow p(\bar{x})$ where $\bar{x}$ denotes a tuple of distinct variables. Identity clauses play a technical role in the definition of the fixed point semantics presented below. These clauses are identities with respect to the operation of unfolding. Namely, unfolding a clause C with an identity clause gives back the clause C. We denote by id the set of identity clauses over the given alphabet.

A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. Two syntactic objects t_1 and t_2 are *equivalent up to renaming* if $t_1\rho = t_2$ for some variable renaming ρ. Given an equivalence class X of syntactic objects and a finite set of variables V, it is always possible to find a representative x of X that contains no variables from V. For a syntactic object s and a set of equivalence classes of objects I, we denote by $\langle c_1, \ldots, c_n \rangle \ll_s I$ that $c_1, \ldots, c_n$ are representatives of elements of I renamed apart from s and from each other. Note that for $n = 0$ (the empty tuple) $\langle\ \rangle \ll_s I$ holds vacuously for any I and s (in particular if $I = \emptyset$). In the discussion that follows, we will be concerned with sets of binary

clauses modulo renaming. For simplicity of exposition, we will abuse notation and assume that a (binary) clause represents its equivalence class. The power set of $\Im$ is denoted $\wp(\Im)$.

3 Operational semantics

The operational semantics for logic programs is formalized as usual 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.

Definition 3.1 (LD-resolution)
Let P be a logic program. An LD-resolution step for P is the smallest relation $\leadsto_P \subseteq Atom^* \times Atom^*$ such that $G \leadsto_P G'$ if and only if

1. $G = \langle a_1, \ldots, a_k \rangle$,
2. $h \leftarrow b_1, \ldots, b_n \ll_G P$,
3. $\vartheta = mgu(a_1, h)$, *and*
4. $G' = \langle b_1, \ldots, b_n, a_2, \ldots, a_k \rangle \vartheta$.

We sometimes write $G \overset{\vartheta}{\leadsto}_P G'$ to indicate explicitly the substitution ϑ associated with the resolution step.

Definition 3.2 (LD-derivation)
Let P be a logic program and G_0 an initial goal. An LD-derivation of P and G_0 is a (finite or infinite) sequence of goals consecutively related by LD-resolution steps, such that the renamed clause used in each derivation step is variable disjoint from the initial query, the substitutions and the renamed clauses used at earlier steps. If $G_0 \overset{\vartheta_1}{\leadsto}_P G_1 \ldots \overset{\vartheta_n}{\leadsto}_P G_n$ is a derivation and $\vartheta = \vartheta_1 \circ \cdots \circ \vartheta_n$ then we write $G_0 \overset{\vartheta}{\leadsto}{}^*_P G_n$. If there is an infinite derivation of the form $G_0 \overset{\vartheta_1}{\leadsto}_P G_1 \ldots \overset{\vartheta_n}{\leadsto}_P G_n \ldots$ then we say that G_0 is non-terminating with P.

The following is an operational definition for the notions of calls and answers.

Definition 3.3 (calls and answers)
Let P be a program and G_0 be a goal. We say that A is a call in a derivation of G_0 with P if and only if $G_0 \leadsto^*_P \langle A, \ldots \rangle$. We denote by $calls_P(G_0)$ the set of calls in the computations of G_0 with P. We say that $G_0\theta$ is an answer for G_0 with P if $G_0 \overset{\theta}{\leadsto}{}^*_P true$. The set of answers for G_0 with P are denoted $ans_P(G_0)$.

The calls-to relation specifies the dependencies between calls in a computation and serves as a convenient link between the operational and denotational semantics with regards to observing termination.

Definition 3.4 (calls-to relation $\hookrightarrow$)
We say that there is a call from a to b in a computation of the goal G_0 with the program P, denoted $a \overset{P,G_0}{\hookrightarrow} b$, if $a \in calls_P(G_0)$ and $b \in calls_P(a)$. When

clear from the context we write $a \hookrightarrow b$ *or* $a \overset{\vartheta}{\hookrightarrow} b$ *to emphasize that* ϑ *is the substitution associated with a corresponding derivation from* $\langle a \rangle$ *to* $\langle b, \ldots \rangle$.

The following lemma provides the connection between termination and the calls-to relation.

Lemma 3.5 (observing termination in the calls-to relation)
Let P *be a program and* G_0 *be a goal. Then, there is an infinite derivation for* G_0 *with* P *if and only if there is an infinite chain in the calls-to relation.*

Proof.

($\Leftarrow$) Immediate by the definition of the calls-to relation. Since two related calls are in different derivation steps, then an infinite chain of calls implies an infinite derivation.

($\Rightarrow$) We show that for each k, there is a chain $a_0 \hookrightarrow \cdots \hookrightarrow a_k$ such that the goal $\langle a_k \rangle$ has an infinite derivation with P.

base: Let $G_0 = c_1, \ldots, c_n$. We pick a_0 to be $c_i\vartheta_0$ such that $G_0 \overset{\vartheta_0}{\leadsto}{}^* \langle (c_i, \ldots, c_n)\vartheta_0 \rangle$ and such that $\langle c_i\vartheta_0 \rangle$ has an infinite derivation with P. Such an atom must exist since G_0 has an infinite derivation.

step: Assume the existence of a chain $a_0 \hookrightarrow \cdots \hookrightarrow a_k$ such that a_k has an infinite derivation $\delta = g_0, g_1, g_2, \ldots$ with P where $g_0 = \langle a_k \rangle$. We show that there exists an atom a_{k+1} such that $a_k \hookrightarrow a_{k+1}$ which has an infinite derivation with P. Let $g_1 = \langle b_1, \ldots, b_m \rangle$. We pick a_{k+1} to be $b_j\vartheta_k$ such that $\langle a_k \rangle \overset{\vartheta_k}{\leadsto}{}^* \langle (b_j, \ldots, b_m)\vartheta_k \rangle$, and $\langle b_j\vartheta_k \rangle$ has an infinite derivation. Such consecutive state must exist since $\langle a_k \rangle$ has an infinite derivation.

□

4 Denotational Semantics

As a basis for termination analysis, we adopt a simplification of the goal independent semantics for call patterns defined in [22]. The definition is given as the fixed point of an operator T_P^β over the domain of *binary clauses*. Intuitively, a binary clause $a \leftarrow b$ specifies that a call to a in a computation implies a subsequent call to b. A clause of the form $a \leftarrow \mathit{true}$ is a fact, and indicates a success pattern. We refer to this semantics as defining the set of *binary unfoldings* of a program. Given any set I of binary clauses, $T_P^\beta(I)$ is constructed by unfolding prefixes of clause bodies to obtain new binary clauses. Let $h \leftarrow b_1, \ldots, b_m$ be a clause in P:

(1) for each $1 \leq i \leq m$, we unfold $b_1, \ldots, b_{i-1}$ with facts $h_1, \ldots, h_{i-1}$ from I to obtain a corresponding instance of $h \leftarrow b_i$.

(2) for each $1 \leq i \leq m$ we unfold $b_1, \ldots, b_{i-1}$ with facts from I and we also unfold b_i with a binary clause $h_i \leftarrow b$ from I, which is not a fact ($b \neq true$), to obtain a corresponding instance of $h \leftarrow b$.

(3) we unfold $b_1, \ldots, b_m$ with facts from I to obtain a corresponding instance of h.

This is expressed concisely in the following definition. Note the use of the identity clause in the third line of the definition so that cases (1) and (2) coincide, and that (3) computes the standard s-semantics.

Definition 4.1 (binary unfoldings semantics)

$$T_P^\beta : \wp(\Im) \rightarrow \wp(\Im)$$

$$T_P^\beta(I) = \left\{ (h \leftarrow b)\vartheta \;\middle|\; \begin{array}{l} C = h \leftarrow b_1, \ldots, b_m \in P,\ 1 \leq i \leq m, \\ \langle h_j \leftarrow true \rangle_{j=1}^{i-1} \ll_C I, \\ h_i \leftarrow b \ll_C I \cup id,\ i < m \ \Rightarrow\ b \neq true \\ \vartheta = mgu(\langle b_1, \ldots, b_i \rangle, \langle h_1, \ldots, h_i \rangle) \end{array} \right\}$$

$$bin\_unf(P) = lfp(T_P^\beta)$$

It is not difficult to show that $bin\_unf(P)$ is closed under unfolding. Namely, if $a \leftarrow b$ and $c \leftarrow d$ are renamed apart elements of $bin\_unf(P)$ such that $(d \neq true)$ and $mgu(b, c) = \theta$ then $(a \leftarrow d)\theta$ is also in $bin\_unf(P)$. To see why it is important that $d \neq true$, consider the program

$$P = \left\{ \begin{array}{l} a \leftarrow b, c. \\ b \leftarrow true. \end{array} \right\}$$

The binary unfoldings of P are $\left\{ \begin{array}{l} a \leftarrow b,\ b \leftarrow true,\ a \leftarrow c \end{array} \right\}$ indicating a call from a to b, a call from a to c and a success for b. But there is no binary clause $a \leftarrow true$ and indeed there is no success for a with P.

Example 1 *Consider the following logic program P:*

$p(X, Y) \leftarrow q(X), r(Y), p(X, Y).$
$p(X, Y) \leftarrow r(X), r(Y).$
$p(a, b).$
$q(a). \quad r(b).$

The binary unfoldings of P are evaluated as follows:

1. $(T_P^\beta)^1(\emptyset) = \left\{ \begin{array}{l} p(A, \_) \leftarrow q(A), p(A, \_) \leftarrow r(A), \\ p(a, b) \leftarrow true, q(a) \leftarrow true, r(b) \leftarrow true \end{array} \right\}$

2. $(T_P^\beta)^2(\emptyset) = \left\{ \begin{array}{l} p(a,A) \leftarrow r(A), p(a,b) \leftarrow p(a,b), \\ p(a,b) \leftarrow q(a), p(a,b) \leftarrow r(a), \\ p(b,A) \leftarrow r(A), p(b,b) \leftarrow true \end{array} \right\} \cup (T_P^\beta)^1(\emptyset)$

3. $(T_P^\beta)^3(\emptyset) = \{ \; p(a,b) \leftarrow r(b) \; \} \cup (T_P^\beta)^2(\emptyset)$

4. $(T_P^\beta)^4(\emptyset) = (T_P^\beta)^3(\emptyset)$ *(fixed point).*

In [22] and similarly in [9] the authors show that the binary unfoldings of a program provide a goal independent representation of its success and call patterns.

Proposition 4.2 (observing calls and answers)
Let P be a program and G an atomic goal. Then, the computed answers for G with P and the calls that arise in the computations of G with P are characterized respectively by :

1. $$ans_P(G) = \left\{ \; G\vartheta \;\middle|\; \begin{array}{l} h \leftarrow true \in bin\_unf(P), \\ \vartheta = mgu(G,h) \end{array} \right\}$$

2. $$calls_P(G) = \left\{ \; b\vartheta \;\middle|\; \begin{array}{l} h \leftarrow b \in bin\_unf(P), \\ \vartheta = mgu(G,h) \end{array} \right\}$$

Example 2 *Consider again the program P from Example 1 and the initial goal $p(a,X)$. Observe that:*

$$calls_P(p(a,X)) = \{ \; q(a), r(a), r(X), p(a,b), r(b) \; \}$$

$$ans_P(p(a,X)) = \{ \; p(a,b) \; \}.$$

This paper illustrates that binary unfoldings exhibit not only the calls and answers of a program but also its termination properties. This motivates the use of binary unfoldings as a semantic basis for termination analysis.

Theorem 4.3 (observing termination)
Let P be a program and G_0 be a goal. Then G_0 is non-terminating for P if and only if G_0 is non-terminating for $bin\_unf(P)$.

Proof. By Lemma 3.5 it is sufficient to show that there is an infinite chain in the calls-to relation for G_0 with P if and only if there is an infinite chain in the calls-to relation for G_0 with $bin\_unf(P)$. We show that in fact the calls-to relations for G_0 with P and with $bin\_unf(P)$ are identical. By Proposition 4.2 (2) it follows that for any goal G, $calls_P(G) = calls_{bin\_unf(P)}(G)$. This implies by Definition 3.4 that

$$a \overset{P,G_0}{\hookrightarrow} b \iff a \overset{bin\_unf(P),G_0}{\hookrightarrow} b.$$

□

```
iterate ← tp_beta, fail.
iterate ← retract(flag),
      iterate.
iterate.

cond_assert(F) ←
      in_database(F), !.
cond_assert(F) ←
      assert(F),
      cond_assert(flag).

in_database(G) ←
      functor(G, N, A),
      functor(B, N, A), call(B),
      variant(B, G), !.

tp_beta ←
      user_clause(Head, Body),
      solve(Head, Body).

solve(Head, [ ]) ←
      cond_assert(fact(Head)).
solve(Head, [B|Bs]) ←
      fact(B),
      solve(Head, Bs).
solve(Head, [B|Bs]) ←
      cond_assert(bin(Head, B)).
solve(Head, [B|_]) ←
      bin(B, C),
      cond_assert(bin(Head, C)).
```

Figure 1: Prolog interpreter for binary unfoldings

Example 3 *Consider the program P from Example 1 which is non-terminating for the initial query $p(a, X)$. The calls-to relation contains the infinite chain $p(a, X) \hookrightarrow p(a, b) \hookrightarrow p(a, b) \hookrightarrow \ldots$ which can (in this simple case) be observed in the binary unfoldings through the clause $p(a, b) \leftarrow p(a, b)$.*

We conclude this section with a simple Prolog interpreter illustrated in Figure 1 which computes the binary unfoldings of a program P, if there are finitely many of them. This interpreter provides the basis for the bottom-up evaluation of the abstract semantics for termination analysis defined in the next sections.

The interpreter assumes that each clause $h \leftarrow b_1, \ldots, b_n$ in P is represented as a fact of the form $user\_clause(h, [b_1, \ldots, b_n])$. The interpreter can be divided conceptually into two components. On the right, the predicate `tp_beta/0` provides the "logic" and the inner loop of the algorithm which for each $user\_clause(Head, Body)$ in P uses the binary unfoldings derived so far to derive new ones. Each time a new fact F is derived it is asserted to the Prolog database as an atom of the form $fact(F)$. Binary clauses are asserted as atoms of the form $bin(H, B)$. The predicate `cond_assert/1` asserts a derived object if it is new — namely not equivalent to any of those derived so far. The first clause in `solve/2` computes new facts (as in the standard s-semantics). The second clause is used to solve prefixes of the clause body with facts from the database. The last two clauses compute new binary clauses. The control component, on the left, invokes iterations of `tp_beta` until no new unfoldings are derived. When a new binary clause is asserted to the Prolog database, a *flag* is raised (unless the flag has already been raised). Iteration terminates when $retract(flag)$ fails in the second clause indicating that nothing new was asserted in the previous

iteration. Bottom-up evaluation is initiated by the query `?- iterate.` which leaves the result of the evaluation in the Prolog database.

Of course, in the general case, the binary unfoldings of a given program are not finitely computable. For termination analysis the Prolog interpreter depicted in Figure 1 is coupled with a suitable notion of abstraction and termination analyzes are based on approximations of the binary unfoldings.

5 Abstracting the Binary Unfoldings

Theorem 4.3 states that the termination behavior of a program P is equivalent to that of its binary unfoldings and justifies the use of this semantics as a basis for termination analysis. Termination analyzes are based on (finite) descriptions of the binary unfolding semantics. To determine that an atomic goal G does not have an infinite computation with $bin\_unf(P)$, the analysis derives a (finite) set of *abstract binary unfoldings* which approximates the (possibly infinite) set of binary unfoldings of P. A sufficient condition for termination determines that none of the elements in this set can represent an infinite subcomputation using corresponding concrete binary clauses. To this end, we adopt the approach described in [40, 16, 31] where termination analyzes are obtained by first abstracting a given program by replacing each term with its size as determined by a suitable norm function. The resulting abstract program expresses relations on the sizes of the programs argument positions. The binary clauses of the abstract program express relations on the sizes of argument positions in subsequent calls.

Syntactically, abstract programs are defined over a first order constraint logic language denoted CLP($\mathcal{N}$) with predicate symbols $\Pi' = \Pi$, function symbols $\Sigma' = N \cup \{+/2\}$ and variables from $\mathcal{V}$. The set of terms that can be constructed from Σ' and $\mathcal{V}$, also called *size expressions*, is denoted $T(\Sigma', \mathcal{V})$. Constraints in CLP($\mathcal{N}$) are conjunctions of $\{=, \leq, \geq, <, >\}$ relations on $T(\Sigma', \mathcal{V})$, sometime denoted as sets.

A solution for a (conjunctive) constraint μ is an assignment to the variables of μ which satisfies μ. The set of solutions which satisfy a constraint μ is denoted $[\mu]_c$. The underlying constraint structure is associated with the standard notions of constraint satisfaction and entailment. A constraint μ is satisfied if $[\mu]_c \neq \emptyset$. A constraint μ_1 entails μ_2 if $[\mu_1]_c \subseteq [\mu_2]_c$. The partial order on constraints $\preceq_c$ is the following :

$$c_1 \preceq_c c_2 \iff [c_1]_c \subseteq [c_2]_c$$

Clauses in CLP($\mathcal{N}$) take the form $h \leftarrow \mu, b_1, \ldots, b_n$ where μ is a constraint, and $h, b_1, \ldots, b_n$ are atoms constructed with predicate symbols from Π and terms from $T(\Sigma', \mathcal{V})$. If $n = 0$ the object is called a *constrained atom*. If $n \leq 1$ the object is a *constrained binary clause*.

In some of the following definitions it is more convenient to use a normalized representation of abstract objects. For example if $p(\bar{t}) \leftarrow \mu$ is a constrained

atom then an alternative equivalent representation is $p(\bar{X}) \leftarrow \mu \wedge \langle \bar{X} = \bar{t} \rangle$ where $\bar{X}$ are distinct variables and $\langle \bar{X} = \bar{t} \rangle$ is a shorthand for $\{X_1 = t_1, \ldots, X_n = t_n\}$. Other objects can be represented in this normal form in a similar way.

A ground instance of a (normalized) CLP($\mathcal{N}$) clause $C = h \leftarrow \mu, \bar{b}$ is an instance of $h \leftarrow \bar{b}$ such that the assignment of natural numbers to the variables of h and $\bar{b}$ satisfies μ. The set of all ground instances of C is denoted $[C]$.

The partial order on CLP($\mathcal{N}$) clauses is the following. Let C_1 and C_2 be CLP($\mathcal{N}$) clauses. We say that C_1 subsumes C_2, denoted $C_2 \preceq C_1$, if $[C_2] \subseteq [C_1]$. The induced equivalence relation is denoted $\approx$. Our abstract domain is the power set of binary CLP($\mathcal{N}$) clauses modulo this notion of equivalence $\wp(\Im_N / \approx)$. For notational simplicity we denote it by $\wp(\Im_N)$, and note that as a power set domain, disjunctive information can be expressed. This has the same effect as in [26] where the authors use interpretations with (possibly infinite) distributive constraints allowed. Therefore the least upper bound is simply set union, the minimal element is the empty set and the maximal element is $\Im_N$. The partial order on this domain is defined as:

$$\mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \iff \bigcup_{x \in \mathcal{I}_1} [x] \subseteq \bigcup_{y \in \mathcal{I}_2} [y]$$

The relation between the concrete and the abstract domains is determined by a norm which maps concrete terms to abstract terms. The norm functions we use are termed *symbolic norms*. Symbolic norms are similar to semi-linear norms as defined in [6], but variables are mapped to variables. Symbolic norms are equivalent to the *norm based abstractions* defined in [40] for semi linear norms.

Definition 5.1 *[31]* **(symbolic norm)**
A symbolic norm is a function $\|\cdot\| : T(\Sigma, \mathcal{V}) \rightarrow T(\Sigma', \mathcal{V})$ *such that*

$$\|t\| = \begin{cases} c + \sum_{i=0}^{n} a_i \|t_i\| & \text{if } t = f(t_1, \ldots, t_n) \\ t & \text{if } t \text{ is a variable} \end{cases}$$

where c *and* $a_1, \ldots, a_n$ *are non negative integer constants depending only on* f/n.

Example 4 *Two frequently used norm mappings are the* term-size *norm which counts the number of edges in the term tree, and the* list-length *norm which counts the number of elements in a list.*

$$\|t\|_{TermSize} = \begin{cases} n + \sum_{i=0}^{n} \|t_i\|_{TermSize} & \text{if } t = f(t_1, \ldots, t_n) \\ t & \text{if } t \text{ is a variable} \end{cases}$$

$$\|t\|_{ListLength} = \begin{cases} 1 + \|Xs\|_{ListLength} & \text{if } t = [X \,|\, Xs] \\ t & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

The following table shows concrete terms and the result of applying the term size and list length symbolic norms.

concrete term t	$\|\|t\|\|_{TermSize}$	$\|\|t\|\|_{ListLength}$
$[a, b, c]$	6	3
$[X, Y, Z]$	$6 + X + Y + Z$	3
$[X, Y \mid Zs]$	$4 + X + Y + Zs$	$2 + Zs$
$f(X, g(Y))$	$3 + X + Y$	0
$f(a, g(b))$	3	0

A symbolic norm is applied to an arbitrary syntactic object s by replacing the terms occurring in s by their sizes. The result of applying the list-length norm on program clauses is demonstrated in Figure 2(**b**) below.

It is important to note that introducing variables into the range of the norm function provides a simple mechanism to express dependencies between the sizes of terms. For example the atom $append(A, B, A+B)$ specifies a relation in which the size of the third argument is equal to the sum of the sizes of the first two arguments.

The relation between the abstract and concrete domains is formalized as a Galois insertion. Concrete clauses are abstracted by replacing terms by the corresponding size expressions. An abstract (constrained) clause describes a concrete clause if the sizes of the arguments in the concrete object satisfy the constraints in the abstract object.

Definition 5.2 (abstraction and concretization)

$\alpha : \wp(\Im) \rightarrow \wp(\Im_N)$ $\qquad$ $\gamma : \wp(\Im_N) \rightarrow \wp(\Im)$

$\alpha(I) = \{ \ \|a \leftarrow b\| \ \big| \ a \leftarrow b \in I \ \}$ $\qquad$ $\gamma(\mathcal{I}) = \{ \ c \ \big| \ \beta \in \mathcal{I}, \|c\| \preceq \beta \ \}$

The following lemma is a straightforward consequence of Definition 5.2.

Lemma 5.3 $(\wp(\Im), \alpha, \wp(\Im_N), \gamma)$ *is a Galois insertion.*

The operational semantics and the binary unfolding semantics for CLP($\mathcal{N}$) programs can both be formalized using standard techniques for constraint logic languages as described for example in [26] or in [28]. The following definition illustrates an operator for the binary unfoldings of a CLP($\mathcal{N}$) program $P_{\|\cdot\|}$. For CLP($\mathcal{N}$) atoms $a = p(t_1, \ldots, t_n)$ and $a' = p(s_1, \ldots, s_n)$ we write $a = a'$ as an abbreviation for the conjunction of equations $\bigwedge_{i=1}^{n} (t_i = s_i)$. For a constraint μ and a set of variables S we denote by $\bar{\exists}(\mu)_S$ the projection of μ onto S.

Definition 5.4 (abstract binary unfoldings semantics)

$$\mathcal{T}^{\alpha}_{P_{\|\cdot\|}} : \wp(\Im_N) \rightarrow \wp(\Im_N)$$

$$\mathcal{T}^{\alpha}_{P_{\|\cdot\|}}(\mathcal{I}) = \left\{ h \leftarrow \mu, b \;\middle|\; \begin{array}{l} C = h \leftarrow \mu_0, b_1, \ldots, b_m \in P_{\|\cdot\|},\ 1 \leq i \leq m, \\ \langle a_j \leftarrow \mu_j \rangle_{j=1}^{i-1} \ll_C \mathcal{I}, \\ a_i \leftarrow \mu_i, b \ll_C \mathcal{I} \cup id,\ i < m \Rightarrow b \neq true \\ \mu' = \mu_0 \bigwedge_{j=1}^{i} (\mu_j \wedge \{b_j = a_j\}) \\ \mu = \bar{\exists}(\mu')_{vars(\{h,b\})} \end{array} \right\}$$

$$bin\_unf^{\alpha}(P_{\|\cdot\|}) = lfp(\mathcal{T}^{\alpha}_{P_{\|\cdot\|}})$$

The correctness of the operator in Definition 5.4, namely that $bin\_unf(P) \subseteq \gamma(bin\_unf^{\alpha}(P_{\|\cdot\|}))$ follows from the framework of generalized semantics and abstract interpretation for constraints logic programs introduced in [26].

It is important to note that the domain of CLP($\mathcal{N}$) binary clauses does not satisfy the ascending chain condition. Most of the abstract interpretation based argument relations analyzes for logic programs described in the literature differ primarily in the way they further approximate this abstract domain to obtain finite analyzes. For example, in [40, 29], a technique is described to derive affine inter-argument relations using linear equations, in [4, 14] the authors propose polyhedral approximations combined with a convex hull operation as a least upper bound and a widening, and in [31, 7] the authors use disjunctions of monotonicity and equality constraints.

Figure 2 illustrates a Prolog program for mergesort together with its list-length abstraction, and approximations of the inter-argument relations for the `split` predicate obtained using the two latter abstract domains.

6 A Sufficient Condition for Termination

The classic approach to termination analysis (see for example [34]) uses the well founded ordering method by Floyd ([21]). Proving termination of a program naturally focuses on the possible sources for non-termination, i.e loops. Proofs involve showing that consecutive iterations of the loop decrease the value of some expression related to the execution (e.g. the size of the program state) which belongs to a well founded set W. Since W does not contain infinite decreasing sequences, termination is guaranteed.

Termination proofs for logic programs are also usually based on well founded orders. The objects measured are the terms in the clause atoms. The notion of a *level mapping* for measuring atoms was introduced by [8] and used in [1, 3, 17, 5] for proving termination of logic programs.

Definition 6.1 (level mapping)
A level mapping for a program P is a function $|\ |: B_P \longrightarrow N$ of ground atoms to natural numbers. For $A \in B_P$, $|A|$ is the level of A.

Definition 6.2 *An atom A is called bounded with respect to a level mapping $|\ |$, if $|\ |$ is bounded on the set of ground instances of A.*

$mergesort([\,],[\,]).$
$mergesort([X],[X]).$
$mergesort([X, Y|X_s], Y_s) \leftarrow$
$\quad split([X, Y|X_s], X_{1s}, X_{2s}),$
$\quad mergesort(X_{1s}, Y_{1s}),$
$\quad mergesort(X_{2s}, Y_{2s}),$
$\quad merge(Y_{1s}, Y_{2s}, Y_s).$

$split([\,],[\,],[\,]).$
$split([X|X_s],[X|Y_s],Z_s) \leftarrow$
$\quad split(X_s, Z_s, Y_s).$

$merge([\,], X_s, X_s).$
$merge(X_s,[\,], X_s).$
$merge([X|X_s],[Y|Y_s],[X|Z_s]) \leftarrow$
$\quad X \leq Y, merge(X_s,[Y|Y_s], Z_s).$
$merge([X|X_s],[Y|Y_s],[Y|Z_s]) \leftarrow$
$\quad X > Y, merge([X|X_s], Y_s, Z_s).$

(a) The mergesort relation

$mergesort(0,0).$
$mergesort(1,1).$
$mergesort(2+X_s, Y_s) \leftarrow$
$\quad split(2+X_s, X_{1s}, X_{2s}),$
$\quad mergesort(X_{1s}, Y_{1s}),$
$\quad mergesort(X_{2s}, Y_{2s}),$
$\quad merge(Y_{1s}, Y_{2s}, Y_s).$

$split(0,0,0).$
$split(1+X_s,1+Y_s,Z_s) \leftarrow$
$\quad split(X_s, Z_s, Y_s).$

$merge(0, X_s, X_s).$
$merge(X_s, 0, X_s).$
$merge(1+X_s,1+Y_s,1+Z_s) \leftarrow$
$\quad X \leq Y, merge(X_s, 1+Y_s, Z_s).$
$merge(1+X_s,1+Y_s,1+Z_s) \leftarrow$
$\quad X > Y, merge(1+X_s, Y_s, Z_s).$

(b) The abstract mergesort relation

$split(A, B, C) \leftarrow \{A = B, A = C\}.$
$split(A, B, C) \leftarrow \{A = B, C < A\}.$
$split(A, B, C) \leftarrow \{B < A, C < A\}.$

$split(A, B, C) \leftarrow \{D < A, F < B, C = E\},$
$\quad split(D, E, F).$
$split(A, B, C) \leftarrow \{D < A, E < B, F < C\},$
$\quad split(D, E, F).$
$split(A, B, C) \leftarrow \{D < A, F < B, E < C\},$
$\quad split(D, E, F).$

(c) Size dependencies for split using monotonicity and equality constraints

$$split(A, B, C) \leftarrow \left\{ \begin{array}{l} A = B + C,\ C \geq 0, \\ B - C \leq 1,\ B - C \geq 0 \end{array} \right\}.$$
$$split(A, B, C) \leftarrow \left\{ \begin{array}{l} B + C \geq E + F + 1, \\ A + E + F = B + C + D, \\ B \geq 1,\ C \geq 0,\ D \geq 0, \\ E \geq 0, F \geq 0 \end{array} \right\},$$
$\quad split(D, E, F).$

(d) Size dependencies for split using polyhedral approximations

Figure 2: The mergesort relation, its list-length abstraction, and inter-argument relations for the split relation

Some of the termination proofs which consider the selection rule and especially left termination proofs use the notion of *acceptability* (defined in [2]) with respect to a level mapping.

Definition 6.3 *Let P be a program, $|\ \ |$ a level mapping for P and I a (not necessarily herbrand) interpretation of P. A clause of P is called acceptable with respect to $|\ \ |$ and I if I is its model and for every ground instance $A \leftarrow B_1, \ldots, B_{i-1}, B_i, \ldots, B_n$ of it such that $I \models B1, \ldots, B_{i-1}$ then $|A| > |B_i|$.*

The information on the program's model is usually is obtained by a separate inter-argument relations analysis. The condition of acceptability was relaxed in [3] to *semi-acceptability* which requires that a strict decrease will be shown only for (direct or indirect) recursive body atoms, and that a weak decrease will be shown for other body atoms to ensure that if an initial goal is bounded then so are all the consecutive calls. For acceptable and semi-acceptable programs termination is proved for any initial goal that is bounded with respect to the given level mapping. In [17] the authors extend this to the notion of acceptability for any set of atoms S, not necessarily ground. The advantage is that boundedness of a call can now be determined using any mode or rigidity analysis. Therefore it is sufficient to show a strict decrease from every instance of the head unified with an atom from S to the corresponding instances of recursive body atoms.

The advantage of our framework is that all the information needed is embedded in the binary clauses. Since we have already shown that the termination behavior of a program P is equivalent to that of $bin\_unf(P)$, the termination proof in our case is performed directly on approximations of $bin\_unf(P)$. Using approximations of binary unfoldings simplifies the proof, since the argument relations which are computed using the abstract size relations domain are applied to the binary unfoldings. Therefore the termination condition is observed directly from the abstract binary unfoldings.

Our termination condition is similar to the one used in most termination proofs. However, the following relaxations are possible. First there is no need to prove acceptability, since the clauses are binary. So it is sufficient to show a strict decrease from the head to the (single) body atom. Moreover, it is sufficient to consider only the "direct recursive" binary clauses, i.e the clauses where the head and the body atoms have the same predicate symbol. This is due to the fact that binary clauses are closed under unfolding. An additional relaxation observed in [31] is the following: it is sufficient to test only the (abstract) calls and binary clauses which unifying the call and the binary clause head results in a new call pattern in the clause body which is equivalent to the current call pattern. All of the above conditions are explained in proposition 6.5.

Boundedness in our context means that the result of applying the symbolic norm on the relevant term will be ground. We recall the notion of *instantiated enough* with respect to a symbolic norm which is closely related to that of rigidity (see for example [6]).

Definition 6.4 *[31]* **(instantiated enough)**
A term t is instantiated enough with respect to a symbolic norm $\|\cdot\|$ if $\|t\|$ is an integer (i.e does not contain variables).

Example 5 *The list of variables $[X_1, X_2, X_3]$ is instantiated enough with respect to the list length symbolic norm since $\|[X_1, X_2, X_3]\|_{ListLength} = 3$ however it is not instantiated enough with respect to the term size norm since $\|[X_1, X_2, X_3]\|_{TermSize} = X_1 + X_2 + X_3 + 6$*

In the following we assume a combined abstract domain representing both size relations and instantiation information with respect to a given symbolic norm. Size relations are obtained as described in section 5. Instantiation information can be obtained by performing any standard groundness analysis on the abstract program, such as that based on the *Prop* domain ([11, 9]). The operations on this domain consist of the standard operation on both domains. We denote by mgu^α the abstract most general unifier over this combined domain. The analysis is for any initial atomic goal which is described by an initial goal description G_0^α. In practice, G_0^α will describe only instantiation information to specify the "input" and "output" argument positions of the initial query (leaving the sizes unconstrained). Let us denote by $\mathcal{B}$ the (finite) set of abstract binary clauses which approximates the binary unfoldings of P over the abstract domain, and by $\mathcal{C}$ the (finite) set of abstract atoms describing the calls that arise in the computations of the initial goals described by G_0^α. These are determined based on (the abstract version of) Proposition 4.2.

Proposition 6.5 *Let $\beta = h \leftarrow b \in \mathcal{B}$ and $c \in \mathcal{C}$ such that $\kappa = mgu^\alpha(h, c)$ and that $b\kappa \approx c$. And let $i_1, \ldots, i_k$ be the argument positions which are instantiated enough both in $h\kappa$ and in $b\kappa$. We denote them by $(h\kappa)_{i_1}, \ldots, (h\kappa)_{i_k}$ and $(b\kappa)_{i_1}, \ldots, (b\kappa)_{i_k}$. Then if for each such $\beta \in \mathcal{B}$ and $c \in \mathcal{C}$ there exists a function f such that $f((h\kappa)_{i_1}, \ldots, (h\kappa)_{i_k}) > f((b\kappa)_{i_1}, \ldots, (b\kappa)_{i_k})$ then G_0 terminates with P.*

In practice the function f for measuring the head and the body argument positions is usually the sum or some linear combination of the relevant (abstract) arguments. Also note that if $mgu^\alpha(h, c)$ exists and $b\kappa \approx c$, then the head and the body of the binary clause have the same predicate symbol.

Proof. By contradiction, assume the premise of the proposition and that G_0 is non-terminating with P. Then by Theorem 4.3 G_0 is non-terminating also with $bin\_unf(P)$ and there exists an infinite (concrete) chain of related calls generated by G_0 and $bin\_unf(P)$. Since the number of abstract binary unfoldings and calls is finite, this chain must have an infinite sub-chain which is generated by a set of binary unfolding B and calls C which are approximated by a single abstract binary clause β and a single abstract call pattern c. Moreover β and c satisfy the assumptions given in the proposition. But since we have

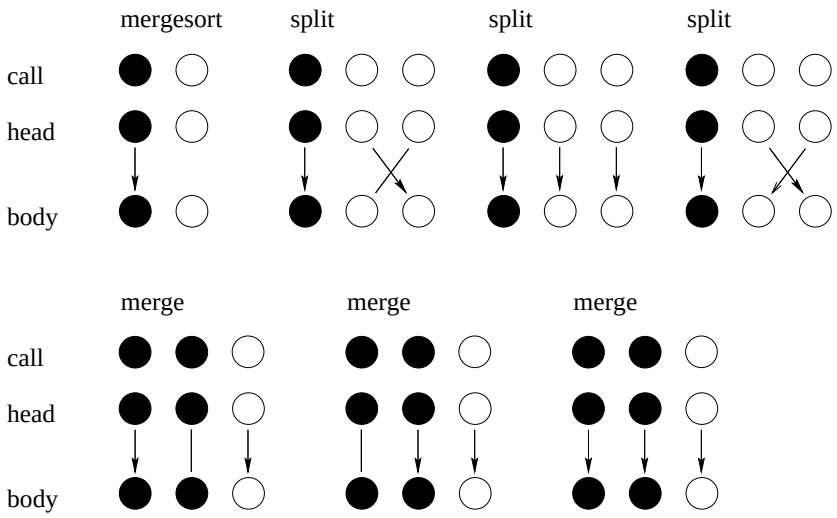

Figure 3: Termination condition for mergesort

shown that for each such binary clause and call a strict decrease from the head to the body arguments which are instantiated enough, then this chain cannot be infinite. □

Figure 3 depicts the pairs of abstract calls and binary clauses for the mergesort program of Example 2. Each pair is indicated as a graph with black and white nodes, similar to the graphs in [31]. Argument positions which are determined as instantiated enough are depicted as black nodes. Arrows between nodes indicate a strict decrease in the size of the corresponding argument positions. An edge indicates that the argument positions are of equal size. Termination for the mergesort program is determined by observing that each pair of the binary clauses in the figure contains a strict decrease in size from a black node in the head to a black node in the body. Notice that only the "recursive" binary clauses with their corresponding call patterns need to be considered.

7 Implementation

We have implemented a termination analyzer based on the meta-interpreter of Figure 1 abstracted to maintain size and instantiation information expressed in CLP($\mathcal{N}$) and Prop [11] respectively. The analyzer and the source code are available for experimentation from `http://www.cs.bgu.ac.il/~taboch/TerminWeb`.

The implementation adopts a semi-naive evaluation strategy which also takes into account the strongly connected components in the program's call graph. The system uses the constraint solver library over the reals CLP(R) [27] provided by SICStus Prolog [39]. To obtain finite analyzes, the system supports two alternative strategies: the polyhedral based approach (with widening) as described in [4]; and the use of disjunctions of equality and monotonicity constraints as described in [31]. Using either method, the analysis produces a finite set of abstract binary unfoldings for a given program. Both of the methods integrate into our semantic basis and are easily implemented using constraint logic programming technology.

The analysis which determines instantiation dependencies is implemented using the approach described in [9] (for logic programs) and using the interpreter of Figure 1 to obtain the corresponding information on the instantiation of terms in the binary unfoldings and calls.

The use of the constraint solver simplifies the operations on both abstract domains. The method adapted from [31] was originally described (and implemented) using weighted graphs. The use of constraints also improves the efficiency of the implementation considerably, since the constraints on pairs of argument positions are obtained using an entailment check on each pair of argument positions. The implementation of the polyhedral approximations follows the approach described in [4]. All of the operations on this domain: projection, convex hull and widening are implemented using the constraints solver. In addition we apply the approach described in [4, 13] where the widening is delayed for several iterations to obtain more precise results.

In the implementation, the function f described in Proposition 6.5 is the sum function. We check that the sum of some argument positions is strictly decreasing from the head to the body. We use the constraint solver to detect this in the following way: Consider a recursive binary clause $p \leftarrow \mu, p'$ and let $h_1, \ldots, h_k$ and $b_1, \ldots, b_k$ be the instantiated enough argument positions of p and p' respectively. If $\mu \wedge \{h_1 \leq b_1, \ldots, h_k \leq b_k\}$ is an inconsistent system of constraints (one check with the CLP(R) constraint solver) then it implies that for some argument positions $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\}$

$$\sum_{j=1}^{i_n} h_j > \sum_{j=1}^{i_n} b_j .$$

The analysis consists of two main stages: the first one is goal independent where given a logic program and the definition of a symbolic norm, it is abstracted to a CLP($\mathcal{N}$) program and the binary unfoldings of the program are computed over the size relations and instantiation domains. In the second stage an initial atomic goal is given and the system computes all the abstract calls and checks the calls with corresponding binary clauses for the termination condition. The advantage of the binary unfoldings approach is that the goal independent analysis (which is more time consuming) can be performed only once and sev-

eral queries can be tested in a relatively fast analysis. The implementation can be tuned by several parameters. For example, the choice of symbolic norm, the abstract domain for size dependencies (e.g. polyhedral approximations or monotonicity and equality constraints), and the number of iterations before applying widening in the polyhedra domain.

We have found that in general, the system performs best when tuned to apply polyhedral approximations for binary clauses of the form $h \leftarrow true$ (i.e. answer patterns) and monotonicity and equality constraints for other binary clauses. For instance the mergesort example of Figure 3 (which is often considered difficult for termination analysis) is obtained this way (and also by delaying the widening for four iterations). This is also the default setting provided by our system, and used in almost all the results shown in Figure 4. The efficiency is due to the fact that in the convex polyhedra domain iterations are kept small since only one atom for each predicate symbol is kept in the interpretation. It is also in general more accurate since this domain is able to express more complex size relations on more than two argument positions. In the recursive binary clauses we are usually interested in relations between head and body argument positions. The monotonicity and equality constraints are sufficient to capture these. Moreover such relations are sometimes lost in the widening if polyhedral approximations are used.

The analyzer has been tested on a large suit of benchmarks, including all of those mentioned in the extensive experiments described in [32]. The experiments have been performed on a one processor 167 MHZ Sparc Ultra with 256 MByte of memory running Solaris 2.5.1. The table in Figure 4 shows the results of the termination analysis for some of the programs analyzed in [32]. Most of them were used in other works that deal with termination ([15, 3, 37]). The results are ordered by the size of the program (number of clauses) which vary from two to 128 clauses. The columns of the table describe the results in the following order: the program analyzed, the norm applied (T for *term-size* and L for *list-length*), the time for the goal independent stage (in seconds), the initial abstract query (given as an abstract atom where the argument positions are abstracted as b and f for input (bound) and output (free) argument positions), the result of the analysis (Y if the system proves termination and N otherwise), the time for the query analysis (in seconds), and total analysis time (which is the sum of the times in both stages).

In general the results of the analysis are similar to the results in [32] since the same norms are used and therefore the same size relations are obtained. An exception are two cases (mergesort and permute2 from [37]) where using the polyhedral approximations domain enables us to prove termination, which cannot be proved using the monotonicity and equality constraints used in [32]. The times compared [32] are similar for small programs but considerably faster for the larger programs tested.

Program	Norm	Time	Query	Ans	Time	Total
append	L	0.24	append(b,f,f)	Y	0.05	**0.29**
			append(f,f,b)	Y	0.06	**0.30**
			append(f,b,f)	N	0.06	**0.30**
sum	T	0.19	sum(f,b,f)	Y	0.11	**0.30**
			sum(f,f,b)	Y	0.10	**0.29**
reverse	L	0.21	reverse(b,f,f)	Y	0.05	**0.26**
			reverse(b,f,b)	Y	0.06	**0.27**
ordered	L	0.06	ordered(b)	Y	0.01	**0.07**
fold	L	0.16	fold(b,b,f)	Y	0.07	**0.23**
ack	T	1.05	ack(b,b,f)	Y	0.09	**1.14**
append3	L	0.30	append3(b,b,b,f)	Y	0.08	**0.38**
			append3(f,b,b,b)	N	0.08	**0.38**
even-odd	T	0.14	even(b)	Y	0.03	**0.17**
			odd(b)	Y	0.03	**0.17**
naive_rev	L	0.33	reverse(b,f)	Y	0.08	**0.41**
permute1	L	0.27	permute(b,f)	Y	0.09	**0.36**
permute2	L	0.39	perm(b,f)	Y	0.11	**0.50**
minsort	L	0.69	minsort(b,f)	N	0.16	**0.85**
fib_t	T	0.76	fib(b,f)	Y	0.24	**1.00**
quicksort	L	1.57	qs(b,f)	Y	0.37	**1.94**
grammar	T	0.24	goal	Y	0.06	**0.30**
mergesort	L	1.85	mergesort(b,f)	Y	0.45	**2.30**
sublist	L	0.41	sublist(b,b)	N	0.07	**0.48**
queens	T	0.50	queens(b,f)	Y	0.19	**0.69**
blocks	T	0.46	tower(b,b,b,f)	N	0.09	**0.55**
turing	T	1.08	turing(b,b,b,f)	N	0.11	**1.19**
occur	T	0.63	occurall(b,b,f)	Y	0.20	**0.83**
money	T	2.14	money(f,f,f,f,f,f,f,f)	Y	0.30	**2.44**
mmatrix	T	1.18	mmultiply(b,b,f)	Y	0.21	**1.39**
			trans_m(b,f)	N	0.14	**1.32**
deriv	T	0.85	d(b,b,f)	Y	0.08	**0.93**
aiakl	T	8.60	init_vars(b,b,f,f)	Y	2.26	**10.86**
plan	T	3.25	transform(b,b,f)	N	0.41	**3.66**
credit	T	1.32	credit(b,f)	Y	0.51	**1.83**
bid	T	2.11	bid(b,f,f,f)	Y	0.61	**2.72**
browse	T	4.08	main	N	1.49	**5.57**
rdtok	T	11.44	goal	N	2.15	**13.59**
peephole	T	12.24	peephole_opt(b,f)	N	1.74	**13.98**
read	T	45.90	goal	N	4.76	**50.66**
boyer	T	8.94	tautology(b)	N	0.72	**9.66**
ann	T	20.00	go(b)	N	3.27	**23.27**

Figure 4: Termination Analysis Results

8 Conclusions

We have provided a formal semantic basis for the termination analysis of logic programs based on a notion of binary unfoldings. This provides a simple yet novel approach to the design of termination analyzers and is of practical use. To substantiate this claim we demonstrate a simple Prolog interpreter for binary unfoldings. When combined with a suitable abstract domain the interpreter provides an implementation vehicle for a termination analyzer. We have implemented a termination analyzer in this way, using two abstract domains recently described in the literature. The advantage in our approach is that the relations between consecutive calls in a computation is observed from within the semantics. Hence abstractions of the semantics can provide information from which we can reason about termination directly.

As a topic for future work, we propose to focus on the integration of recent developments in the study of termination analysis for logic programs into the semantic based approach proposed in this paper. In particular, new ideas on: inferring norms automatically [19] and on well-quasi orders and homeomorphic embedding [30].

References

[1] K. R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Computational Logic*, pages 150–176. Springer-Verlag, Berlin, 1990.

[2] K. R. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Information and Computation*, 106(1):109–157, Sept. 1993.

[3] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure prolog programs. In *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[4] F. Benoy and A. King. Inferring argument size relationships with clp(r). In *Sixth International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, 1996.

[5] M. Bezem. Characterizing Termination of Logic Programs with Level mappings. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 69–80, Cleveland, Ohio, USA, 1989.

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

[7] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in datalog programs. In *Proceedings of the Eighth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 190–199, 1989.

[8] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 571–584, Lisbon, 1989. The MIT Press.

[9] M. Codish and B. Demoen. Analysing logic programs using "`prop`"-ositional logic programs and a magic wand. *The Journal of Logic Programming*, 25(3):249–274, December 1995.

[10] M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In J. Hanus, M. Heering and K. Meinke, editors, *6th International Joint Conference on Algebraic and Logic Programming*, pages 31–45, Southhampton UK, 1997. LNCS 1290 Springer-Verlag.

[11] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[13] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Jan. 1978.

[15] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[16] D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(02):117–154, 1995.

[17] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 481–488, ICOT, Japan, 1992. Association for Computing Machinery.

[18] S. Decorte and D. De Schreye. Demand-driven and constraint-based automatic left-termination analysis for logic programs. In Naish [36], pages 78–92.

[19] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 420–436, Vancouver, Canada, 1993. MIT Press.

[20] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Comput. Sci.*, 69(3):289–318, 1989.

[21] R. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[22] M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the ACM Symposium on Applied Computing*. ACM Press, 1994.

[23] M. Gabbrielli, G. Levi, and M. C. Meo. Observable Behaviors and Equivalences of Logic Programs. *Information and Computation*, 122(1):1–29, 1995.

[24] M. Gabbrielli and M. C. Meo. Fixpoint Semantics for Partial Computed Answer Substitutions and Call Patterns. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the Third International Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 1992.

[25] A. V. Gelder. Deriving Constraints Among Argument Sizes in Logic Programs. In *Proc. of the eleventh ACM Conference on Principles of Database Systems*, pages 47–60. ACM, 1990.

[26] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 25(3):191–247, Dec. 1995.

[27] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3.* Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[28] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994.

[29] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[30] M. Leuschel. Homeomorphic embedding for online termination. In *Static Analysis, 5th International Symposium, SAS '98*, 1998. to appear.

[31] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In Naish [36], pages 63–77.

[32] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs (with detailed experimental results). Technical report, Hebrew University, Jerusalem, 1997. Available at `http://www.cs.huji.ac.il/~naomil/app_v.ps`.

[33] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2^{nd} edition, 1987.

[34] Z. Manna. *Lectures on the Logic of Computer Programming*. Society for Industrial and Applied Mathematics, 1980.

[35] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. J. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 7–21, Bonn, Germany, 1996. The MIT Press.

[36] L. Naish, editor. *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium, 1997. The MIT Press.

[37] L. Plumer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Inteligence*. Springer-Verlag, 1990.

[38] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Static Analysis: Proceedings of the Fourth International Symposium*, Lecture Notes in Computer Science 1302, pages 157–171. Springer, 1997.

[39] Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Release 3 #6*, November 1997.

[40] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 301–315, Paris, France, 1991. The MIT Press.