Practical Verification of Logic Program Termination

Citation

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364644

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story
The Harvard community has made this article openly available. Please share how this access benefits you. Submit a story.

Accessibility
Practical Verification of Logic Program Termination

Brian Leo Sapozhnikov

supervised by
Professor Stephen Chong

A thesis submitted in partial fulfillment of the requirements for the joint degree of Bachelor of Arts in Computer Science and Mathematics with Honors

Harvard University
Cambridge, Massachusetts
March 29, 2019
Abstract

Datalog is a simple and canonical logic programming language, variants of which are being used for data analytics and program analyses. While termination is guaranteed for programs written in pure Datalog, some variants include language features that make them Turing-complete. Krishnamurthy et al. [1996] introduce an algorithm for checking whether a program with these additional language features will terminate. We implement this algorithm and evaluate its practicality for real-world programs. We find that the algorithm does have a potential to be practical, though the current implementation does not reach this potential. We also modify the theoretical underpinnings of the algorithm’s correctness, allowing us to formally demonstrate the validity of our implementation. Finally, the results of our implementation ground a discussion of extensions of the algorithm that may lead to truly practical termination-checking for this domain.
Acknowledgements

I am deeply grateful to Professor Stephen Chong for all his guidance and mentorship throughout my experiences as an undergraduate in computer science, from taking his courses to conducting research under his supervision and, of course, writing this thesis.

I would also like to sincerely thank Aaron Bembenek, whose thoughtful advice and detailed feedback has been invaluable throughout the research and thesis-writing processes.

Finally, I would like to thank Professor Stuart Shieber and Professor Harry Lewis for agreeing to be my thesis readers, as well as for all the wisdom and mentorship they have each shared with me over the past four years both in and out of the classroom.
1 Introduction

In logic programs we have a clean separation between the logic, which specifies those facts and relations that define the meaning of the program, and the control, which is the way that the program is evaluated [Kowalski, 1979]. Datalog is a simple and canonical such language, and its variants are being used for data analytics [Aref et al., 2015, Seo et al., 2013, Shkapsky et al., 2016] and program analyses [Bravenboer and Smaragdakis, 2009, Madsen et al., 2016, Scholz et al., 2016, Whaley et al., 2005]. Some of these Datalog variants are extended with features that make the language more powerful, more useful, and Turing-complete [Aref et al., 2015, Madsen et al., 2016, Scholz et al., 2016].

We are interested in termination-checking programs written in these Datalog variants. Not only do we want to know whether code written by a programmer will terminate, but termination-checking is also important for extending optimizations to these variants. There are some optimizations that can be applied to Datalog, and often we would like to apply these optimizations to the Datalog variants as well. For some programs, however, these optimizations may cause the evaluation of the program to run forever rather than terminate. This motivates checking whether the evaluation of a given program, for example one that has been transformed by an optimization, will terminate. In Turing-complete languages, it cannot be algorithmically decided whether an arbitrary program will terminate. We can, however, determine that programs with certain properties will terminate, and we can write algorithms that check to see if a given program has these properties. Doing so grants us formal assurance that many of our programs will terminate, making it easier to reason about and optimize the execution of those programs.

A paper by Krishnamurthy et al. [1996] describes an algorithm for termination-checking Extended Datalog, a simple Datalog variant that nicely captures the key language feature that typically causes non-termination in Datalog variants. Krishnamurthy et al.’s algorithm looks at how values change throughout the evaluation of the program and tries to find a proof in the input program that the evaluation will eventually terminate. Their paper also includes a proof of correctness of the algorithm that establishes a connection between the algorithm’s results on an input program and properties that this program must satisfy, and a connection between these properties and the termination of the program. We will show that Krishnamurthy et al.’s definition of these properties is too strong even to satisfy the simple examples they describe in their paper. We offer a weaker definition that allows for a reasonable implementation of their algorithm and reformulate their proof to maintain the connection between these properties and the algorithm’s safety.

In this paper we focus on FormuLog, which extends Datalog with several features that facilitate its use for writing program analyses [Bembenek and Chong, 2018]. We implement the algorithm described by Krishnamurthy et al. and adapt it specifically for FormuLog. Our implementation is able to correctly determine that several complex FormuLog programs terminate, though it fails to validate many other programs that also terminate. There are several ways in which we believe the algorithm and its implementation can be extended to reason about a larger class of safe programs,
thus making the algorithm more practical. Overall, we claim that our weakening of a
definition used in Krishnamurthy et al.’s paper allows us to prove the correctness of our
implementation, and that this implementation demonstrates the potential practicality
of this termination-checking algorithm. We also argue that further extensions of the
algorithm would go a long way in achieving a truly practical termination-checker for
real-world programs.
We begin by presenting some preliminary concepts and definitions (§2). We then
present an overview of the termination-checking algorithm (§3), which sets us up to
prove its correctness (§4). We then discuss our particular implementation (§5) and
examine it results on various programs, which will give us a better understanding of
potentially powerful extensions of the algorithm (§6). We close with a discussion of
related works on the termination-checking of logic programs (§7) and some concluding
remarks (§8).

2 Preliminaries
2.1 Logic Programs

Definition 1. We use the following definitions when discussing logic programs:

1. A term is a constant, a variable, or an n-ary constructor symbol (e.g. cons)
followed by n terms in parentheses. We use capital letters such as X, Y, and N
for variables in logic programs and metavariables t, s to range over terms.

2. An atom is an n-ary predicate symbol (e.g. p) followed by n terms in parentheses.
We use lowercase letters such as p, q, and b for predicate symbols in logic programs
and metavariables p, q to range over predicate symbols.

3. A rule has the form p(t_0) :- q_1(t_1), q_2(t_2), ..., q_n(t_n), where p(t_0), q_i(t_i)
are atoms. p(t_0) is the rule head and q_1(t_1), q_2(t_2), ..., q_n(t_n) is the rule
body. We use the metavariable R to range over rules.

4. A base fact is a rule with an empty body, e.g. p(3, 5). In general, the term
“rule” is used to refer to a rule that is not a base fact.

A Datalog program, then, is a collection of base facts and rules. The predicates
which do not appear in any rule heads (except in base facts) are called extensinal
database predicates, or EDB predicates. The other predicates in the program,
those that do occur in at least one rule head, are called intensional database predi-
cates, or IDB predicates.

A Datalog program must have finitely many base facts and rules. It also may not
have any n-ary constructor symbols (for n > 0). The result of these restrictions is that
termination for Datalog programs is decidable. In fact, given the well-formedness
assumption described below, all Datalog programs terminate.

An Extended Datalog program, however, may have infinitely many base facts,
so long as it has finitely many rules and predicate symbols. This extension is enough
to make the language Turing-complete and therefore the question of termination un-
decidable. It is also worth noting that allowing infinitely many base facts allows us to
model complex terms (though these are not directly allowed in Extended Datalog). For example, we can use an infinite 3-ary EDB predicate \texttt{cons} and a constant \texttt{nil} to model lists of integers, where the base fact \texttt{cons}(X, Y, Z) represents the fact that Z is the list formed by prepending the integer X to the list Y. Concretely, \texttt{cons}(3, \texttt{nil}, Y) represents the list \(Y = [3]\), and \texttt{cons}(5, Y, Z) represents the list \(Z = [5, 3]\).

We revisit this and relate it to our algorithm implementation in §5.1.1.

Also note that since Datalog is a special case of Extended Datalog, every Datalog program is also an Extended Datalog program. Thus, any definitions and claims we make for general Extended Datalog programs also apply to Datalog programs.

**Definition 2. (Stratification)**

If \(P\) is a logic program, then the stratification \(\mathcal{S}\) of \(P\) is an ordered partition of the set of IDB predicate symbols in \(P\) such that if \(p \in S_1 \in \mathcal{S}\) and \(q \in S_2 \in \mathcal{S}\) and there is a rule in \(P\) in which \(p\) occurs in the body and \(q\) occurs in the head, then either \(S_1 = S_2\) or \(S_1\) precedes \(S_2\) in \(\mathcal{S}\). The elements of \(\mathcal{S}\) are called strata.

**Definition 3. (Well-Formedness and Well-Modedness)**

We say that an Extended Datalog program is well-formed if every variable in the head of a rule also appears in the body of that rule. This is also known as the range restriction.

Suppose we have an Extended Datalog program rule with an EDB predicate instance \(p(t_1, t_2, \ldots, t_n)\) in the body. Informally, we can think of some of the argument positions in this instance as comprising the “input” to \(p\) and the others as comprising the “output”. The predicate instance is well-moded if there are always finitely many “outputs” when looking up base facts with predicate instance \(p\) using some fixed “input”. We will now formalize this notion.

Let \(I \subseteq \{1, 2, \ldots, n\}\) be the set of argument positions \(i\) for which either \(t_i\) is a constant or \(t_i\) is a variable which also occurs to the left of this predicate instance in that rule body. \(I\) specifies which arguments to \(p\) comprise the “input”. Now let \(V\) be the set of variables corresponding to \(I\), i.e. \(V = \{t_i \mid i \in I, t_i\ is\ a\ variable\}\). If \(\theta\) is any function from \(V\) to constants and \(i \in I\), define \(\theta(t_i) = c\) if \(t_i\) is some constant \(c\) and \(\theta(t_i) = \theta(X)\) if \(t_i\) is some variable \(X \in V\). We call \(\theta\) a substitution function because it substitutes constants in for terms. Note that \(\theta(t_i)\) is a constant.

Now let \(S_0\) be the set of base facts with predicate symbol \(p\). For any \(\theta\) define \(S_\theta := \{p(s_1, s_2, \ldots, s_n) \mid i \in I \Rightarrow s_i = \theta(t_i)\}\) \(\subseteq S_0\). Informally, \(S_\theta\) is the subset of base facts consistent with a given input to \(p\); it represents all possible “outputs” for a given “input”. If \(S_\theta\) is finite for all substitution functions \(\theta\), then we say that this is a well-moded predicate instance. A program is well-moded if every EDB predicate instance is well-moded.

Every Datalog program is trivially well-moded, and (as we mentioned above) every well-formed Datalog program will terminate. Going forward, we will assume that all programs we deal with are well-formed and well-moded.

6
2.2 Safety and Effective Computability

Here we give an overview of the model of evaluation in which we are working, and we define the notions of safety and effective computability formally.

In this paper we are considering the setting of bottom-up evaluation. We iteratively build up a collection of derived facts as follows:

1. We start with no derived facts and only base facts (both of which will be referred to as facts).

2. If there is a rule \( p(t_1, \ldots, t_n) :- q_1(s_{1,1}, \ldots, s_{1,n_1}), \ldots, q_m(s_{m,1}, \ldots, s_{m,n_m}) \) and a substitution \( \theta \) from variables in the rule to constants such that:
   - for each body atom \( q_i(s_{i,1}, \ldots, s_{i,n_i}), q_i(\theta(s_{i,1}), \ldots, \theta(s_{i,n_i})) \) is a fact
   - \( p(\theta(t_1), \ldots, \theta(t_n)) \) is not a fact
   then we add \( p(\theta(t_1), \ldots, \theta(t_n)) \) to our collection of derived facts.

3. Repeat step (2) until there is no such rule.

We call the resulting collection of derived facts the result of the evaluation. We say that the program is effectively computable if its evaluation terminates. Note that if it does terminate, then it must do so with finitely many derived facts. We say that these derived facts satisfy the program because if we start with these derived facts then step (2) above will not cause the derivation of any new facts. We say that a program is safe if there exists any finite set of facts that satisfy the program. Note that safety does not require that the bottom-up model of evaluation described here ever terminate. However, given our assumptions about programs it is true that a program is safe if and only if it is effectively computable. This follows from the definition of strong safety in Krishnamurthy et al. [1996] and their Theorem 3.2, but we will not prove it here as this is only tangential to our concern in this paper.

Now that we have formally introduced base facts and derived facts, we can introduce another useful definition: a fact \( p(\bar{t}) \) with predicate symbol \( p \) is called a \( p \)-fact. Each predicate symbol has an associated arity, and we require that \( \bar{t} \) have the same arity. Each predicate symbol also has an associated type signature, e.g. perhaps \( p \) is a 2-ary predicate whose first argument is an integer and second argument is a boolean. We therefore require that \( \bar{t} \) match the type signature of \( p \), e.g. we would require that the first element of \( \bar{t} \) be an integer and the second be a boolean.

3 Algorithm Overview

The algorithm uses a structure called an argument mapping to reason about sequences of rule applications and understand how a fact can lead to the derivation of other facts. By leveraging inferred information about how terms within each rule application relate to one another, it searches for information about how derived facts relate to each other. For example, it might be able to prove that a \( p \)-fact \( p(t_1) \) can only lead to the derivation of a \( p \)-fact \( p(t_2) \) if \( t_1 < t_2 \). Argument mappings also have information about lower and upper bounds of terms, e.g. perhaps the argument to some predicate \( p \) in a derived \( p \)-fact must always be at most 100. By inferring, accumulating, and reasoning about
such information within and across argument mappings, the algorithm is sometimes able to verify that a program is safe. Below we describe argument mappings and the algorithm’s conditions for verification in more detail.

3.1 Argument Mappings

An in-depth explanation of argument mappings can be found in Krishnamurthy et al. [1996] – here we include just enough explanation for our purposes.

Let $P$ be some program, then for every rule $R$ with head predicate instance $p(i)$ and body predicate instance $q(s)$ we can write $p \overset{R}{\leftarrow} q^i$, where $i$ indicates the position of the body predicate instance in the rule body (as there may be multiple predicate instances of $q$ in the body of a rule). For expositional simplicity, we will often drop the index and simply write $p \overset{R}{\leftarrow} q$

For every IDB predicate instance $p(t_1, \ldots, t_n)$ we can associate an argument mapping component defined as a set of $n$ distinct graph nodes (with no edges). Each node represents an argument position for this predicate instance, and we can label each node with its predicate symbol (which is the same for all nodes within an argument mapping component, in this case $p$) and its corresponding argument position $1, 2, \ldots, n$. We use the notation $\langle p, i \rangle$ for a node corresponding to the $i$th argument position of predicate $p$. Below is an example of an argument mapping component for a 2-ary predicate:

\[
\begin{align*}
\langle p, 1 \rangle \\
\langle p, 2 \rangle
\end{align*}
\]

Now suppose we have a sequence of rules in $P$ such that $p_1 \overset{R_1}{\leftarrow} p_2 \overset{R_2}{\leftarrow} p_3, \ldots, p_n \overset{R_n}{\leftarrow} p_1$, where each $p_i$ is distinct. We may also write this sequence as $p_1 \overset{R_1}{\leftarrow} p_2 \overset{R_2}{\leftarrow} p_3 \cdots \overset{R_n}{\leftarrow} p_1$. Note that, as the direction of the arrows indicate, these rules would be applied in right-to-left order since the head of each rule appears to the left of the arrow. This sequence induces a graph we call a cyclic argument mapping of $p_1$, which we often refer to simply as an argument mapping. The nodes of the argument mapping are the union of the argument mapping components of each predicate instance in the sequence. As a technical convention, we include separate components for both instances of $p_1$. We call the argument mapping component corresponding to the $p_1$ instance in $R_1$ the second or last $p_1$ component (consistent with the right-to-left application order of the rules in the sequence), and the other the first $p_1$ component. So, there is a node for each argument position of each predicate instance in the sequence of rules. Note that by construction, for every pair of adjacent components $p_i$ and $p_{i+1}$ there is a corresponding rule $R_i$. We say that an argument mapping contains a predicate $p$ if it contains a component associated with an instance of $p$, and we say then that $p$ occurs on the argument mapping.
Below is an example of two program rules and the argument mapping corresponding to the rule sequence \( p \leftarrow R_1 \) \( q \leftarrow R_2 \) \( p \). We will refer to and explain the details of this example as we present them formally in this section. In the rules below, \( b \) is a finite 1-ary EDB predicate and \( \text{succ} \) is an infinite 2-ary EDB predicate which holds of two values \( X, Y \) if and only if \( Y = X + 1 \), i.e. \( Y \) is the successor of \( X \). We label the rules \( R_1 \) and \( R_2 \) for expositional clarity.

\[
\begin{align*}
R_1: & \quad p(X, Y) :- q(Y, X) . \\
R_2: & \quad q(X, Y) :- p(U, V), \text{succ}(X, V), \text{succ}(U, Y), b(X).
\end{align*}
\]

An argument mapping has both undirected edges and directed arcs. Intuitively, an undirected edge between two nodes represents the fact that the terms corresponding to those nodes are equal. An arc from a node in one component to a node in an adjacent component represents the fact that the corresponding term of one node is strictly greater than the other, i.e. the application of the rule corresponding to the pair of adjacent components either increases or decreases the value of one of the terms. There are always edges connecting corresponding nodes between the two \( p_1 \) components, i.e. between the node corresponding to the \( i \)th term in the first \( p_1 \) instance and the node corresponding to the \( i \)th term in the second \( p_1 \) instance (for each argument position \( i \)). Otherwise, edges and arcs are found only between nodes of adjacent components.

Let \( X \) be an argument to \( p_i \) and \( Y \) an argument to \( p_{i+1} \) in \( R_i \). If we can show from \( R_i \) that the value of \( X \) is always equal to the value of \( Y \), then we add an edge between the corresponding nodes in the argument mapping. If we can show that the value of \( X \) is always strictly greater than the value of \( Y \), then we add an arc from \( X \) to \( Y \). We call such an arc an **increasing arc**. If we can show that the value of \( Y \) is always strictly greater than the value of \( X \), then we add an arc from \( Y \) to \( X \). We call such an arc a **decreasing arc**. In the example above, we see undirected edges to represent the fact that in \( R_1 \) the first argument to the head predicate is equal to the second argument to the body predicate and the second argument to the head predicate is equal to the first argument to the body predicate. We see arcs (a decreasing arc to \( \langle q, 1 \rangle \) and an increasing arc from \( \langle q, 2 \rangle \)) to represent the fact that in \( R_2 \), the values of \( V \) and \( Y \) are, respectively, greater than the values of \( X \) and \( U \) because of the two \( \text{succ} \) predicate instances in the rule body.

Nodes in an argument mapping can also be marked as bounded above and/or below. Suppose \( X \) is an argument to \( p_i \) in \( R_{i-1} \) and \( Y \) is the corresponding argument to \( p_i \) in \( R_i \). Let \( N \) be the corresponding node in the argument mapping. If we can show from \( R_{i-1} \) that \( X \) is always bounded below (resp. above) by some constant, then we mark \( N \) as bounded below (resp. above). If we can show from \( R_i \) that \( Y \) is always bounded below (resp. above) by some constant, then we mark \( N \) as bounded below
(resp. above). In our example above, the node \( \langle q, 1 \rangle \) is colored black to indicate that it is bounded both above and below. We know this because of the \( b(X) \) predicate instance in the body of \( R_2 \).

A cycle in an argument mapping is a cycle in the graph such that there is exactly one node of the cycle for each component in the argument mapping. A cycle is strictly increasing if it contains at least one increasing arc and no decreasing arcs. A cycle is strictly decreasing if it contains at least one decreasing arc and no increasing arcs. A cycle is bounded below (resp. above) if it contains at least one bounded-below (resp. -above) node. In our example above we have an increasing cycle through \( \langle p, 1 \rangle \) and a decreasing, bounded-above, and bounded-below cycle through \( \langle p, 2 \rangle \).

3.2 Conditions for Verification

The algorithm verifies a program iff it can verify every stratum of that program. It verifies a stratum \( S \) iff it can verify every predicate in that stratum. It verifies a predicate \( p \in S \) iff it can associate a certain kind of property, called a well-founded property, with \( p \). Well-founded properties are defined and discussed in the section on proving the algorithm’s correctness, but here we simply describe the technique the algorithm uses to determine that there is such a property for \( p \). We begin with an example that demonstrates this technique, followed by a formal description.

Consider the following program rules, in which \( b \) is a finite 1-ary EDB predicate and \( \text{succ} \) is an infinite 2-ary EDB predicate as in the example in §3.1: \( \text{succ} \) holds of two values \( X, Y \) if and only if \( Y = X + 1 \). We label the rules below for expositional clarity.

\[
\begin{align*}
R_1: & \quad p(X, Y) :- q(Y, X). \\
R_2: & \quad q(X, Y) :- p(U, V), \text{succ}(X, V), \text{succ}(U, Y), b(X). \\
R_3: & \quad p(X, Y) :- p(X, V), \text{succ}(Y, V), b(Y). \\
R_4: & \quad q(X, Y) :- q(U, Y), \text{succ}(X, U), b(X). 
\end{align*}
\]

Note that the predicates \( p \) and \( q \) are in the same stratum because each one occurs in the body of a rule in which the other occurs in the head. Let’s begin by considering the argument mapping corresponding to the rule sequence \( p \leftarrow R_1 q \leftarrow R_2 p \). This is the same rule sequence we saw in the example from §3.1; we reproduce the argument mapping below:

This argument mapping contains a decreasing, bounded-below cycle through \( \langle p, 2 \rangle \). Intuitively, this means that applying the corresponding rule application to some \( p \)-fact causes the second argument of \( p \) to decrease while remaining above some lower bound.
Next we look for a similar cycle in all other argument mappings of \( p \). In this example there is only one other such argument mapping: the one corresponding to the rule sequence \( p \leftarrow_{R_1} p \). We show the argument mapping below:

![Argument Mapping](image)

Again, we see a decreasing, bounded-below cycle through \( (p, 2) \), reflecting the EDB predicates in the body of that rule.

Note that in our first argument mapping, the decreasing cycle we have been interested in contains the node \( (q, 1) \). We therefore look at argument mappings of \( q \) that do not include \( p \). (We exclude argument mappings that include \( p \) for a technical reason which will become clearer in the formal description below.) In this example there is only one such argument mapping: the one corresponding to the rule sequence \( q \leftarrow_{R_1} q \). We show this argument mapping below:

![Argument Mapping](image)

Notice the decreasing, bounded-below cycle through \( (q, 1) \) in this argument mapping. At this point, there are no more argument mappings to consider in checking predicate \( p \). Because each of the considered argument mappings has a corresponding desired cycle, the algorithm would at this point verify \( p \).

More generally, then, the algorithm determines that there is a well-founded property for predicate \( p \) in stratum \( S \) when:

1. There is some argument position \( i \) and cycle \( C = ((p, i), \ldots, (p, i)) \) in an argument mapping of \( p \) through the node corresponding to argument position \( i \).
2. \( C \) is either strictly increasing and bounded above, or strictly decreasing and bounded below. Without loss of generality we assume that it is strictly increasing and bounded above.
3. Every argument mapping of \( p \) contains a cycle through its \( (p, i) \) nodes that is also strictly increasing and bounded above. We refer to these cycles as primary \( (p, i) \)-cycles.
4. If \( (q, j) \) occurs on a primary \( (p, i) \)-cycle (or on the original cycle \( C \)), let \( M \) denote the set of argument mappings of \( q \) on which \( p \) does not occur. Then every argument mapping \( M \in M \) contains a cycle through its \( (q, j) \) nodes that is also strictly increasing and bounded above. We refer to these cycles as \( (p, i) \)-secondary \( (q, j) \)-cycles that exclude \( \{p\} \).
5. If \( \langle r, k \rangle \) occurs on a \( \langle p, i \rangle \)-secondary \( \langle q, j \rangle \)-cycle that excludes a set of predicates \( A \), let \( M' \) denote the set of argument mappings of \( r \) on which no elements of \( A \) occur. Then every argument mapping \( M' \in M' \) contains a cycle through its \( \langle r, k \rangle \) nodes that is also strictly increasing and bounded above.

6. And so on, continuing with \( \langle p, i \rangle \)-secondary \( \langle r, k \rangle \)-cycles that exclude \( A \cup \{ q \} \).

Intuitively, argument mappings correspond to sequences of rule applications, and these conditions imply that for every sequence of rule applications starting at a predicate instance \( p(t_1, \ldots, t_n) \) and ending at a predicate instance \( p(s_1, \ldots, s_n) \), after the first rule application there is a term \( t_i' \) that is always greater than or equal to \( t_i \), after the second rule application there is a term \( t_i'' \) that is always greater than or equal to \( t_i' \), and so on, ending at \( s_i \), and that at least one of these intermediate terms is strictly greater than the previous one. This means that \( s_i \) must always be greater than \( t_i \), i.e. any such sequence of rule applications increases the value in the \( i \)th argument position of \( p \). In the proof of correctness, we formalize this intuition and show why it guarantees safety.

Also, note that the recursive description given is well-defined and will itself terminate because at each step we are increasing the set of predicates that must be excluded from the argument mappings. Since there are only finitely many predicates in the stratum (and the program in general), eventually there will be no argument mappings that exclude the set \( A \).

4 Proof of Correctness

Now that we have formally described the algorithm, we will present a proof of its correctness. After some preliminaries, we present Krishnamurthy et al.’s definition of a well-founded property and show that it is too strong. We then offer our own definition and show that it solves this issue. We are then able to formally relate this definition to the algorithm, setting us up to prove the algorithm’s correctness.

4.1 Preliminary Definitions and Conventions

4.1.1 Conventions

Here we establish some terms and notation that will be convenient to use going forward.

A **predicate over domain** \( D \) is a function \( R : D \rightarrow \{0, 1\} \). If \( R(d) = 1 \) then we say that \( d \) **satisfies** \( R \). This should not be confused with a predicate in a program, though the conceptual connection between the two should be clear.

A **derivation tree** has a derived fact (of an IDB predicate) as its root and base facts (of EDB predicates) as leaves. A **path** refers to a path in a derivation tree. It begins at a leaf and ends at the tree’s root. A **prefix** of a path refers to a continuous portion of a path that begins at the same point as the path, but may not end at the root. A **segment** refers to a continuous portion of a path that may not begin at a leaf or end at the root.
4.1.2 Canonical Program

The algorithm operates on programs in a certain canonical form. To describe our notion of a canonical program, we first describe a special predicate: the unification predicate. This is an EDB predicate on two arguments that holds if the two arguments are equivalent. The unification of terms \( t_1 \) and \( t_2 \) is written \( t_1 = t_2 \). For example, \( 5 = 5 \) is a base fact, while \( 5 = 6 \) is not. Any predicate other than the unification predicate is referred to as a normal predicate.

If we have a program rule \( R \), we can replace it with an equivalent rule such that the arguments to every normal predicate instance is a variable. We can do this by adding a new unification rule \( X = t \) to the body of \( R \), where \( X \) is a fresh variable and \( t \) is the original argument term. We can also require that every unification has a variable as its first argument (the left-hand side of the unification), and we can achieve this by adding a new unification rule \( X = t \) to the body, where \( X \) is a fresh variable and \( t \) is the original first argument to the unification. By applying this transformation to every rule, we get an equivalent canonical program for every program. In theoretical discussion of the algorithm, we therefore assume without loss of generality that every program is in this canonical form. In our actual implementation, this invariant is established by preprocessing the input program into canonical form, and the rest of the implementation leverages this invariant for clearer, concise code.

4.2 Krishnamurthy et al.’s Well-Founded Property

While the proof of correctness of the original algorithm in Krishnamurthy et al. [1996] is sound, its definition of a well-founded property is too strong to apply to the inference that would be required for a practical implementation of the algorithm. Here we state the original definition of well-foundedness and prove that a simple and important property that should be well-founded is not so under that definition.

**Definition 4.** A property over domain \( D \) is a predicate over sequences of elements from \( D \).

**Definition 5.** A property over domain \( D \) is K-well-founded if the following holds:

1. There is a total order \( \succeq \) on \( D \) such that for all \( d_1, d_2 \in D \), if \( d_1 \succeq d_2 \) then \( \{ d | d_1 \succeq d \succeq d_2 \} \) is finite.
2. If \( d_1, d_2, \ldots \) satisfies the property, then \( d_{i+1} \succeq d_i \) for all \( i \in \mathbb{Z}^>0 \).
3. \( \exists d \in D \) such that if \( d_1, d_2, \ldots \) satisfies the property, then \( d \succeq d_i \) for all \( i \in \mathbb{Z}^>0 \).

Let \( p \) be some fixed predicate symbol of interest, then the relevant domain \( D \) for the algorithm’s proof of correctness is the set of tuples \( \vec{i} \) such that \( p(\vec{i}) \) is a valid \( p \)-fact (see the definition of a path satisfying a property in §4 of Krishnamurthy et al.).

Now suppose facts of \( p \) have the form \( p(\text{int, int}) \), and let \( D = \mathbb{Z} \times \mathbb{Z} \). Consider the property \( \mathcal{R} \) over \( D \) that holds of sequences of tuples in which the first element of the tuples is strictly increasing and bounded above by 100, i.e.

\[
\mathcal{R}((a_1, b_1), (a_2, b_2), \ldots) := \forall i \in \mathbb{Z}^>0. (100 \geq a_{i+1} > a_i)
\]
This property is important because it and similar properties are the ones involved in verifying many of the examples in Krishnamurthy et al.’s paper and in our own work. However, this property is not K-well-founded:

**Claim 1.** \( R \) is not K-well-founded.

**Proof.** Assume towards contradiction that \( R \) is K-well-founded, and let \( \geq \) be the corresponding total order.

Let \( b \in \mathbb{Z} \) be given, then the sequence \((1, 1), (2, b), (3, 1)\) clearly satisfies \( R \). Thus \((3, 1) \geq (2, b) \geq (1, 1)\).

However, since this holds for all integers \( b \), we have \( \{2\} \times \mathbb{Z} \subseteq \{ c \mid (3, 1) \geq c \geq (1, 1)\} \). This contradicts the finiteness requirement in the definition of K-well-foundedness.

\[ \square \]

### 4.3 Weakened Definition of Well-Founded Property

We offer an alternative definition of a well-founded property and show that the property discussed in the previous section is well-founded according to this definition. This allows the algorithm to work correctly.

**Definition 6.** A property \( R \) over domain \( D \) is **well-founded** if there is an order \( > \) on \( D \) such that the following holds:

1. \( > \) is antisymmetric and transitive
2. If a sequence \((d_1, d_2, \ldots)\) satisfies \( R \) then \( d_{i+1} > d_i \) for all \( i \in \mathbb{Z}^{>0} \)
3. \( \forall d \in D, \exists b \in \mathbb{Z}^{\geq0} \) such that any sequence that starts at \( d \) and satisfies \( R \) has length at most \( b \).
4. If the sequence \((d_1, d_2, \ldots, d_n)\) satisfies \( R \) and the sequence \((d_n, d_{n+1}, \ldots, d_{n+m})\) satisfies \( R \), then the sequence \((d_1, d_2, \ldots, d_{n+m})\) satisfies \( R \).

The main modification to the definition is that our definition allows for infinitely many elements in between two fixed elements, so long as the length of satisfying sequences is bounded once the first element of the sequence is fixed. As we will see, this makes the natural and important properties over tuples well-founded when they were not before. Our implementation of the algorithm requires that these properties be well-founded, and we will see later that we can still prove the correctness of the algorithm using this weaker definition of well-foundedness.

Our definition also adds the requirement that satisfaction of the property be transitive. This is a technical requirement that makes our proof more convenient and holds of the properties we are interested in.

Now consider the property \( R \) from before:

\[ R((a_1, b_1), (a_2, b_2), \ldots) := \forall i \in \mathbb{Z}^{>0}. \ (100 \geq a_{i+1} > a_i) \]

We show that \( R \) is well-founded according to this definition:
Define \( \succ \) as follows: \((a_1, b_1) \succ (a_2, b_2) \iff a_1 > a_2\).

1. \( \succ \) is clearly antisymmetric and transitive.
2. Suppose a sequence \(((a_1, b_1), (a_2, b_2), \ldots)\) satisfies \(\mathcal{R}\). Then for \(i > 0\) we have \(a_{i+1} > a_i\) and thus \((a_{i+1}, b_{i+1}) \succ (a_i, b_i)\).
3. It is easy to see that for any starting element \(a_1\), every sequence must have length at most \(\max(0, 101 - a_1)\).
4. The transitivity requirement is satisfied because the \(\succ\) order is transitive and the union of two sets of terms, each bounded above by 100, is also bounded above by 100.

This proof generalizes to prove a more general theorem that will be useful in our proof of correctness:

**Definition 7.** If \(D_0\) is a domain of values with order \(\succ\), then:

- For any integers \(n, j\) with \(n \geq j \geq 1\), we can lift \(D_0\) to the domain \(D\) of \(n\)-tuples where \(D_0\) is the domain of the \(j\)th element of each tuple, i.e.
  \[
  D = \{(a_1, a_2, \ldots, a_n) \mid a_j \in D_0\}
  \]

- We can lift the order \(\succ\) on \(D_0\) to an order \(\succ_j\) on the lifted domain \(D\) that compares two tuples based only on their \(j\)th component, i.e.
  \[
  (a_1, a_2, \ldots, a_n) \succ_j (b_1, b_2, \ldots, b_n) \iff a_j > b_j
  \]

- We extend a lifted order \(\succ_j\) to a relation on \(D_0 \times D\) that compares an element \(a \in D_0\) with the \(j\)th component of an element \((a_1, \ldots, a_n) \in D\), i.e.
  \[
  a \succ_j (a_1, \ldots, a_n) \iff a > a_j
  \]

**Theorem 1.** Let the following be given:

- \(D_0\) a (possibly infinite) domain of discrete values with strict total order \(\succ\) and corresponding reflexive total order \(\succeq\) (here by discrete we mean that there are finitely many values between any two fixed values in \(D_0\); we require this so that part (3) in the proof above also generalizes here)
- \(n\) a positive integer
- \(j\) a positive integer with \(n \geq j \geq 1\)
- \(b \in D_0\)

Lift \(D_0\) to the domain \(D\) of \(n\)-tuples where \(D_0\) is the domain of the \(j\)th element of each tuple, and let \(\mathcal{R}\) be the property over \(D\) defined by

\[
\mathcal{R}((d_1, d_2, \ldots)) = \forall i. \ b \succeq_j d_{i+1} \succ d_i
\]

Then \(\mathcal{R}\) is well-founded.
We say that \(D_0, n, j, \) and \(b\) induce \(\mathcal{R}\).
Note that the set of integers is a valid choice for \(D_0\) (with the standard order), as is the set of lists of integers (with the structural order, in which one list is greater than another if and only if it has a greater length). The set of rational numbers, however, is not a valid choice: a sequence of strictly increasing rational numbers starting at 0 and bounded above by 100 can have unbounded length.

**Corollary 1.** Suppose \(D_0, n, j, \text{ and } b\) induce a well-founded property \(R\). If \(p\) is a predicate with \(n\) arguments, let \(R_p\) be a property over \(p\)-facts defined by

\[
R_p((p(t_1), p(t_2), \ldots)) := R((t_1, t_2, \ldots))
\]

Then \(R_p\) is well-founded.

### 4.4 Algorithm’s Relation to Well-Founded Properties

To understand the correctness of our algorithm, it is necessary to establish its relation to well-founded properties. Recall that argument mappings are related to certain sequences of rule applications. The correctness of the algorithm is related to the connection between these rule applications and derivations of facts in the evaluation of the program. In our proof of correctness, we reason about paths in derivation trees and their relations to well-founded properties. To do this, we first define some special kinds of derivation path segments that will be useful. For all of these, let \(p \in S\) be given for some program stratum \(S\).

**Definition 8.** A path segment \(F\) is a \(p\)-segment if it starts and ends at a \(p\)-fact.

**Definition 9.** A \(p\)-segment \(F\) is a simple \(p\)-segment if it contains only predicate symbols in \(S \setminus \{p\}\) in-between its first and last facts.

We use these definitions and facts about our algorithm to prove a result that is important to the algorithm’s correctness:

**Lemma 1.** Let \(R\) be a well-founded property over \(p\)-facts. If every simple \(p\)-segment satisfies \(R\), then every \(p\)-segment satisfies \(R\).

**Proof.** Suppose every simple \(p\)-segment satisfies \(R\), and let any \(p\)-segment \(F\) be given. We proceed by induction on the number \(N_p(F)\) of \(p\)-facts in \(F\).

Base case: \(N_p(F) = 2\)

Then \(F\) is a simple \(p\)-segment, so we are given that \(F\) satisfies \(R\).

Inductive step: Suppose every \(p\)-segment \(F'\) with \(N_p(F') = k - 1\) satisfies \(R\), and suppose \(N_p(F) = k\)

Let \(F_0\) be the subsegment of \(F\) that starts at the beginning of \(F\) and ends at the second \(p\)-fact in \(F\). Then \(F_0\) is a simple \(p\)-segment, so \(F_0\) satisfies \(R\). Let \(F_1\) be the subsegment of \(F\) that starts at the second \(p\)-fact and continues until the end of \(F\). Then \(N_p(F_1) = k - 1\), so \(F_1\) satisfies \(R\).

Since the end of \(F_0\) is the beginning of \(F_1\), each satisfies \(R\), and well-founded properties are transitive by definition, their concatenation \(F\) must also satisfy \(R\).
Theorem 2. Let $P$ be a program, $S$ be a stratum of that program, and $p$ be a predicate in $S$. If our algorithm verifies $p$, then there is some well-founded property $\mathcal{R}$ over $p$-facts such that every $p$-segment $F$ satisfies $\mathcal{R}$.

Proof. Suppose our algorithm verifies $p$, then there is some argument position $i$ such that every argument mapping of $p$ contains a $p_i$-cycle. Without loss of generality we assume that these cycles, as well as all the $p_i$-secondary cycles, are increasing and bounded above. There are finitely many such cycles, so we can take the maximum over the upper bounds to get a uniform upper bound $b$. Then there is a well-founded property $\mathcal{R}$ over $p$-facts induced by the type of value in $p_i$, the arity of $p$, the index $i$, and the bound $b$. We will show that $F$ satisfies $\mathcal{R}$. By our lemma, we may assume without loss of generality that $F$ is a simple $p$-segment. Thus, if $F = (p(\overline{t_1}), \ldots, p(\overline{t_2}))$, then it suffices to show that $(\overline{t_1})_i < (\overline{t_2})_i \leq b$.

A formal proof of this is in Krishnamurthy et al. [1996] and uses a structure called a transformation tree to relate the primary and secondary cycles to the path segment $F$. Informally, $F$ can be constructed from the argument mapping containing some primary $p_i$-cycle by recursively expanding predicates of that argument mapping with other cyclic argument mappings, each of which contains some $p_i$-secondary cycle. Since each of these cycles is independently increasing and bounded above, so is their finite concatenation $F$.

4.5 Proof of Algorithm’s Correctness

For the sake of completeness and clarity, we believe it is useful to reformulate the proof of correctness of the algorithm. We follow Krishnamurthy et al. in using EXTENDED DATALOG programs in our proof, though our actual implementation is for verifying programs of a different language (FORMULOG). In describing our implementation (§5) we discuss how we deal with differences in the two languages’ features, but ultimately for every FORMULOG program there is a semantically equivalent EXTENDED DATALOG program. It therefore suffices to work with EXTENDED DATALOG for the purposes of proving the algorithm’s correctness, so this is how we will proceed here.

Lemma 2. Let $P$ be an EXTENDED DATALOG program, and let $D$ be a valid EDB for $P$. If there exists an upper bound $c$ on the height of derivation trees for this program and EDB, then $P$ is safe.

Proof. The well-formedness and well-modedness of $P$ ensures that each rule application produces only finitely many facts (see Krishnamurthy et al. Lemma 3.1 for a proof of this). Since the height of a derivation for a fact corresponds to the number of rule applications in the derivation of that fact, $c$ puts an upper bound on the number of rule applications required to derive any fact. Thus there is also an upper bound on the number of facts that can be derived, so $P$ is safe.
**Theorem 3.** Suppose our algorithm verifies a stratum $S$ in Extended Datalog program $P$. Then there is an upper bound on the length of paths in derivation trees for facts with a predicate symbol in $S$. (This upper bound depends only on $S$ and $P$.)

**Proof.** If $F$ is a path in the derivation of a $p$-fact for some $p \in S$, let $N(F)$ be the number of (distinct) predicate symbols in $S$ along $F$. Then we have $|S| \geq N(F) \geq 1$, so it suffices to show that for every $k > 0$ there is a constant $c_k$ such that for all $F$ with $N(F) = k$ we have $c_k \geq |F|$. Then $\max\{c_k \mid 1 \leq k \leq |S|\}$ is an upper bound on the length of paths in general. We will prove this by induction on $k$.

We start by proving the base case: there is an upper bound on paths with only one recursive predicate symbol.

The well-formedness and well-modedness of $P$ ensures that there are only finitely many facts that can be derived by non-recursive predicates. There is therefore an upper bound $b_0$ on the length of any path containing no recursive predicates.

If a path contains a recursive predicate only as its last fact, then that fact must have been derived using a non-recursive rule. Thus, similarly to the above argument, there are only finitely many tuples $\bar{t}$ such that there exists a path containing non-recursive predicates until a final fact $p(\bar{t})$. We call such tuples valid first tuples of $p$, and their corresponding facts valid first $p$-facts.

Since our algorithm verifies $S$, there is at least one well-founded property $R_p$ for each predicate $p \in S$ such that every argument mapping of $p$ respects $R_p$. Then every path segment consisting of only $p$-facts satisfies $R_p$. Then for each $p \in S$ and valid first tuple $\bar{t}$ of $p$ there is an upper bound $b_{p,\bar{t}}$ on the length of path segments starting at $p(\bar{t})$. Then $b_p := \max\{b_{p,\bar{t}} \mid \bar{t} \text{ is a valid first tuple of } p\}$ is an upper bound on the length of path segments starting at a valid first $p$-fact, and $b_1 := \max\{b_p \mid p \in S\}$ is an upper bound on the length of path segments starting at a valid first tuple of any recursive predicate in $S$. We will show that $c_1 := b_0 + b_1 + 1$ is an upper bound on the length of paths with only one recursive predicate symbol. Let $F$ be such a path, and let $p$ be the recursive predicate it contains.

$F$ has (at most) two components:

- a prefix $F_0$ that starts at the beginning of $F$ and ends just before the first recursive predicate instance
- a segment $F_1$ that starts at the first recursive predicate instance and ends at the end of $F$

We have $|F| = F_0 + 1 + F_1$. $F_0$ contains no recursive predicates and thus has length at most $b_0$. $F_1$ must start with a valid first $p$-fact. Since the other predicates in $F$ are not defined in terms of $p$, $F_1$ must contain only $p$-facts. Thus $F_1$ has length at most $b_1$, so we have our desired result $|F| \leq b_0 + b_1 + 1 = c_1$. 

18
Now suppose that there is an upper bound on paths with fewer than \( k \) recursive predicate symbols. We will show that there is an upper bound on paths with exactly \( k \) recursive predicate symbols:

Let \( c_{k-1} \) be the upper bound on paths with \( k-1 \) recursive predicate symbols.

Since our program is well-formed and well-moded, there are finitely many facts that can be produced by \( c_{k-1} + 1 \) rule applications. For any \( p \in S \), let \( V_p \) be the set of all such \( p \)-facts. Since our algorithm verified \( S \), for each \( p \in S \) and \( l \in V_p \) there is an upper bound \( b_{p,l} \) on the number of \( p \)-facts on \( p \)-segments starting with \( p(l) \). For each \( p \in S \) let \( b_p = \max\{b_{p,l} \mid l \in V_p\} \), and let \( b = \max\{b_p \mid p \in S\} \). Let \( N \) be the number of predicate symbols in \( P \), then we will show that \( c_k := c_{k-1} + bN + 1 \) is an upper bound on the length of paths with exactly \( k \) predicate symbols.

Let \( F \) be any path with exactly \( k \) predicate symbols, and let \( p \) be the recursive predicate symbol that appears the greatest amount of times on \( F \). Let \( F_0 \) be the prefix that starts at the beginning of \( F \) and ends just before the first \( p \)-fact, and let \( F_1 \) be the segment that is the rest of the path. \( F_0 \) does not contain any \( p \)-facts, so \( N(F_0) < k \) and thus \( |F_0| \leq c_{k-1} \).

For any \( q \in S \) let \( L_{F_1}(q) \) denote the number of \( q \)-facts that occur on \( F_1 \), and note that \( F_1 \) contains only recursive predicates in \( S \). Then we have

\[
|F_1| = \sum_{q \in S} L_{F_1}(q) \leq |S| L_{F_1}(p) \leq N L_{F_1}(p)
\]

Now if \( p(l) \), then we have \( L_{F_1}(p) \leq b_{p,l} \leq b \). Putting these together, we see that \( |F_1| \leq bN \).

Thus we have \( |F| = |F_0| + 1 + |F_1| \leq c_{k-1} + bN + 1 = c_k \).

\[\square\]

**Corollary 2.** If our algorithm verifies an Extended Datalog program \( P \), then \( P \) is safe.

**Proof.** \( P \) has a stratification \( S \) of finitely many strata, and the above theorem guarantees that there is a bound \( c_S \) on the length of paths in derivation trees of facts with a predicate symbol in \( S \) for each stratum \( S \in S \). Thus \( c := \max\{c_S \mid S \in S\} \) is a bound on the length of paths in derivation trees for \( P \). The height of a derivation tree is just the length of its longest path, \( c \) is also an upper bound on the height of derivation trees for \( P \). Thus by the lemma above, \( P \) is safe.

\[\square\]

### 5 Algorithm Implementation for FormuLog

We now turn our attention from the theoretical presentation of the algorithm to our implementation of it. We describe how we adapted the algorithm for our setting of FORMULOG, and how we implemented the equality and monotonicity inferences that allow us to add edges and arcs to our argument mappings.
5.1 Adapting Algorithm for FormuLog

The original algorithm runs on an extension of Datalog that is distinct from FormuLog. Consequently, the algorithm had to be adapted for FormuLog—in part to make the implementation line up with the specification of the original algorithm (which was for Extended Datalog), and in part to simplify that implementation.

5.1.1 Constructor Symbols

Unlike Extended Datalog, FormuLog does allow complex terms with n-ary constructor symbols. However, as we noted in §2.1, we can represent such terms in Extended Datalog by leveraging the fact that we can have infinitely many base facts in Extended Datalog. For example, consider the :: ("cons") 2-ary constructor symbol and [] ("nil") constant built into FormuLog to represent lists, e.g. 5 :: 3 :: [] represents the list [5, 3]. In Extended Datalog we can model this with a 3-ary EDB predicate cons such that cons(X, Y, Z) holds if and only if Z = X :: Y. So, in our example, we could write cons(3, nil, Y), cons(5, Y, Z) in Extended Datalog to model Z = 5 :: 3 :: [] in FormuLog. This is why we are able to associate a semantically equivalent Extended Datalog program to every FormuLog program, justifying the connection between the algorithm’s proof of correctness (which assumes that it operates on Extended Datalog programs) and the algorithm’s implementation (which operates on FormuLog programs).

5.1.2 NiceProgram Transformation

In our implementation, we preprocess the input Program into a NiceProgram which is in canonical form. As described in §4.1.2, this means that it has the following convenient properties for each rule:

- each argument to every normal predicate is a variable
- the first argument to every unification is a variable

We use Rule, NiceRule, Program, and NiceProgram to refer to classes in our implementation.

Any Rule can be turned into an equivalent NiceRule:

1. for the head atom and each body atom in the Rule:
   (a) if the atom is a normal predicate instance:
      i. for each term t in the instance that is not already a variable:
         A. generate a fresh variable X
         B. replace t with X
         C. add a unification X = t to the body of the Rule
   (b) if the atom is a unification predicate and the first term t is not a variable:
      i. generate a fresh variable X
      ii. replace t with X
      iii. add a unification X = t to the body of the Rule
5.1.3 Functions

One of the ways in which FORMULOG extends DATALOG is by providing support for ML-style function definitions [Bembenek and Chong, 2018]. A function call is a valid term in a program rule, and is evaluated by evaluating the function on the given inputs and using the resulting output as the term. Our current implementation handles this in the most conservative way: it does not reason about any code inside a function definition and therefore does not infer anything about a function call’s inputs, outputs, or how they relate to each other. Below is an example of a safe program that we cannot currently verify due to this conservative implementation:

\[
\text{fun succ}(X : \text{i32}) : \text{i32} = X + 1.\\
\text{p}(X) :- \text{b}(X).\\
\text{p}(\text{succ}(X)) :- \text{p}(X), \text{b}(X).
\]

Here \( \text{b} \) is a finite EDB predicate. The fact that the argument to \( \text{p} \) increases with each rule application is hidden behind the \( \text{succ} \) function.

A less conservative solution, and possible extension of the current implementation, would be to translate function definitions into recursive rule definitions. The fact that FORMULOG functions are not higher-order (i.e. their inputs and outputs cannot themselves be functions) makes such a translation relatively straightforward. This would allow us to use the same reasoning that is already implemented to infer facts about the inputs and outputs of a function call in the rules we are analyzing. The more such facts the algorithm can infer, the more safe programs it can verify.

5.2 Implemented Inference

Each argument mapping needs to be populated with edges based on the rules used to construct that particular argument mapping. That is, the algorithm needs to infer from the rules which variables are equal to, greater than, or less than each other, which are bounded below by a constant, and which are bounded above by a constant. Krishnamurthy et al. [1996] leaves the method of doing this as an implementation detail, so we describe our general process for doing so followed by an account of the specific reasoning that is implemented.

5.2.1 High-Level Overview

Inference for an argument mapping occurs as it is built up, one rule at a time. Since each edge and arc connects adjacent argument mapping components, it suffices to add the edges at the time that each rule is used to build up the argument mapping. The method that does so iterates through the given rule’s body atoms several times. Since the argument mapping represents a series of rule applications, and the application of a rule requires that its body be satisfied, the inference may assume that each atom in the body is satisfied and thereby infer facts about the values of variables in the rule. It iterates through the body atoms, collecting facts from each atom into sets of bounded-above variables, bounded-below variables, and pairs of variables representing
the fact that one variable is greater than another in value. If any such inference information is found, this step is repeated because this information may allow the method to identify even more information. The transitive closure of the monotonicity information is taken, and these are then used to add directed arcs to the argument mapping. Nodes corresponding to the same variables are also connected by undirected edges signifying equality.

5.2.2 Identifying Facts from Rule Body Atoms

We list below the reasoning that was implemented in the algorithm for extracting boundedness and ordering facts from the atoms in a rule body. We include this not because any of the reasoning is particularly complex, but rather because its relative simplicity supports the power and potential practicality of the algorithm as a whole. If this relatively simple reasoning has reasonable results, then it suggests that very complex, and perhaps not tractable, reasoning may not be required to achieve a practical implementation of the algorithm. The preprocessing of the program into a NiceProgram simplifies the case-work involved here.

- If \( b \) is a finite EDB predicate, or being treated as one because it is in a lower stratum than the one we are considering for this argument mapping, then the values of arguments to \( b \) are bounded above and below. That is, if \( b(X) \) holds then \( X \) is one of finitely many values, so there is an upper and lower constant bound on it.
- If \( X \) is unified with a constructor, its value has a natural structural measure which is non-negative, so \( X \) is bounded below.
- If \( X \) is unified with a constructor that contains a subterm \( Y \) of the same constructor type, then \( X \) is structurally more complex than \( Y \) and therefore we have \( X > Y \) using the structural order.
- If \( X \) is unified with the algebraic expression \( Y + k \) for some positive integer \( k \), then we have \( X > Y \). We implement similar inferencing for \( k < 0 \) and for \( Y - k \).
- If it is true that \( X \geq k \) for some integer \( k \), then \( X \) is bounded below. We implement similar inferencing for \( X \leq k \) and strict inequality.
- If it is true that \( X \geq Y \) and we have already inferred that \( Y \) is bounded below, then \( X \) is also bounded below (and similarly for other inequality relations).

6 Results on Test Programs

We ran the algorithm on several FORMULOG programs of varying complexity and recorded which programs our algorithm verified. When the algorithm failed to verify a safe program, we investigated which predicate it failed to verify, and which argument mapping it was unable to find a primary or secondary cycle in. From this we developed a sense of the algorithm’s practicality. It successfully validates many safe simple programs, suggesting that the general idea of the algorithm is promising with respect to its being practical. It also fails to verify many more complex programs (and some simpler
programs), suggesting that our implementation is not currently sufficiently practical. The results do, however, point us towards potential extensions of the algorithm that would allow it to verify a larger class of safe programs.

6.1 Basic Toy Programs

Our algorithm does well on basic programs constructed specifically for evaluating the algorithm. A useful set of examples is taken from Krishnamurthy et al. [1996] Consider the following program rules, where $b$, $c$, and $d$ are finite EDB predicates:

\[
\begin{align*}
    p(X, Y) & : - p(U, V), X = U - 1, Y = V + 1, b(X). \\
    q(X, Y) & : - q(U, V), X = U - 1, Y = V + 1, b(X). \\
    p(X, Y) & : - c(X, Y). \\
    q(X, Y) & : - d(X, Y).
\end{align*}
\]

There are two argument mappings that the algorithm needs to consider here, one each for the predicates $p$ and $q$. Both argument mappings contain just one predicate (and therefore two components) and four nodes each. It correctly infers from the atom $X = U - 1$ that the first argument decreases with each application of the first rule (which corresponds to the argument mapping of $p$) and of the second rule (which corresponds to the argument mapping of $q$). It also infers from the atom $b(X)$ that the first argument is bounded above and below in both argument mappings. From this information, the algorithm correctly identifies that the first argument in both $p$ and $q$ is bounded below and decreases with every recursive rule application, and that the program is therefore safe.

Now suppose we add an additional rule $p(X, Y) : - q(Y, X)$. The predicates $p$ and $q$ are still not mutually recursive. Once it establishes that $q$ is safe by the same reasoning as above, the algorithm treats it as a finite EDB predicate and is able to apply the same reasoning as above to prove that $p$ (and therefore the program overall) is safe.

In addition to the rule above, suppose we add another rule $q(X, U) : - p(U, V), b(X)$. Now $p$ and $q$ are in the same stratum and the algorithm finds new argument mappings to consider. In those argument mappings, the algorithm is not able to find cycles that prove the safety of $p$ and $q$. It therefore marks this program as unsafe. Indeed, this program generates infinitely many facts. To see this, suppose for example that the only base facts are $b(0), b(1), c(1, 1),$ and $d(1, 1)$. Then we immediately derive $q(1, 1)$. From the second rule we derive $q(0, 2)$. From the fifth rule we derive $p(2, 0)$, and from the sixth rule we derive $q(1, 2)$. Repeating this process, we derive $q(0, 3), p(3, 0),$ and $q(1, 3)$. We can repeat this any number of times, generating every positive integer in the second argument position of $q$.

6.2 Simple Real Programs

Next we consider the algorithm’s performance on two representative programs that, while they are also fairly simple, have relatively interesting semantics and may reasonably arise in real-world programs.

Consider the membership relation, defined below:
As is, the `member` relation is not safe. For example `member(nil, Xs)` holds for every list, leading to infinitely many facts. However, if we have a query that fixes the second argument, e.g. `:- member(X, [1, 2, 3])`, then there are finitely many answers to that query. We can perform a magic set transformation to get an equivalent program that can be evaluated bottom-up, where supplementary predicates enforce the fact that the second argument to `member` is fixed (and therefore bounded)[Ramakrishnan, 1991]. The details of magic set transformations are not relevant here, but it suffices to know that it results in an equivalent program and that it is one of the DATALOG optimizations we would like to know when it is safe to apply. Once we do so, our algorithm is able to prove that the transformed program is safe by identifying that `member` facts are derived by starting with a second argument of fixed length and decreasing it by one at each application of a rule with head predicate `p`.

Similarly, consider the `path` relation, which represents whether there is a path (in a fixed graph) from one node to another of a given length:

\[
\begin{align*}
\text{path}(X, X, 0). \\
\text{path}(X, Y, N) & :- M = N - 1, M >= 0, \text{path}(X, Z, M), \text{edge}(Z, Y).
\end{align*}
\]

Here `edge` is an EDB predicate. If we use `path("a", Y, 5)` as our query (where "a" is a node in our graph), our algorithm is able to correctly identify the transformed program as being safe. Fixing the first argument to `path` in our query prevents the first rule from breaking safety by giving us a concrete, finite starting point. Fixing the third argument gives an endpoint to an increasing, bounded-above cycle: every application of the second rule decreases the third argument by 1, the first rule starts the third argument value at 0, and the query bounds it at 5.

### 6.3 Static Analysis Programs

When run on thirteen separate programs used for code analyses, our algorithm had overall negative, though promising, results. FORMULOGEN was developed to facilitate writing certain classes of static analyses, and these programs were developed by the FORMULOGEN team to evaluate the expressiveness and practicality of FORMULOGEN [Bembenek and Chong, 2018]. They operate over toy languages and calculi, but some of them are nonetheless relatively complex programs. We therefore use them to evaluate the success of our algorithm. We give a brief description of the two programs which it successfully identifies as safe, followed by a discussion of why it is unable to verify some of the other programs. We believe that all of these analyses are indeed safe, though this is not known for certain. Below we provide a table listing each program’s size, whether or not the algorithm verified it, and the subsection corresponding to the explanation of the algorithm’s results on this program.
<table>
<thead>
<tr>
<th>#</th>
<th># of Predicates</th>
<th># of Rules</th>
<th>Max Stratum Size</th>
<th>Verified</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>No</td>
<td>§6.3.4</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>300</td>
<td>109</td>
<td>No</td>
<td>§6.3.3</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>55</td>
<td>10</td>
<td>Yes</td>
<td>§6.3.1</td>
</tr>
<tr>
<td>4</td>
<td>74</td>
<td>151</td>
<td>72</td>
<td>No</td>
<td>§6.3.3</td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>122</td>
<td>54</td>
<td>No</td>
<td>§6.3.5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>No</td>
<td>§6.3.4</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>No</td>
<td>§6.3.2</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>55</td>
<td>22</td>
<td>No</td>
<td>§6.3.4</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>35</td>
<td>14</td>
<td>No</td>
<td>§6.3.5</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>20</td>
<td>7</td>
<td>No</td>
<td>§6.3.5</td>
</tr>
<tr>
<td>11</td>
<td>40</td>
<td>84</td>
<td>38</td>
<td>No</td>
<td>§6.3.4</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>No</td>
<td>§6.3.4</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>Yes</td>
<td>§6.3.1</td>
</tr>
</tbody>
</table>

### 6.3.1 Successful Verifications

Two of the thirteen programs are proven safe by the algorithm. One is a FORMuLOG implementation of symbolic execution (Program #13), where a simple imperative program is interpreted with some values being symbolic, i.e. they represent sets of possible runtime values rather than a single concrete value. It is roughly 100 lines long and contains 4 function definitions and 5 rules (before transformation). The insight that is at the heart of why the program is safe, and why the algorithm is able to validate it, is that the recursive predicate `reach` has as its third argument position a value representing the “fuel” of the analysis: the value starts with some positive initial integer value, is decreased as the analysis proceeds, and is required to be positive for the analysis to continue. There are other recursive predicates in the program, but their safety follows naturally once we verify `reach`.

The other success is a FORMuLOG implementation of goal-oriented symbolic execution (Program #3), which aims to optimize symbolic execution by exploring only those code paths in the target program that are relevant to the query of interest. Our algorithm is run on a magic set transformation of this program. It is roughly 200 lines and contains 15 functions and 10 rules (before transformation). This program is verifiably safe for a reason similar to the previous program: the recursive predicate `reach` in this program has a “counter” as its third argument that is increased as the analysis proceeds and required to be less than a fixed upper bound. The verifiable safety of the other recursive predicates follows from this predicate’s safety in this program as well.

### 6.3.2 Unsuccessful, Difficult Verification

For one case, our algorithm is unable to verify a program in a way that we believe does not detract significantly from its overall practicality. This program is an implementation of a reaching-definitions analysis, in which we determine which variable definitions reach each point in a target program. In this implementation, the algorithm is not able to prove the safety of the program, though we do believe the program is safe. For the predicate on which the algorithm gets stuck, however, the guarantee of safety is not
straight-forward. The program keeps track of a set of facts, implemented as a simple list with the maintained invariants that the list is sorted and does not contain any duplicates. Thus any set has a unique canonical representation in this implementation. As the program is evaluated, the values corresponding to this set may grow in size as a result of set unions (implemented as a user-defined function). It can be proven that the set of possible facts is finite, due to the fact that the program is finite and therefore so is the number of definitions it contains. Thus, the set of facts will eventually reach a fixed-point. This claim is necessary to show the safety of the program, and not obvious from the program itself. It therefore would require quite non-trivial reasoning on the part of any person or algorithm to derive from the program its safety. The algorithm is not able to reason abstractly about what the set of facts represents about the target program of the analysis, and so it is not able to verify the analysis. Consequently, we do not consider this instance of the algorithm’s failure to verify a safe program as counting against its practicality. As we mention in §7, one possibility to handle programs like this is to allow the user to annotate the program with equality and monotonicity information and hints for verifying this information. Our implementation could then be extended to add more edges and arcs to its argument mappings (after verifying the annotated information), which would allow it to verify more safe programs like this one.

6.3.3 Algorithm Did Not Finish

For two of the programs, the algorithm took too long and did not finish its analysis. The runtime of the algorithm is linear in the number of program strata, but verifying a stratum takes time exponential in the size of that stratum (i.e. the number of predicates in the stratum). This is generally not an issue even for real-world programs, except for the fact that the algorithm is often run on the magic set transformation of programs. These transformed programs often have more predicates and, more importantly, larger strata than the original programs. The transformed programs had stratum sizes of 109 and 72 predicates, respectively. Krishnamurthy et al. [1996] addresses this by presenting an extension of the algorithm that takes advantage of the details of magic set transformation. We have not implemented this extension, though we do expect it to work in our setting.

6.3.4 Lack of Monotonic and Bounded Cycles

In five of the programs, the algorithm does not verify a predicate because there is an argument mapping of that predicate which does not contain any strictly increasing, bounded-above (or decreasing, bounded-below) cycles. We believe this is a great opportunity for extending the algorithm, as just a few additional sound modes of reasoning should allow significant improvement in results.

We consider an example taken from part of a larger program and simplified here:

\[
\begin{align*}
\text{p}(X, Y) & \leftarrow \text{b}(X), \text{b}(Y). \\
\text{p}(X, Y) & \leftarrow \text{p}(Z, Y), \text{c}(X).
\end{align*}
\]
Here $b$ and $c$ are finite EDB predicate. First we should establish that this program is indeed safe. The first rule is not recursive and poses no challenges to safety. The second rule only allows us to derive a $p$-fact where the first argument is bounded and the second argument is in a previously-derived fact. One way to see more formally that this program is safe is to consider the set of facts $p(X, Y)$ for all $X, Y$ such that $b(X)$ holds and either $b(Y)$ or $c(Y)$ holds. This set is finite because $b$ and $c$ are finite by assumption. The set clearly satisfies the first rule in the example (recall the definitions of safety and satisfying from §2.2). The set also satisfies the second rule: suppose $p(Z, Y)$ and $b(X)$ holds, then we must also have that either $b(Y)$ or $c(Y)$ holds, so $p(X, Y)$ also holds. Thus the program is safe and, using our bottom-up evaluation strategy, will result in the derivation of finitely many facts.

Our algorithm cannot currently verify this program, however. In particular, the argument mapping corresponding to the second rule has no directed arcs, precluding the possibility of a monotonic cycle. This suggest a possible extension to the algorithm.

Let $\pi_i(d)$ be the $i$th projection of the tuple $d$ (e.g. $\pi_2((a, b)) = b$) and consider the following property over 2-tuples of integers:

$$R_{m,n}(d_1, d_2, \ldots) := \forall i \in \{1, 2\}. (\forall j, k > 0. \pi_i(d_j) = \pi_i(d_k)) \lor (\forall j > 1. m \leq \pi_i(d_j) \leq n)$$

That is, for some fixed $m, n$, a sequence satisfies $R_{m,n}$ iff each projection of the sequence onto an argument position is either constant or bounded below by $m$ and above by $n$ (though the first fact need not be bounded, i.e. we do not require that $m \leq \pi_i(d_1) \leq n$ in the second disjunct above).

This property is not, in general, well-founded:

**Proof.** Fix $m, n$ with $m < n$ and assume towards contradiction that $R_{m,n}$ is well-founded. Let $\succ$ be the corresponding order guaranteed by well-foundedness.

Set $d_1 = (0, m)$ and $d_2 = (0, n)$. Note that $d_1 \neq d_2$, $R_{m,n}(d_1, d_2) = R_{m,n}(d_2, d_1) = 1$. Therefore we must have both $d_2 \succ d_1$ and $d_1 \succ d_2$, contradicting the antisymmetry of $\succ$.

We believe, however, that the definition of well-foundedness can be further weakened such that $R_{m,n}$ is well-founded for a fixed $m$ and $n$. If so, the algorithm could be modified in a straight-forward way to accomodate the reasoning associated with this property. Specifically, in addition to looking for a monotonic and bounded cycle, it would disjointly look for an equality cycle (one with only undirected edges) through every argument position of the last $p$ argument mapping component not bounded on both sides. Such reasoning would be sufficient to show the safety of the predicate in this example and many others.

### 6.3.5 Secondary Cycles Not Found

Another common, yet simple, source of impracticality in the algorithm has to do with secondary cycles. Recall that once the algorithm finds a cycle in an argument mapping $M$ of predicate $p$ that is monotonic and bounded, it requires that every argument mapping of a predicate along $M$ (but not containing $p$) also have such a cycle through the corresponding node.
We believe that this requirement is stronger than necessary, as exemplified by the simplified version of another program fragment (with rule numbers added to the left for expositional clarity):

\[ R_1: \quad p(X) :- b(X). \]
\[ R_2: \quad q(X) :- b(X). \]
\[ R_3: \quad p(X + 1) :- q(X), X < 100. \]
\[ R_4: \quad q(X) :- p(X). \]
\[ R_5: \quad q(X) :- q(X). \]

Here \( b \) is a finite EDB predicate. It is a little less obvious (relative to our previous example) that this program is safe. The first two rules are clearly not sources of danger. The only place we see a new term is in \( R_3 \), from which we can conclude that all new terms are greater than or equal to terms found in the (finitely many, by assumption) EDB facts. \( R_3 \) also ensures that these new terms are all less than 100. There are therefore finitely many terms in the facts generated by this program, which means there can only be finitely many facts generated, i.e., the program is safe.

Note that \( p \) and \( q \) are in the same stratum. For the purposes of this example, we suppose that the algorithm first checks the argument mapping corresponding to the rule sequence \( p \leftarrow R_3 \) \( q \leftarrow R_1 \) \( p \). Here it finds an increasing cycle, bounded above by 100. The algorithm then looks for a secondary cycle in the argument mapping corresponding to \( q \leftarrow R_5 \) \( q \). This argument mapping contains only undirected edges and therefore no increasing cycles.

We believe that the sufficiency condition for a secondary cycle respecting a primary cycle can be weakened. Specifically, a strictly increasing and bounded-above primary cycle should be treated as being respected by a secondary cycle so long as that secondary cycle is non-decreasing and bounded above. We are not sure if the boundedness constraint on the secondary cycles can, in full generality, be relaxed as well.

### 7 Related Work

Other papers on the termination of logic programs can be categorized on a number of dimensions. Some papers aim at theoretical work and examining interesting subsets of safe programs, while others have more practical and algorithmic motivations such as runtime efficiency. Some discuss general algorithms that are novel in their theoretical underpinnings or their implementation, while other works focus on specific extensions or special cases of previous algorithms. While many of the works discussed here use a similar kind of algorithm as us, a few use a different approach involving constraint satisfiability. Among all these works we see termination-checking applied to several different logic programming languages and models of evaluation. De Schreye and Decorte [1994] provide a good survey of many of the works and where they stand in relation to each other. Here we discuss two papers that take a different approach to checking termination, then several papers that present solutions similar to the algorithm by Krishnamurthy et al. [1996], and finally a related paper that points to another potential extension of our implementation.
Ullman and Van Gelder [1988] consider the PROLOG language, another classic example of logic programming that allows for \(n\)-ary constructors, consequently making it Turing-complete. They use a top-down model of computation, in which computation starts at the query and substitutes predicates in the head of a rule with predicates in the body of the rule until the derivation tree can be grounded by base facts. Their paper considers programs where all values are lists (constructors), and the only measure used in their reasoning is the structural measure on these lists. They use a graph-based approach (different from the one considered in our paper) to reason about values within a rule to identify relations between them. The output is ultimately a set of inequalities. For a certain class of PROLOG programs, the satisfiability of these inequalities implies the safety of the program. Ullman et al. are also more motivated by optimizing for the runtime of their algorithm.

Decorte et al. [1999] is similar in approach to Ullman et al. in reducing program safety (in a top-down setting) to constraint satisfiability. They correctly point out a limitation of the approach taken by us and Krishnamurthy et al.: the more reasoning we add to the implementation, the more complex and time-consuming our algorithm becomes. Like Ullman et al., their algorithm aims for efficiency.

Dershowitz et al. [2001] present an algorithm similar to ours but for a top-down evaluation model. They begin with an algorithm that only reasons about the structural measure of values and extend it to also use integer measure. (Our algorithm uses both for its inference as well.)

Kifer et al. [1988] provide what can be seen as a follow-up to the paper by Krishnamurthy et al. They also consider the setting of EXTENDED DATALOG. The paper identifies a special case of the safety condition which they call supersafety. Unlike the question of safety, supersafety of EXTENDED DATALOG is a decidable problem for which a decision algorithm is given. That means that their algorithm can verify a subset of safe programs, namely those that are supersafe. If we consider programs where the predicates only take a single argument, we believe all such programs that our algorithm can verify are supersafe, so that their safety is decidable by the reasoning described by Kifer et al. However, the following program is not supersafe, despite being easily verified by our algorithm:

\[
\begin{align*}
  p(X, Y) & :- b(X), b(Y). \\
  p(X + 1, Y) & :- p(X, Y), b(X).
\end{align*}
\]

Neither our algorithm nor Kifer et al.’s can currently verify the following simple program, though one of the extensions we describe above would allow us to verify it:

\[
\begin{align*}
  p(X) & :- b(X). \\
  p(X) & :- p(X).
\end{align*}
\]

A similar paper by Ramakrishnan et al. [1987] identifies a subset of programs for which safety is decidable and describes a decision algorithm for that subset. They then extend this algorithm to verify some programs not in this subset (this extension is not a decision algorithm, i.e. it may fail to verify a safe program). The extension is just a new presentation of the same algorithm presented by Krishnamurthy et al., except that it considers a top-down evaluation model.
Lindenstrauss et al. [2004]’s work is probably the most similar to our own. They have an argument-mapping-based implementation (which they call TermiLog) of a very similar algorithm for Prolog. Their presentation, however, includes a different theoretical underpinning which involves Ramsey’s Theorem and partial mappings from terms in the program to a finite ordered set. One feature of TermiLog is that it allows the user to specify other orders to use for identifying monotonicity (corresponding to arcs in the argument mappings). This would be an interesting extension of our implementation. Neither TermiLog nor our own implementation can handle more complex relations between terms, e.g. a predicate in which the sum of the first two arguments is always equal to the third argument. Allowing the user to specify such relations (and verify them automatically) might be another interesting extension. It is unclear how some of the suggestions we made for extending our implementation would work based on TermiLog’s theoretical backing, for example looking for argument mappings with many equality cycles and bounded nodes instead of requiring a monotonic and bounded cycle.

Lastly, another noteworthy paper is by Brodsky and Sagiv [1999], who write about inferring monotonicity constraints. Our implementation infers that the value of one variable is greater than another either by a structural measure based on the type of the value or by knowing something about a built-in function or predicate (e.g. that $x + 1$ is greater than $x$). Brodsky and Sagiv describe how to infer monotonicity constraints guaranteed by user-defined (IDB) predicates from those guaranteed by EDB predicates. Such a process might increase the power of our implementation.

8 Concluding Remarks

In this paper we have described our implementation of the termination-checking algorithm presented by Krishnamurthy et al. [1996], which we adapted for the Datalog extension FormuLog [Bembenek and Chong, 2018]. The algorithm itself is parametric in the specific inference used to reason about an individual argument mapping, so long as that reasoning is consistent with the notion of well-foundedness necessary for the algorithm to be sound. The strength of the definition of well-foundedness therefore represents a balance between the algorithm’s correctness and its practicality. We showed that K-well-foundedness was too strong for this reason, and we established a weaker definition of well-foundedness that is applicable to the reasoning mentioned by Krishnamurthy et al. and implemented by us.

This refined formulation of the algorithm’s correctness will facilitate the process of extending it. By evaluating the algorithm’s performance on a number of programs of varying complexity, we developed a better understanding of its potential practicality and its current weaknesses. The results seem to indicate that extending the algorithm should be worthwhile in terms of added practicality. The two general areas of extension are in the detection of primary cycles (or similar properties of argument mappings) and in the validation of these properties by secondary cycles (or similar). We mentioned specific potential extensions in both of these areas. Some of them may require further weakening of our notion of well-foundedness, which in turn would require another reformulation of the algorithm’s proof of correctness. This iterative process connecting
the algorithm to its theoretical underpinnings promises both a better understanding of sufficient conditions for program safety and more practical tools for automatically proving those conditions in real-world logic programs.

References


