A semantic basis for the termination analysis of logic programs

Michael Codish *, Cohavit Taboch 1

Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, P.O. Box 653, 84105 Beer-Sheva, Israel

Received 7 February 1998; received in revised form 2 July 1998; accepted 1 December 1998

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 the 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. © 1999 Elsevier Science Inc. All rights reserved.

Keywords: Logic programs; Termination analysis; Semantics; Abstraction

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 in Ref. [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].

* Corresponding author. E-mail: mcodish@cs.bgu.ac.il.
1 E-mail: taboch@cs.bgu.ac.il.

0743-1066/99/$ – see front matter © 1999 Elsevier Science Inc. All rights reserved.
PII: S 0 7 4 3 - 1 0 6 6 ( 9 9 ) 0 0 0 0 6 - 0
We study left-termination, i.e. the universal termination of logic programs executed by means of LD-resolution, which consists of SLD-resolution combined 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 analyses for logic programs apply this common approach but focus on different aspects of proving termination of programs. Several papers [19,6,38] handle the problem of inferring norms and well-founded orders. Others [16,38,7,25,4] present techniques for computing inter-argument relations, an essential component in any termination analysis. The analysis presented in Ref. [36] shows how to infer classes of terminating queries using approximations in Natural and Boolean constraint domains. The work of Ref. [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 Refs. [40,31]. An extensive survey of the most common techniques is given in Ref. [15].

This paper focuses on the semantics basis for termination analysis. Our semantics, similar to the definition in Ref. [22] is based on the notion of binary unfoldings. The idea to use clauses as semantic objects which capture call patterns first appeared in Refs. [23,24]. A simplification of this semantics which uses binary clauses 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 also facilitates the implementation of termination analyses 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 Refs. [42,16,31] 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 Refs. [42,31,16,38], 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
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 \( \Pi, \Sigma \) and \( \mathcal{T} \). We let \( T(\Sigma, \mathcal{T}) \) denote the set of terms constructed using symbols from \( \Sigma \) and variables from \( \mathcal{T} \); \( \text{Atom} \) denotes the set of atoms constructed using predicate symbols from \( \Pi \) and terms from \( T(\Sigma, \mathcal{T}) \). A goal is a finite sequence of atoms. The set of goals is denoted by \( \text{Atom}^* \). The empty goal is denoted by \( \text{true} \). Substitutions and their operations are defined as usual. The most general unifier of syntactic objects \( s_1 \) and \( s_2 \) is denoted as \( \text{mgu}(s_1, s_2) \). A syntactic object \( s_1 \) is more general than a syntactic object \( s_2 \) denoted as \( s_2 \preceq s_1 \) if there exists a substitution \( \theta \) such that \( s_2 \hat{=} s_1 \theta \).

A clause is an object of the form \( \text{head} \leftarrow \text{body} \) where \( \text{head} \) is an atom and \( \text{body} \) is a goal. If \( \text{body} \) consists of at most one atom then the clause is said to be binary. The set of binary clauses is denoted as \( \mathcal{I} \). A binary clause can be viewed as a relation \( \mathcal{R} \) on \( \text{Atom} \). An identity clause is a binary clause of the form \( p(x) \leftarrow p(x) \) where \( x \) 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 binary clause \( C \) with an identity clause gives back the clause \( C \). We denote by \( \text{id} \) the set of identity clauses over the given alphabet.

A variable renaming is a substitution that is a bijection on \( \mathcal{T} \). Two syntactic objects \( t_1 \) and \( t_2 \) are equivalent up to renaming denoted as \( t_1 \sim t_2 \), if \( t_1 \rho = t_2 \) for some variable renaming \( \rho \). 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 \( \mathcal{I} \) is denoted as \( \mathcal{V} \).

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 $\mapsto_p \subseteq \text{Atom}^* \times \text{Atom}^*$ such that $G \mapsto_p G'$ if and only if

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

We sometimes write $G \mapsto_p G'$ to indicate explicitly the substitution $\vartheta$ 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 $G_0 \vartheta_1 \cdots \vartheta_n G_n$ is a derivation and $\vartheta = \vartheta_1 \circ \cdots \circ \vartheta_n$ then we write $G_0 \mapsto_p G_n$. If there is an infinite derivation of the form $G_0 \mapsto_p G_1 \mapsto_p \cdots$ 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 $G_0 \mapsto_p \langle A, \ldots \rangle$. We denote by $\text{calls}_P(G_0)$ the set of calls in the computations of $G_0$ with $P$. We say that $G_0 \vartheta$ is an answer for $G_0$ with $P$ if $G_0 \vartheta \text{true}$. The set of answers for $G_0$ with $P$ are denoted $\text{ans}_P(G_0)$.

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 regard to observing termination.

Definition 3.4 (calls-to relation $\leftarrow$). We say that there is a call from $a$ to $b$ in a computation of the goal $G_0$ with the program $P$, denoted $a \leftarrow_{P \leftarrow c} b$, if $a \in \text{calls}_P(G_0)$ and $b \in \text{calls}_P(a)$. When clear from the context we write $a \leftarrow b$ or $a \leftarrow \vartheta b$ to emphasize that $\vartheta$ 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.
We show that for each \( k \), there is a chain \( a_0 \leftarrow \cdots \leftarrow a_k \) 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 \( c_i \theta_0 \) such that \( G_0 \rightarrow^* \langle (c_1, \ldots, c_n) \theta_0 \rangle \) and such that \( \langle c_i \theta_0 \rangle \) 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 \( a_0 \leftarrow \cdots \leftarrow a_k \) 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 \( a_k \leftarrow a_{k+1} \) which has an infinite derivation with \( P \). Let \( g_1 = \langle b_1, \ldots, b_m \rangle \). We pick \( a_{k+1} \) to be \( b_i \theta_k \) such that \( \langle a_k \rangle \rightarrow^* \langle (b_1, \ldots, b_m) \theta_k \rangle \), and \( \langle b_i \theta_k \rangle \) has an infinite derivation. Such a consecutive state must exist since \( \langle a_k \rangle \) has an infinite derivation.

\( \square \)

### 4. Denotational semantics

As a basis for termination analysis, we adopt a simplification of the goal-independent semantics for call patterns defined in Ref. [22]. The definition is given as the fixed point of an operator \( T_P^\theta \) over the domain of **binary clauses**. Intuitively, a binary clause \( a \leftarrow b \) specifies that a call to \( a \) in a computation implies an eventual subsequent call to \( b \). A clause of the form \( a \leftarrow \text{true} \) 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, \( T_P^\theta(I) \) 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 (\( b \neq \text{true} \)), 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, also note that (3) computes the standard s-semantics.

**Definition 4.1 (binary unfoldings semantics).**

\[
T_P^\theta : \varphi(\mathcal{Z}) \rightarrow \varphi(\mathcal{Z})
\]

\[
T_P^\theta(I) = \left\{ (h \leftarrow b)^\theta \begin{array}{ll}
\text{ C = h } & \leftarrow b_1, \ldots, b_m \in P, \quad 1 \leq i \leq m, \\
\langle h_i \leftarrow \text{true} \rangle & \leftarrow c I,\\n\langle h_i \leftarrow b \rangle & \leftarrow c (I \cup \text{id}, \quad i < m \Rightarrow b \neq \text{true}\\n\end{array}
\right\}
\]

\[
\text{bin_unf}(P) = \text{lfp}(T_P^\theta)
\]

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

\[ p(X, Y) \leftarrow q(X), r(Y), p(X, Y). \]
\[ p(X, Y) \leftarrow r(X), r(Y). \]
\[ p(a, b). \]
\[ q(a), \ r(b). \]

The binary unfoldings of $P$ are evaluated as follows:

1. \[(T_P^0)^1(\emptyset) = \{ \begin{array}{l}
  p(A, .) \leftarrow q(A), \ p(A, .) \leftarrow r(A), \\
  p(a, b) \leftarrow true, \ q(a) \leftarrow true, \ r(b) \leftarrow true
\end{array} \} \]

2. \[(T_P^0)^2(\emptyset) = \{ \begin{array}{l}
  p(a, b) \leftarrow q(a), \ p(a, b) \leftarrow r(a), \\
  p(b, A) \leftarrow r(A), \ p(b, b) \leftarrow true
\end{array} \} \cup \ (T_P^0)^1(\emptyset) \]

3. \[(T_P^0)^3(\emptyset) = \{ p(a, b) \leftarrow r(b) \} \cup \ (T_P^0)^2(\emptyset) \]

4. \[(T_P^0)^4(\emptyset) = (T_P^0)^3(\emptyset) \mbox{ (fixed point)} \]

Note that since $P$ is a Datalog program (i.e. does not contain function symbols) $\mbox{bin\_unf}(P)$ is finite.

In Ref. [22] and similarly in Ref. [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. \[ \mbox{ans}_P(G) = \left\{ G^\emptyset \left| h \leftarrow true \in \mbox{bin\_unf}(P), \ \emptyset = \mbox{mgu}(G, h) \right\} \right. \]

2. \[ \mbox{calls}_P(G) = \left\{ b^\emptyset \left| h \leftarrow b \in \mbox{bin\_unf}(P), \ \emptyset = \mbox{mgu}(G, h) \right\} \right. \]

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

\[ \mbox{calls}_P(p(a, X)) = \{q(a), r(a), r(X), p(a, b), r(b)\} \]
\[ \mbox{ans}_P(p(a, X)) = \{p(a, b)\}. \]

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 $\text{bin\_unf}(P)$.

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 $\text{bin\_unf}(P)$. We show that in fact the calls-to relations for $G_0$ with $P$ and with $\text{bin\_unf}(P)$ are identical. By Proposition 4.2 (2) it follows that for any goal $G$, $\text{calls}_P(G) = \text{calls}_{\text{bin\_unf}(P)}(G)$. This implies by Definition 3.4 that

$$
P.G_0 \quad a \leftarrow b \iff a \leftarrow \text{bin\_unf}(P).G_0 \quad b. \quad \Box
$$

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 $p(a,X) \leftarrow p(a,b) \leftarrow p(a,b) \leftarrow \cdots$ 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 Fig. 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 $\text{user\_clause}(h, [b_1, \ldots, b_n])$. The interpreter can be divided conceptually into two components. On the right, the predicate $\text{tp\_beta}/0$ provides the “logic” and the inner loop of the algorithm which for each $\text{user\_clause}(\text{Head}, \text{Body})$ 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 $\text{fact}(F)$. Binary clauses are asserted as atoms of the form $\text{bin}(H,B)$. The predicate $\text{cond\_assert}/1$ asserts a derived object if it is new – namely not equivalent to any of those derived so far. The first clause in $\text{solve}/2$ 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

```
iterate \leftarrow \text{tp\_beta}, \text{fail}.
iterate \leftarrow \text{retract}(\text{flag}), \text{iterate}.
iterate:
\text{cond\_assert}(F) \leftarrow 
\text{in\_database}(F), !.
\text{cond\_assert}(F) \leftarrow 
\text{assert}(F), \text{cond\_assert}(\text{flag}).
\text{in\_database}(G) \leftarrow 
\text{functor}(G, N, A), 
\text{functor}(B, N, A), \text{call}(B), 
\text{variant}(B, G), !.
\text{tp\_beta} \leftarrow 
\text{user\_clause}(\text{Head}, \text{Body}), 
\text{solve}(\text{Head}, \text{Body}).
\text{solve}(\text{Head}, []) \leftarrow 
\text{cond\_assert}(\text{fact}(\text{Head})).
\text{solve}(\text{Head}, [B|Bs]) \leftarrow 
\text{fact}(B), 
\text{solve}(\text{Head}, Bs).
\text{solve}(\text{Head}, [B|Bs]) \leftarrow 
\text{cond\_assert}(\text{bin}(\text{Head}, B)).
\text{solve}(\text{Head}, [B|]) \leftarrow 
\text{bin}(B, C), 
\text{cond\_assert}(\text{bin}(\text{Head}, C)).
```

Fig. 1. Prolog interpreter for binary unfoldings.
database. The last two clauses compute new binary clauses. The control component, on the left, invokes iterations of \texttt{tp\_beta/0} until no new unfoldings are derived. When a new binary clause is asserted to the Prolog database, a \texttt{flag} is raised (unless the flag has already been raised). Iteration terminates when \texttt{retract(flag)} fails in the second clause indicating that nothing new was asserted in the previous iteration. Bottom-up evaluation is initiated by a call to the predicate \texttt{iterate/0} 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 Fig. 1 is coupled with a suitable notion of abstraction and termination analyses 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 analyses are based on (finite) descriptions of the binary unfolding semantics. To determine that an atomic goal \( G \) does not have an infinite computation with \( \text{bin\_unf}(P) \), 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 Refs. [42,16,31] where termination analyses 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 \( \text{CLP}(\mathcal{N}) \) with predicate symbols \( \Pi' = \Pi \), function symbols \( \Sigma' = \mathbb{N} \cup \{+/2\} \) and variables from \( \mathcal{V} \). The set of terms that can be constructed from \( \Sigma' \) and \( \mathcal{V} \), also called size expressions, is denoted \( T(\Sigma', \mathcal{V}) \). Constraints in \( \text{CLP}(\mathcal{N}) \) are conjunctions of \( \{=, \leq, \geq, <, >\} \) relations on \( T(\Sigma', \mathcal{V}) \), sometimes denoted as sets. By \( \mathcal{D} \) we denote the the domain over which computation is performed – the natural numbers with the standard interpretations of \( \{=, \leq, \geq, <, >\} \) and +. Like other constraint domains \( \text{CLP}(\mathcal{N}) \) supports the following operations:

- a test for consistency or satisfiability: \( \mathcal{D} \models \exists \mu \).
- implication (or entailment) of one constraint by another: \( \mathcal{D} \models \mu_0 \rightarrow \mu_1 \).
- the projection \( \exists_{\mathcal{V}}(\mu_0) \) of a constraint \( \mu_0 \) onto the set of variables \( \mathcal{V} \).

Clauses in \( \text{CLP}(\mathcal{N}) \) take the form \( h \leftarrow \mu, b_1, \ldots, b_n \) where \( \mu \) is a constraint, and \( h, b_1, \ldots, b_n \) are atoms constructed with predicate symbols from \( \Pi' \) 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 the following we adopt a normalized representation of abstract objects so that clause heads contain distinct variables. For example if \( p(t) \leftarrow \mu \) is a constrained atom then an alternative equivalent representation is \( p(\bar{X}) \leftarrow \mu \land (\bar{X} = t) \) where \( \bar{X} \) are distinct variables and \( (\bar{X} = t) \) is a shorthand for \( \{X_1 = t_1, \ldots, X_n = t_n\} \).
Constraints are partially ordered by entailment:

\[ \mu_1 \preceq \mu_2 \iff \emptyset \models \mu_1 \rightarrow \mu_2. \]

The ordering on CLP(\(\mathcal{M}\)) syntactic objects is induced by the partial order on constraints. For example, for two normalized constrained atoms \(a_1 = p(\bar{x}) \leftarrow c_1\) and \(a_2 = p(\bar{x}) \leftarrow c_2\),

\[ a_1 \preceq a_2 \iff c_1 \leq c_2. \]

The induced equivalence relation is denoted \(\approx\).

Our abstract domain is the power set of binary CLP(\(\mathcal{M}\)) clauses modulo this notion of equivalence \(\wp(\mathcal{F}_N/\approx)\). For notational simplicity we denote it by \(\wp(\mathcal{F}_N)\), and note that as a power set domain, disjunctive information can be expressed. This has the same effect as in Ref. [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 \(\mathcal{F}_N\).

We consider the lower power domain (or Hoare power domain [27]) with the ordering defined as:

\[ \mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \iff \forall \beta_1 \in \mathcal{I}_1 \exists \beta_2 \in \mathcal{I}_2: \beta_1 \leq \beta_2 \]

the corresponding equivalence relation:

\[ \mathcal{I}_1 \approx \mathcal{I}_2 \iff (\mathcal{I}_1 \sqsubseteq \mathcal{I}_2) \land (\mathcal{I}_2 \sqsubseteq \mathcal{I}_1) \]

induces a partial order \(\wp(\mathcal{F}_N/\approx)\).

The relation between the concrete and the abstract domains is determined by a norm function 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 Ref. [6], but variables are mapped to variables, and are equivalent to the norm based abstractions defined in Ref. [42].

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

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

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

The following table indicates several concrete terms and the result of applying the term-size and list-length symbolic norms.
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 Example 5 and in Fig. 2(b).

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

(a) The mergesort relation

(b) The abstract mergesort relation

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

(d) Size dependencies for split using polyhedral approximations

Fig. 2. The mergesort relation, its list-length abstraction, and inter-argument relations for the split relation.
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 \texttt{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).**

\[ \forall (\exists), \gamma : \phi(\exists_N) \rightarrow \phi(\exists) \]
\[ \forall (\exists) = \{ [a \leftarrow b] | a \leftarrow b \in I \} \]
\[ \gamma(\mathcal{I}) = \{ [c|\beta \in \mathcal{I}, ||c|| \leq \beta] \} \]

The following lemma is a straightforward consequence of Definition 5.2.

**Lemma 5.3.** \((\phi(\exists), \forall, \phi(\exists_N), \gamma)\) 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 Ref. [26] or in Ref. [29]. The following definition illustrates an operator for the binary unfoldings of a CLP(\(\mathcal{N}\)) program \(P_{||}\). 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 \(\land_{i=1}^n (t_i = s_i)\).

**Definition 5.4 (abstract binary unfoldings semantics).**

\[ \mathcal{T}_{P_{||}}^a : \phi(\exists_N) \rightarrow \phi(\exists_N) \]
\[ \mathcal{T}_{P_{||}}^a(\mathcal{I}) = \left\{ h \leftarrow \mu, b \left| \begin{array}{l}
C = h \leftarrow \mu_0, b_1, \ldots, b_m \in P_{||}, 1 \leq i \leq m,
\langle a_i \leftarrow \mu_{i-1} \rangle \sqsubseteq c \mathcal{I},
\langle a_i \leftarrow \mu_i, b \rangle \sqsubseteq c \mathcal{I} \cup \text{id}, i < m \Rightarrow b \neq \text{true}
\end{array} \right. \right\} \]

\[ \text{bin_unf}^a(P_{||}) = \text{lfp}(\mathcal{T}_{P_{||}}^a) \]

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

It is important to note that the domain of CLP(\(\mathcal{N}\)) binary clauses does not satisfy the ascending chain condition. As a simple example consider the evaluation of the abstraction of the \texttt{append/3} relation for the list-length norm.

**Example 5.** Consider the program \texttt{append} (on the left) and its abstract version obtained by applying the list-length norm, on the right.

\texttt{append([], Ys, Ys).} \hspace{1cm} \texttt{append(0, Ys, Ys).}
\texttt{append([X|Xs], Ys, [X|Zs]) \leftarrow} \hspace{1cm} \texttt{append(1 + Xs, Ys, 1 + Zs) \leftarrow}
\texttt{append(Xs, Ys, Zs).} \hspace{1cm} \texttt{append(Xs, Ys, Zs).}
The abstract binary unfoldings for this program are evaluated as follows:

1. \((T_{P_{\parallel}}^2)^1(\emptyset) = \{\text{append}(0, Ys, Ys), \text{append}(1 + Xs, Ys, 1 + Zs) \} \cup \text{append}(Xs, Ys, Zs)\)

2. \((T_{P_{\parallel}}^2)^2(\emptyset) = \{\text{append}(2 + Xs, Ys, 2 + Zs) \} \cup \text{append}(Xs, Ys, Zs)\)

3. \((T_{P_{\parallel}}^2)^3(\emptyset) = \{\text{append}(3 + Xs, Ys, 3 + Zs) \} \cup \text{append}(Xs, Ys, Zs)\)

4. ...

Most of the abstract interpretation based argument relations analyses for logic programs described in the literature differ primarily in the way they further approximate this abstract domain to obtain finite analyses. For example, in Refs. [42,30], a technique is described to derive affine inter-argument relations using linear equations, in Refs. [4,14] the authors propose polyhedral approximations combined with a convex hull operation as a least upper bound and a widening, and in Refs. [31,7] the authors use disjunctions of monotonicity and equality constraints.

Fig. 2 illustrates a Prolog program for mergesort (a) together with its list-length abstraction (b), and approximations of the inter-argument relations for the split predicate obtained using monotonicity and equality constraints (c) and polyhedral approximations (d).

6. A sufficient condition for termination

The classic approach to termination analysis (see for example Ref. [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 \(W\). Termination proofs for logic programs are also mostly based on well-founded orders. The objects measured are the terms that arise as arguments in the calls to predicates during a computation. The notion of a level mapping for measuring atoms was introduced in Ref. [8] and used in Refs. [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 \(|\cdot| : B_P \rightarrow \mathbb{N}\) of ground atoms to natural numbers. For \(A \in B_P\), \(|A|\) is the level of \(A\).

Most techniques that consider left-termination reason about the size of the arguments in the calls using the original program clauses. The notion of acceptability (defined in Ref. [2]) was introduced to reason about the effect of solving the body atoms to the left of a call, on that call. The notion of boundedness with respect to a level...
mapping was introduced to ensure that when a decrease in size is identified, it is over a well-founded domain.

**Definition 6.2 (acceptability).** Let $P$ be a program, $\cdot : \cdot$ a level mapping for $P$ and $I$ an interpretation of $P$ (not necessarily a Herbrand interpretation). A clause $C \in P$ is called acceptable with respect to $\cdot : \cdot$ 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 $C$ such that $I \models B_1, \ldots, B_{i-1}$ then $|A| > |B_i|$.

**Definition 6.3 (boundedness).** An atom $A$ is called bounded with respect to a level mapping $\cdot : \cdot$, if $\cdot : \cdot$ is bounded on the set of ground instances of $A$.

The required information about a program’s model is usually obtained by a separate inter-argument relations analysis. The condition of acceptability is relaxed in Ref. [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 of the consecutive calls. For acceptable and semi-acceptable programs termination is proven for any initial goal that is bounded with respect to the given level mapping. In Ref. [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 the recursive body atoms.

In our context boundedness means that terms are sufficiently instantiated so that the result of applying a symbolic norm is an integer (a ground term). We recall the notion of instantiated enough from Ref. [31] with respect to a symbolic norm which is closely related to that of rigidity (see for example Ref. [6]).

**Definition 6.4 (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 6.** The list of variables $[X_1, X_2, X_3]$ is instantiated enough with respect to the list-length symbolic norm since $\| [X_1, X_2, X_3] \|_{\text{ListLength}} = 3$. However, it is not instantiated enough with respect to the term-size norm since $\| [X_1, X_2, X_3] \|_{\text{TermSize}} = X_1 + X_2 + X_3 + 6$.

The main advantage of our semantics-based approach is that all of the information needed for a proof of termination is embedded in the binary unfoldings of the program. Since we have already shown that the termination behavior of a program $P$ is equivalent to that of $\text{bin_unf}(P)$, a proof of termination focuses directly on a finite approximation of $\text{bin_unf}(P)$.

Let us first consider a simple case. Recall the Datalog program of Example 1 and its finite set of binary unfoldings. Note that for programs which have a finite set of binary unfoldings, termination is decidable. In particular, the conditions formulated in [39] are necessary and sufficient for termination. The binary unfoldings semantics
provides another decision procedure for determining the termination of a Datalog program \( P \) and an initial goal \( G \):

- generate the (finite) set \( \text{bin} \_\text{unf}(P) \).
- generate the (finite) set of calls \( \text{calls}_P(G) \) from \( G \) and \( \text{bin} \_\text{unf}(P) \).
- \( P \) is non-terminating with \( G \) if and only if there exists a call \( q \) and a binary clause \( h \leftarrow b \) such that \( \theta = \text{mgu}(q, h) \) and \( b\theta \sim q \).

The intuition behind this algorithm is the following: For a Datalog program, the underlying alphabet contains no function symbols (besides constants) and hence only a finite number of different calls may arise in a computation (modulo renaming). If all of the pairs \( a \leftarrow a' \) in the calls-to relation involve non-equivalent atoms, then all the chains of related calls are finite and therefore all derivations are finite, too. However, if there are two equivalent calls \( a \sim a' \) such that \( a \leftarrow a' \), then the chain of related calls that connect these two calls may be duplicated in a non-terminating derivation.

The important observation here is that because there are only finitely many binary unfoldings, \( G \) is non-terminating with \( P \) if and only if there exists a call \( q \) in the set \( \text{calls}_P(G) \) and a binary clause \( c \) in the set \( \text{bin} \_\text{unf}(P) \) such that \( q \) is non-terminating with \( c \). If no pair consisting of a call \( q \) and a binary clause \( c \) generates an equivalent call pattern (in a single resolution step with \( q \) and \( c \)) then termination is guaranteed. This observation is the key to the termination condition described below.

In the general case, the set of binary unfoldings for a given program is infinite and finite approximations are used. For this reason we can only provide a sufficient condition for termination. 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 “directly 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 follows from the above discussion on programs with finite binary unfoldings, and was also observed in Ref. [31]: it is sufficient to test only those (abstract) calls and binary clauses for which unifying the call and the binary clause head results in an equivalent new call pattern. All of the above conditions are considered in Proposition 6.5.

In the following, we assume an abstract domain combining both size relations and instantiation information with respect to a given symbolic norm. Size relations are obtained as described in Section 5. Instantiation information is obtained by performing any standard groundness analysis on the abstract program, such as that based on the \( \text{Pos} \) domain of positive Boolean functions [35,11,9]. The operations on this domain consist of the standard operation on both domains. We denote by \( \text{mgu} \) the abstract most general unifier over this combined domain. Equivalence of syntactic objects is denoted by \( \sim \).

The analysis is for any initial atomic goal which is described by an initial goal description \( G_0^z \). In practice, \( G_0^z \) 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 \( \mathcal{A} \) 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 \( G_0^z \). These are determined based on (the abstract version of) Proposition 4.2.
Proposition 6.5. Let $\beta = h \leftarrow b \in B$ and $c \in C$ such that $\kappa = \text{mgu}^2(h, c)$ and that $b \kappa \approx c$. Let $i_1, \ldots, i_k$ be the argument positions which are instantiated enough both in $h \kappa$ and in $b \kappa$ (as specified by the $\text{Pos}$ component of $(h \leftarrow b)\kappa$). We denote them by $(h \kappa)_{i_1}, \ldots, (h \kappa)_{i_k}$ and $(b \kappa)_{i_1}, \ldots, (b \kappa)_{i_k}$. We also denote by $\mu$ the size relation associated with $(h \leftarrow b)\kappa$. Then, $G_0$ terminates with $P$ if for each such $\beta \in B$ and $c \in C$ there exists a function $f$ such that

$$\mu \models f((h \kappa)_{i_1}, \ldots, (h \kappa)_{i_k}) > f((b \kappa)_{i_1}, \ldots, (b \kappa)_{i_k}).$$

Note that if $\text{mgu}^2(h, c)$ exists and $b \kappa \approx c$, then the head and the body of the binary clause have the same predicate symbol. In practice the function $f$ for measuring the head and the body argument positions is usually the sum of a subset of the relevant (abstract) arguments. Also note that a different $f$ can be applied to each binary clause tested.

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 $\text{bin\_unf}(P)$ and there exists an infinite (concrete) chain of related calls generated by $G_0$ and $\text{bin\_unf}(P)$. 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 unfoldings $B$ and calls $C$ which are approximated by a single abstract binary clause $b$ and a single abstract call pattern $c$. Moreover $b$ and $c$ satisfy the assumptions given in the proposition. But since we have shown that for each such binary clause and call there is a strict decrease in size from the head to the body arguments which are instantiated enough, then this chain cannot be infinite. $\square$

Fig. 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 Ref. [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 Fig. 1 abstracted to maintain size and instantiation information expressed in CLP ($\mathcal{M}$) and $\text{Pos}$, respectively. The analyzer and the source code are available for experimentation from http://www.cs.bgu.ac.il/~mcodeh/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) [28] provided by SICStus Prolog [41]. To obtain finite analyses, the system supports two alternative strategies: the polyhedral based approach (with widening) as
described in Ref. [4]; and the use of disjunctions of equality and monotonicity constraints as described in Ref. [31]. Using either method, the analysis produces a finite set of abstract binary unfoldings for a given program. Both 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 Ref. [9] (for logic programs) and using the interpreter of Fig. 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 of monotonicity and equality constraints adapted from Ref. [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 Ref. [4]. All of the operations on this domain: projection, convex hull and widening are implemented using the constraint solver. In addition, we apply the approach described in Refs. [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 can be any function which maps bound arguments to non-negative integers. Since in most cases only a small number of argument positions participate in the proof, typically one or two, and since it is hard to tell which argument positions satisfy the well-foundedness condition, we chose to implement a simple but strong termination test. We use the constraint solver for this test 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 \( \mu \land \{ h_1 \leq b_1, \ldots, h_k \leq b_k \} \) is an inconsistent system of constraints (a simple test performed with the CLP(R) solver) then it implies that

\[
\mu \models h_1 > b_1 \lor \cdots \lor h_k > b_k
\]
This test is strong enough to capture the case in which there is a strict decrease over at least one argument position, and also cases where a decrease is detected over the sum of some argument positions since

\[ h_i + \cdots + h_{in} > b_i + \cdots + b_{in} \Rightarrow h_i > b_i \lor \cdots \lor h_{in} > b_{in} \]

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(\(\delta\)) 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. An advantage of our approach is that the goal independent analysis (which is more time consuming) can be performed only once and several different initial query patterns can be tested in a relatively fast analysis.

The implementation can be tuned by several parameters. For example, the choice of the 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 \(h \leftarrow \text{true}\) (i.e. answer patterns) and monotonicity and equality constraints for other binary clauses. For instance the analysis of the mergesort example of Fig. 3 (which is often considered difficult for termination analysis) is obtained this way (and also by delaying the widening until sufficiently meaningful size relations are generated). This is also the default setting provided by our system, and used to obtain almost all the results shown in Table 1.

The analyzer has been tested on a large suit of benchmarks, including all of those mentioned in the extensive experiments described in Ref. [32]. The experiments have been performed on a one processor 247 MHz Ultra Sparc II with 128 Mbyte of memory. Table 1 presents the results of the termination analysis for some of the programs analyzed in Ref. [32]. Most of these originate from the works described in Refs. [15,3,38]. The columns of the table describe the results in the following order:

- **Program**: The program analyzed – ordered by the number of clauses which varies from 2 to 71 clauses.
- **Norm**: The norm applied – \(T\) for term-size, \(L\) for list-length, and \(S\) for a specifically designed norm.
- **GI**: The time for the goal independent stage (in seconds).
- **Query**: 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.
- **Ans**: The result of the analysis – \(Y\) if the system proves termination, and \(N\) otherwise. If the program is known to be non-terminating then \(+\) annotates the \(N\). If the program is known to be terminating then \(-\) annotates the \(N\). These annotations are compiled from Ref. [32].
- **GD**: The time for the goal-dependent query analysis (in seconds).
- **Total**: The total analysis time (in seconds).
In general the results of the analysis are similar to those described in Ref. [32] (where the same norms are used). An exception are two cases (mergesort and permute2 from Ref. [38]) where using polyhedral approximations enables us to prove termination, which cannot be proved using the monotonicity and equality constraints. The times compared with Ref. [32] are similar for small programs but considerably faster for the larger programs tested (even when performed on a similar machine).

<table>
<thead>
<tr>
<th>Program</th>
<th>Norm</th>
<th>GI</th>
<th>Query</th>
<th>Ans</th>
<th>GD</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>L</td>
<td>0.08</td>
<td>append(b,f,f)</td>
<td>Y</td>
<td>0.01</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>append(f,f,b)</td>
<td></td>
<td></td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>append(f,b,f)</td>
<td></td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td>sum</td>
<td>T</td>
<td>0.06</td>
<td>sum(f,b,f)</td>
<td>Y</td>
<td>0.02</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sum(f,f,b)</td>
<td></td>
<td></td>
<td>0.08</td>
</tr>
<tr>
<td>reverse</td>
<td>L</td>
<td>0.07</td>
<td>reverse(b,f,f)</td>
<td>Y</td>
<td>0.01</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reverse(b,f,b)</td>
<td></td>
<td></td>
<td>0.08</td>
</tr>
<tr>
<td>ordered</td>
<td>L</td>
<td>0.03</td>
<td>ordered(b)</td>
<td></td>
<td></td>
<td>0.03</td>
</tr>
<tr>
<td>fold</td>
<td>L</td>
<td>0.06</td>
<td>fold(b,h,f)</td>
<td></td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>ack</td>
<td>T</td>
<td>0.38</td>
<td>ack(b,b,f)</td>
<td></td>
<td>0.02</td>
<td>0.40</td>
</tr>
<tr>
<td>append3</td>
<td>L</td>
<td>0.09</td>
<td>append3(b,b,b,f)</td>
<td>Y</td>
<td>0.01</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>append3(f,b,b,b)</td>
<td></td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td>even-odd</td>
<td>T</td>
<td>0.06</td>
<td>even(b)</td>
<td></td>
<td>0.02</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>odd(b)</td>
<td></td>
<td></td>
<td>0.08</td>
</tr>
<tr>
<td>naive_rev</td>
<td>L</td>
<td>0.12</td>
<td>reverse(b,f)</td>
<td></td>
<td>0.02</td>
<td>0.14</td>
</tr>
<tr>
<td>permutel</td>
<td>L</td>
<td>0.11</td>
<td>permutel(b,f)</td>
<td></td>
<td>0.02</td>
<td>0.13</td>
</tr>
<tr>
<td>permute2</td>
<td>L</td>
<td>0.15</td>
<td>perm(b,f)</td>
<td></td>
<td>0.03</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>con(b)</td>
<td></td>
<td>0.02</td>
<td>0.11</td>
</tr>
<tr>
<td>fib_t</td>
<td>T</td>
<td>0.23</td>
<td>fib(b,f)</td>
<td></td>
<td>0.05</td>
<td>0.28</td>
</tr>
<tr>
<td>sublist</td>
<td>L</td>
<td>0.14</td>
<td>sublist(b,b)</td>
<td></td>
<td></td>
<td>0.15</td>
</tr>
<tr>
<td>quicksort</td>
<td>L</td>
<td>0.45</td>
<td>q(b,f)</td>
<td></td>
<td>0.07</td>
<td>0.52</td>
</tr>
<tr>
<td>turing</td>
<td>T</td>
<td>0.29</td>
<td>turing(b,b,b,f)</td>
<td></td>
<td>0.02</td>
<td>0.31</td>
</tr>
<tr>
<td>zebra</td>
<td>T</td>
<td>0.34</td>
<td>zebra(f,f,f,f,f,f)</td>
<td>Y</td>
<td>0.03</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>houses(f)</td>
<td></td>
<td>0.03</td>
<td>0.37</td>
</tr>
<tr>
<td>grammar</td>
<td>T</td>
<td>0.09</td>
<td>goal</td>
<td></td>
<td>0.01</td>
<td>0.10</td>
</tr>
<tr>
<td>mergesort</td>
<td>L</td>
<td>0.62</td>
<td>mergesort(b,f)</td>
<td></td>
<td>0.11</td>
<td>0.73</td>
</tr>
<tr>
<td>queens</td>
<td>T</td>
<td>0.21</td>
<td>queens(b,f)</td>
<td></td>
<td>0.05</td>
<td>0.26</td>
</tr>
<tr>
<td>msort</td>
<td>L</td>
<td>0.19</td>
<td>msort(b,f)</td>
<td></td>
<td>0.02</td>
<td>0.21</td>
</tr>
<tr>
<td>huffman</td>
<td>S</td>
<td>0.21</td>
<td>huffman(b,f)</td>
<td></td>
<td>0.02</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>code(b,f,f)</td>
<td></td>
<td>0.01</td>
<td>0.22</td>
</tr>
<tr>
<td>occur</td>
<td>T</td>
<td>0.20</td>
<td>occurall(b,b,f)</td>
<td></td>
<td>0.04</td>
<td>0.24</td>
</tr>
<tr>
<td>money</td>
<td>T</td>
<td>0.80</td>
<td>money(f,f,f,f,f,f,f)</td>
<td>Y</td>
<td>0.07</td>
<td>0.87</td>
</tr>
<tr>
<td>deriv</td>
<td>T</td>
<td>0.43</td>
<td>d(b,b,f)</td>
<td></td>
<td>0.02</td>
<td>0.45</td>
</tr>
<tr>
<td>mmatrix</td>
<td>T</td>
<td>0.39</td>
<td>mmmultiply(b,b,f)</td>
<td>Y</td>
<td>0.04</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>trans_m(b,f)</td>
<td></td>
<td></td>
<td>0.42</td>
</tr>
<tr>
<td>blocks</td>
<td>T</td>
<td>0.18</td>
<td>tower(b,b,b,f)</td>
<td></td>
<td></td>
<td>0.21</td>
</tr>
<tr>
<td>aiaa1</td>
<td>T</td>
<td>2.20</td>
<td>int_vars(b, f, b, f, f)</td>
<td>Y</td>
<td>0.41</td>
<td>2.61</td>
</tr>
<tr>
<td>plan</td>
<td>T</td>
<td>0.73</td>
<td>transform(b,b,f)</td>
<td></td>
<td>0.10</td>
<td>0.83</td>
</tr>
<tr>
<td>credit</td>
<td>T</td>
<td>0.50</td>
<td>credit(b,f)</td>
<td></td>
<td>0.13</td>
<td>0.63</td>
</tr>
<tr>
<td>bid</td>
<td>T</td>
<td>0.77</td>
<td>bid(b,f,f,f)</td>
<td></td>
<td>0.19</td>
<td>0.96</td>
</tr>
<tr>
<td>browse</td>
<td>T</td>
<td>1.14</td>
<td>main</td>
<td></td>
<td>0.40</td>
<td>1.54</td>
</tr>
<tr>
<td>rdtok</td>
<td>T</td>
<td>3.30</td>
<td>goal</td>
<td></td>
<td>0.45</td>
<td>3.75</td>
</tr>
<tr>
<td>peephole</td>
<td>T</td>
<td>4.49</td>
<td>peephole_opt(b, f)</td>
<td>N+</td>
<td>0.54</td>
<td>5.03</td>
</tr>
</tbody>
</table>
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 [19,18].

Acknowledgements

This work was supported by a grant from the Israel Science Foundation. We thank Florence Benoy, Andy King, Naomi Lindenstrauss and Yehoshua Sagiv for the many useful discussions on this work.

References


S. Decorte, D. De Schreye, Demand-driven and constraint-based automatic left-termination analysis for logic programs, in Ref. [37], pp. 78–92.


