

A Framework of Directionality for Proving Termination of Logic Programs

Francois Bronsard

*T.K. Lakshman*¹

Uday S. Reddy

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, USA

internet: `<bronsard,lakshman,reddy>@cs.uiuc.edu`

Abstract

In this paper we propose a rich notion of directionality of predicates that combines modes and regular tree types. We provide a semantic soundness result for this notion and give inference systems to decide *well-modedness* of logic programs and goals. We show how this rich notion of directionality can be used to prove the *universal* termination of LD-resolution for logic programs with non-ground goals by using simple syntactic orderings of the kind used in rewriting theory.

Keywords: Directionality, Modes, Regular tree types, Mode-dependence, Well-moded programs, Universal termination, Well-founded orderings.

1 Introduction

Proof techniques for proving termination² of logic programs have recently emerged as a focus of attention [27, 3, 1, 4, 2, 17, 29, 18, 24, 25, 26, 28, 20]. A key concept in the study of termination of logic programs is the notion of the *directionality* of a predicate i.e., the designation of some arguments to the predicate as “input” and other arguments as “output”. Directionality is normally specified via notions of *modes* of predicates and *well-modedness* of programs and goals.

In general, the mode information only specifies which arguments are expected to be ground (these are the input arguments) and which arguments are expected to be non-ground (these are the output arguments) ([6, 21]). This ground/non-ground distinction, which is almost always used to express directionality, is however, too rigid for most programs. For example, the classical program for `append` terminates if the first (or the third) argument is a finite list, independently of whether the elements of this list are ground or not.

In this work, we generalize the original notion of ground/non-ground modes by combining modes and a subset of the regular tree types [14, 15] to define *mode-annotations*. Mode-annotations describe sets of possibly non-ground terms sharing a common structure. For example, the mode-annotation describing a set of arbitrary terms (including variables) is “?”;

¹presently at IBM Almaden Research Center (lakshman@almaden.ibm.com)

²“Termination” is taken to mean *universal termination* [27], i.e., every branch of the SLD-tree is finite.

lists of arbitrary terms have the mode-annotation “list(?)”. The essential difference between our approach and [15] is that mode-annotations describe sets of (possibly nonground) terms whereas types include only ground terms. Thus the syntax of mode-annotations includes the additional symbol, “?”, which represents arbitrary terms, including non-ground terms. Hence, these mode-annotations provide a formal basis for studying termination of non-ground goals. In Section 7 we discuss the possibility of using the full regular tree types of [22, 19] to give directionality to programs using partially instantiated structures such as difference lists.

Absence of directionality distinctions has been cited as one of the strengths of logic programming. Since non-directional predicate definitions can be *reused* with different directionalities, non-directionality is indeed an advantage in terms of *code size* and *conciseness*. However, properties such as complexity and termination do depend on the directionality of predicates and hence need to be verified afresh for each directionality. Furthermore, most predicate definitions can only be used with specific directionalities.

We thus view a predicate as a *directional* procedure that, when applied to a tuple of terms satisfying an *input mode*, generates a tuple of terms satisfying a more specific³ *output mode*. In Section 3 we propose a notion of *mode-dependency* for a predicate that defines such a relationship between the structure of inputs and outputs of the predicate. An assignment of mode-dependencies to predicates is then used to syntactically define the notion of *well-moded* clauses and goals. Well-modedness is justified by a semantic soundness result that says that when a well-moded goal is resolved with a well-moded program, if its inputs are restricted to satisfy the assigned mode-dependency, then after resolution the outputs also satisfy the assigned mode-dependency.

Though we assume the standard Prolog computation rule, the results extend to any other choice of a computation rule. Similarly, we assume that predicates are assigned a unique mode-dependency (or equivalently that each mode-dependency defines a different version of the predicate), although, again, our results extend to multiple mode-dependencies for predicates.

The second part of this paper deals with the use of well-founded orderings to prove termination. Typical proofs of termination of programs involve showing that each step of the computation represents a reduction according to a well-founded ordering. Formally, this is expressed for logic programs as follows:

Remark 1 A logic program P universally terminates if and only if there exists a well-founded ordering \succ , such that for all goals G_1, G_2 ,

$$G_1 \vdash_P {}^4 G_2 \text{ implies } G_1 \succ G_2$$

Two observations arise from this remark. First, for a program that contains recursive predicates it is not normally possible to show universal termination, unless there are some restrictions on goals. For example, given the usual definition of the predicate `append`, from the goal `append(X, Y, Z)` we can infer the new goal `append(X', Y, Z')`. Since this new goal is equivalent to the initial goal, up to renaming of variables, there cannot exist a well-founded

³“More specific” intuitively means that mode-annotations of the form “?” are replaced by mode-annotations such as “ground” or “list(?)”.

⁴“ \vdash_P ” denotes one step of SLD resolution.

ordering \succ on goals such that $\text{append}(X, Y, Z) \succ \text{append}(X', Y, Z')$. Consequently, such goals should not be allowed.

Secondly, universal termination requires showing termination of all possible goals that can arise during execution. To be practical, a proof technique to show termination must instead rely on the static structure of the program, for instance, relationships between heads and bodies of clauses. Hence, we must insure that the relationships between heads and bodies of clauses can be *translated* to goals. In Section 4, we show that if the head and body of clauses are compared but the comparison is restricted to the structure described by the mode, then the ordering relation holds for *any* well-moded goal.

To restrict orderings to the structure described by the mode-dependency, we define a transformation, τ , which, given a term and a mode-annotation, strips the term of any structure not explicitly indicated in the mode-annotation. This transformation is extended to atoms. We can then use syntactic orderings to compare the head and the body of clauses. In Section 5 we present some simple syntactic orderings adapted from the orderings commonly used in term rewriting theory (see [8] for a survey). Our adaptation of these orderings allows the use of *inter-argument relationships*⁵ which are sometimes needed to prove termination of programs with local variables.

Consider the following illustrative example:

Example 1

$$\begin{aligned} & \text{append}(\text{nil}, Y, Y). \\ & \text{append}(A.X, Y, A.Z) \leftarrow \text{append}(X, Y, Z). \end{aligned}$$

Let us assume that the mode-dependency of `append` is

$$\text{append}(\text{list}(\?), \text{list}(\?), \?) \rightarrow \text{append}(\text{list}(\?), \text{list}(\?), \text{list}(\?))$$

The transformation τ applied to the the head and the body of the clause gives (\square denotes arbitrary terms)

$$\begin{aligned} \tau(\text{append}(A.X, Y, A.Z), \text{append}(\text{list}(\?), \text{list}(\?), \?)) &= \text{append}(\square.X, Y, \square) \\ \tau(\text{append}(X, Y, Z), \text{append}(\text{list}(\?), \text{list}(\?), \?)) &= \text{append}(X, Y, \square) \end{aligned}$$

We can conclude, using some syntactic ordering $>$ (see Section 5), that $\text{append}(\square.X, Y, \square) > \text{append}(X, Y, \square)$. Hence, using this clause with a well-moded goal generates a smaller goal, so the program terminates for all such goals. A similar construction shows that any goal in the input mode $\text{append}(\?, \?, \text{list}(\?))$ also leads to termination.

Alternatively, with the input mode $\text{append}(\?, \?, \?)$, the result of the transformation is

$$\begin{aligned} \tau(\text{append}(A.X, Y, A.Z), \text{append}(\?, \?, \?)) &= \text{append}(\square, \square, \square) \\ \tau(\text{append}(X, Y, Z), \text{append}(\?, \?, \?)) &= \text{append}(\square, \square, \square) \end{aligned}$$

In this case, as expected, we cannot show termination of the program.

⁵These are also called *inter-arguments constraints*.

The two main contributions of this paper are, first, the adaptation of regular tree types to define a powerful notion of modes that can express a rich notion of directionality, and, second, the adaptation of syntactic orderings to the context of well-moded logic programs and their use to prove universal termination⁶. Our method highlights the distinction between directionality, orderings, and the use of inter-argument relationships. We believe that our notion of directionality can also be used for applications other than termination such as specifying control in concurrent logic languages. Lastly, for programs that use partially instantiated structures, such as difference lists, the treatment of directionality is a difficult problem. We suggest in section 7 an extension to our approach to give directionality to such programs. A promising alternative, which we are studying, is the use of linear logic in defining *directional logic programs* [23].

1.1 Related work

The problem of improving directionality information, i.e., to go beyond the simple ground/non-ground characterization, has been studied by many recent works. In [18, 5, 25, 26], a combination of modes, orderings and inter-argument relationships provides a more powerful, but more abstract, notion of directionality. Informally, the “input \Leftrightarrow ground” idea is replaced by the notion “input \Leftrightarrow terms acceptable by the ordering relation”. This technique resembles our use of the transformation τ with the mode “?”. However, we believe that there are advantages in distinguishing between the issues of directionality and orderings.

An approach similar to our notion of modes is the notion of *integrated types* of [12] which have been used in the context of proving termination of programs by [28]. Although, our approach is more general than [28] because of our treatment of mode-annotations and orderings, the main differences between our modes and the *integrated types* appears when we discuss the extension of our modes with implication and quantification in section 7.

We show termination using purely syntactic orderings. By allowing the use of inter-argument relationships, our approach can profit from the many works concerned with deriving inter-argument relationships (e.g. [17, 5, 11]). For instance, one could express the notion of propagating inter-argument relationships to prove termination (as in [17, 25, 26]) by appropriate inference systems for orderings.

It should be pointed out that the use of inter-argument inequality relationships as done in [17, 26] is, from a theoretical point of view, more powerful than our use of syntactic orderings with inter-argument monotonicity relationships. Naturally, our method can be adapted to use inter-argument relationships, it is only a matter of expressing reasoning with these inter-argument relationships in term of inference systems. An ever more powerful tool would be the use of polynomial interpretations (see [8]) with inter-argument polynomial inequality relationships. Reasoning with these is however much harder, so, in practice one most often rely on syntactic orderings. It should be noted that even though our approach uses only syntactic ordering and monotonicity relationships, it can show the termination of all the classical examples such as quicksort and mergesort and even of examples that seemed to require inter-argument inequality relationships such as Plumer’s PERM example or the arithmetic expression parser

⁶While the use of syntactic orderings can clearly be automated, we do not address the issue of the automation of our treatment of modes.

in Section 6. This is also noted by [20] whose approach can be, we believe, simulated in our methodology.

Since all that is really necessary to prove termination is to show that recursive calls to predicates correspond to some reduction, many authors [25, 26, 17] have either relied on the call graph generated by the program or on the treatment of the whole program as an interdependent system needed to be solved. In comparison, in this work we rely on comparisons between head and body of clauses. Although this is often simpler and it is theoretically sufficient, it might require more ordering information than these other methods. In practice, a rapid study of the call graph, if only to check that there are indeed some recursive predicates, should be part of our method as well.

The approaches of [20, 10] transform logic programs into rewrite systems and then perform termination analysis using the classical tool of rewrite theory. In contrast, our approach is more straightforward in adapting the techniques from rewrite theory directly to logic programs.

We believe that the work of [2] , although it concentrates on ground goals, could provides an abstract framework characterizing our use of syntactic orderings. The use of syntactic orderings and “don’t care” types is also suggested in [9]. That approach is however, geared toward the development of programs rather than an a posteriori test of termination.

2 Mode-Annotations

2.1 Syntax of Mode-annotations

Terms and mode-annotations are defined using the following syntactic categories:

$$\begin{aligned}
 X &\in \mathcal{V} && \text{Variables} \\
 f &\in \mathcal{F} && \text{Function symbols} \\
 t &\in \mathcal{T} && \text{Terms} \\
 \alpha &\in \mathcal{M} && \text{Mode_variables} \\
 a &\in \mathcal{A} && \text{Mode_annotations}
 \end{aligned}$$

The expressions belonging to the categories \mathcal{T} and \mathcal{A} are defined with the following context-free syntax:

$$\begin{aligned}
 t &::= X \mid f(t_1, \dots, t_k) \\
 a &::= \textit{ground} \mid ? \mid \alpha \mid f(a_1, \dots, a_k) \mid a_1 + a_2 \mid \mathbf{fix} \alpha.a
 \end{aligned}$$

We impose two restrictions on the syntax of mode-annotations. First, mode-annotations must be *closed a-terms*⁷, i.e., they contain no free occurrences of mode variables. Second, they must be *discriminative*⁸, i.e, for mode-annotations expressed as a sum $a_1 + a_2$, the outermost symbol of a_1 and a_2 must be distinct.

As a notational convenience we allow, in addition to *ground*, mode-annotations such as *int*, whose semantics is well defined as a subset of ground terms. We also use the following

⁷In section 7 we discuss the need to drop this condition and extend mode-annotations to full regular trees [19] to handle partially instantiated structures.

⁸This condition (taken from [15]) is imposed for reasons of efficiency (see Appendix B).

shorthand notation for mode-annotations that describe commonly used structures.

$$\begin{aligned} list(a) &= \mathbf{fix}\alpha.(nil + a.\alpha) \\ tree(a) &= \mathbf{fix}\alpha.(nil + node(a, \alpha, \alpha)) \end{aligned}$$

Note that $list(a)$ denotes the set of lists whose elements satisfy the annotation a . Similarly, $tree(a)$ denotes the set of binary trees whose elements satisfy the annotation a .

Terms and mode-annotations are syntactically related by the notion of *well-modedness* of terms. Well-modedness is defined via the notion of *mode contexts*. A *mode context*, is a finite set of (unique) mode-annotation assertions for variables, each of the form $X : a$ ⁹. Mode contexts are needed to disambiguate the role of variables in non-ground terms. For instance, variables occurring in atomic goals, should be interpreted as terms with the mode annotation $?$; while, in the head of a clause, variables are intended as parameters to which the context binds mode-annotations.

Definition 1 A term t is *well-moded* in a mode-annotation a relatively to a mode context Γ (denoted by $\Gamma \vdash t : a$), if the judgement $\Gamma \vdash t : a$ is derivable using the inference rules given in Appendix A.

For example, $\{X : list(?)\} \vdash f(X) : f(list(?))$, $\emptyset \vdash X : ?$ and $\{X : list(a), Y : a\} \vdash Y.X : list(a)$. By extension, we write $\Gamma \vdash \{t_1 : a_1, \dots, t_n : a_n\}$ as an abbreviation for $\Gamma \vdash t_1 : a_1, \dots$, and $\Gamma \vdash t_n : a_n$.

Definition 2 A mode-annotation a_1 *covers* a mode-annotation a_2 , (written $a_1 \succeq a_2$) if $a_1 \succeq a_2$ is derivable using the inclusion checking rules (adapted from [15]) given in Appendix B.

For example, $? \succeq list(?)$, $? \succeq ground$, and $a_1 + a_2 \succeq a_1$.

The following properties will be used later to show the soundness of LD-resolution relatively to the notion of well-modedness.

Proposition 2 If $\Gamma \vdash f(t_1, \dots, t_n) : a$ then there exists a_1, \dots, a_n such that

- 1) $\forall i \in [1 \dots n]. \Gamma \vdash t_i : a_i$ and
- 2) for any set of terms $\{s_1, \dots, s_n\}$, $\forall i \in [1 \dots n]. \Gamma \vdash s_i : a_i \Rightarrow \Gamma \vdash f(s_1, \dots, s_n) : a$.

Proof: By induction on the length of the derivation of $\Gamma \vdash t : a$ where $t = f(t_1, \dots, t_n)$

- (basis) length = 2 : The only possibility is the use of the introduction rule for function or predicate symbols. Thus, a must have the form $f(a_1, \dots, a_n)$ and the result is immediate.
- (Induction step) length $> n$: Consider the last inference. Four cases are possible.
 - Introduction of a function or predicate symbol : as above.

⁹A mode context can also be viewed as a partial mapping: $\mathcal{V} \rightarrow \mathcal{A}$.

- Introduction of $+$: $a = a' + a''$. The last inference was

$$\frac{\Gamma \vdash t : a'}{\Gamma \vdash t : a' + a''}$$

First note that $\Gamma \vdash t : a'$, so by the inductive hypothesis, there exist a_1, \dots, a_n such that $\forall i. \Gamma \vdash t_i : a_i$. Moreover, for any set $\{s_1, \dots, s_n\}$, $\forall i. \Gamma \vdash s_i : a_i$ implies $\Gamma \vdash f(s_1, \dots, s_n) : a'$, this, in turn, implies $\Gamma \vdash f(s_1, \dots, s_n) : a' + a''$.

- Use of inclusion : $b \subseteq a$. The last inference was

$$\frac{\Gamma \vdash t : b}{\Gamma \vdash t : a}$$

We have $\Gamma \vdash t : b$ so by the inductive hypothesis there exist a_1, \dots, a_n such that $\forall i. \Gamma \vdash t_i : a_i$. Moreover, for any set $\{s_1, \dots, s_n\}$, $\forall i. \Gamma \vdash s_i : a_i$ implies $\Gamma \vdash f(s_1, \dots, s_n) : b$, this, in turn, implies $\Gamma \vdash f(s_1, \dots, s_n) : a$.

- Introduction of **fix**: similar to the above cases.

Proposition 3 (Instantiation I) If $\vdash t : a$ then $\vdash t\theta : a$ where θ is any substitution.

Proof: By induction on the structure of t .

- $t = X$: Then $a = ?$. Then $\vdash X\theta : a$, since $\vdash t : ?$ for any term t .
- $t = f(t_1, \dots, t_n)$: So we must have that $\vdash t_1 : a_1, \dots, \vdash t_n : a_n$ (by prop. 2). By the inductive hypothesis, $\vdash t_1\theta : a_1, \dots, \vdash t_n\theta : a_n$, so $\vdash f(t_1, \dots, t_n)\theta : a$. \square

If $\Gamma = \{X_1 : a_1, \dots, X_n : a_n\}$ then $\Gamma\theta = \{\theta(X_1) : a_1, \dots, \theta(X_n) : a_n\}$ is a set of (term, mode-annotation) pairs that denotes the application of a substitution θ to the mode context Γ .

Lemma 4 (Instantiation II) If $\Gamma \vdash t : a$ and $\Gamma' \vdash \Gamma\theta$ then $\Gamma' \vdash t\theta : a$.

Proof: By induction on the structure of t .

- $t = X$: If $(X : a) \in \Gamma$, then $\Gamma' \vdash X\theta : a$ by hypothesis. If $X \notin \text{dom}(\Gamma)$ then, since $\Gamma \vdash X : a$, a must be ?. Hence, $\vdash X\theta : a$, so $\Gamma' \vdash X\theta : a$.
- $t = f(t_1, \dots, t_n)$: Using the inductive hypothesis on $\Gamma \vdash t_i : a_i$, we obtain $\Gamma' \vdash t_i\theta : a_i$. Hence, $\Gamma' \vdash t\theta : a$. \square

Lemma 5 (lifting) If $\Gamma_0 \vdash t\theta : a$ then there exists a Γ such that $\Gamma \vdash t : a$ and $\Gamma_0 \vdash \Gamma\theta$.

Proof: Let us first define the following transformation of a term and a mode context into a mode-annotation:

$$\begin{aligned}\rho(X, \Gamma) &= \begin{cases} ? & \text{if } X \notin \text{dom}(\Gamma) \\ \Gamma(X) & \text{otherwise} \end{cases} \\ \rho(f(t_1, \dots, t_n), \Gamma) &= f(\rho(t_1, \Gamma), \dots, \rho(t_n, \Gamma)).\end{aligned}$$

ρ satisfies two important properties:

- $\Gamma \vdash t : \rho(t, \Gamma)$.
- If $\Gamma \vdash t : a$ then $\rho(t, \Gamma) \subseteq a$. This can be proved by induction.

Now we can prove the lemma by constructing the appropriate Γ using induction on t .

- $t = X$: Let $\Gamma = \{X : \rho(X\theta, \Gamma_0)\}$. Hence, we have $\Gamma_0 \vdash \Gamma\theta$, since $\Gamma_0 \vdash X\theta : \rho(X\theta, \Gamma_0)$, and $\Gamma \vdash t : a$ since $\rho(t\theta, \Gamma_0) \subseteq a$.
- $t = f(t_1, \dots, t_n)$: Let $\Gamma = \bigcup_{1 \leq i \leq n} \Gamma_i$ where the Γ_i are recursively constructed using the t_i and Γ_0 . We trivially have that this construction is well-defined and that $\Gamma_0 \vdash \Gamma\theta$. Again, we rely on the property that there exist mode-annotations a_1, \dots, a_n such that for all $i \in [1 \dots n]$ $\Gamma_0 \vdash t_i\theta : a_i$ to conclude that $\Gamma \vdash t_i : a_i$ and so that $\Gamma \vdash t : a$. \square

2.2 Semantics of Mode-annotations

As before, let \mathcal{T} denote the set of all (possibly non-ground) terms. Note that $(2^{\mathcal{T}}, \subseteq)$ is a reflexive partial order where \subseteq is the set-inclusion relation on sets of terms. Let \mathcal{H} denote the Herbrand universe of ground terms.

Definition 3 The semantic function for mode-annotations $\llbracket \cdot \rrbracket \eta : \mathcal{A} \longrightarrow 2^{\mathcal{T}}$ is defined using mode valuations $\eta \in \mathcal{M} \longrightarrow 2^{\mathcal{T}}$ as follows:

- $\llbracket ? \rrbracket \eta = \mathcal{T}$.
- $\llbracket \alpha \rrbracket \eta = \eta(\alpha)$.
- $\llbracket \text{ground} \rrbracket \eta = \mathcal{H}$.
- $\llbracket f(a_1, \dots, a_k) \rrbracket \eta = \{f(t_1, \dots, t_k) \mid t_1 \in \llbracket a_1 \rrbracket \eta \dots t_k \in \llbracket a_k \rrbracket \eta\}$
- $\llbracket a_1 + a_2 \rrbracket \eta = \llbracket a_1 \rrbracket \eta \cup \llbracket a_2 \rrbracket \eta$.
- $\llbracket \text{fix}_{\alpha}.a \rrbracket \eta =$ the least $S \subseteq \mathcal{T}$ such that $S = \llbracket a \rrbracket (\eta[S/\alpha])$.

Since we restrict mode-annotations to closed a-terms, the mode valuation η is only a notational formalism that is convenient in defining the semantics of **fix**. In what follows we generally omit η .

For example, the semantics of recursively defined lists and trees is:

$\llbracket \text{list}(a) \rrbracket =$ the least $S \subseteq \mathcal{T}$ such that $\text{nil} \in S$ and if $s \in S$ then $t.s \in S$ for all $t \in \llbracket a \rrbracket$.

$\llbracket tree(a) \rrbracket =$ the least $S \subseteq \mathcal{T}$ such that $nil \in S$ and if $s, s' \in S$ then $node(t, s, s') \in S$ for all $t \in \llbracket a \rrbracket$.

We can now provide a semantic interpretation for $\Gamma \vdash t : a$. First, since the variables in the domain of Γ are intended as parameters, we need the notion of *valuations*. A *valuation* ζ is a partial mapping $\zeta : V \rightarrow \mathcal{T}$ ¹⁰. A valuation is said to *respect* a mode context Γ if ζ and Γ have the same domain and for all $X \in Dom(\Gamma)$, $\zeta(X) \in \llbracket (\Gamma(X)) \rrbracket$. A valuation is identified with its extension to terms. Therefore, the semantics counterpart of $\Gamma \vdash t : a$ is the following: for all valuations ζ which respect Γ , $\zeta(t) \in \llbracket a \rrbracket$.

3 Mode assignment to programs

An n-ary *mode* is an n-tuple over *mode-annotations*. An n-ary mode for an n-ary predicate p is syntactically specified as : “ $p(a_1, \dots, a_n)$ ”.

Definition 4 An n-ary *mode-dependency* is a pair of n-ary modes (written $m^i \rightarrow m^o$) where m^i (m^o) is called the input (output) mode.

Mode-dependencies for a predicate specify the structure of the input and output arguments to the predicate (treated as a directional procedure).

A moded clause (goal) is a clause (goal) together with an assignment of mode-dependencies to every literal occurring in the clause (goal). The notion of a well-moded clause is now defined:

Definition 5 A moded clause $A \leftarrow B_1, \dots, B_n$, with the assigned mode-dependencies $m_a^i \rightarrow m_{B_1}^o, m_{B_1}^i \rightarrow m_{B_2}^o, \dots, m_{B_n}^i \rightarrow m_{B_n}^o$ is *well-moded* if for all mode contexts Γ , the following hold:

1. $\forall j < n. \Gamma \vdash A : m_a^i, \Gamma \vdash B_1 : m_{B_1}^o \quad \dots \quad \Gamma \vdash B_j : m_{B_j}^o \quad \text{implies} \quad \Gamma \vdash B_{j+1} : m_{B_{j+1}}^i$
2. $\Gamma \vdash A : m_a^i, \Gamma \vdash B_1 : m_{B_1}^o \quad \dots \quad \Gamma \vdash B_n : m_{B_n}^o \quad \text{implies} \quad \Gamma \vdash A : m_a^o$

A logic program P is *well-moded* if all clauses in P are well-moded. For example, the program defining `append` in example 1 is well-moded with respect to the assigned mode-dependency for `append`.

Definition 6 A moded goal C_1, \dots, C_n is *well-moded* if the following hold:

1. $\vdash C_1 : m_{C_1}^i$
2. For all mode contexts Γ , and for all $j \in [1 \dots n - 1]$

$$\Gamma \vdash C_1 : m_{C_1}^o \quad \dots \quad \Gamma \vdash C_j : m_{C_j}^o \quad \text{implies} \quad \Gamma \vdash C_{j+1} : m_{C_{j+1}}^i$$

For example, with the mode-dependency for `append` assumed in section 1, the goals `append([X, 1, 2], [Y], Z)` and `append([X, 1, 2], [Y], Z), append(Z, [Y, 1], U)` are well-moded.

The well-modedness of a moded clause or a moded goal can be checked as follows: For each implication condition in the definition of well-modedness, first, the elimination rules given in

¹⁰A valuation can also be viewed as a substitution.

Appendix A are used to construct a most general mode context Γ in which the atom(s) in the antecedent of the implication are in their assigned modes. Next, this mode context Γ is used to check (using the introduction rules) whether the atom in the consequent is in its assigned mode.

Note that there can be multiple assignments of mode-dependencies for predicates for which the program is well-moded. For example, even with the mode-dependency

$$\text{append}(?, ?, \text{list}()) \rightarrow \text{append}(\text{list}(), \text{list}(), \text{list}())$$

the program defining `append` in Section 1 is well-moded. On the other hand, with the alternate mode-dependency

$$\text{reverse}(?, \text{list}()) \rightarrow \text{reverse}(\text{list}(), \text{list}())$$

the second clause defining `reverse` in Section 5 is not well-moded. The body literals in the clause have to be exchanged in order for the clause to be well-moded with the above mode-dependency. We view the resulting program for `reverse` as having a directionality that is distinct from the original version. Though, from now on we assume that a unique mode-dependency is assigned to each predicate occurring in a program, our approach can easily be extended to handle multiple mode-dependencies for predicates.

Throughout the rest of this paper “programs” and “goals” denote well-moded programs and well-moded goals.

The following theorems express the “correctness” of the notion of mode-dependency.

Theorem 6 Mode-consistency of LD resolution One step of LD resolution transforms a well-moded goal into another well-moded goal.

Proof: Consider a well-moded goal B_1, \dots, B_n . Let B_1 be the selected atom and $B_1\theta = A\theta$ where $A \leftarrow A_1, \dots, A_m$ is a well-moded clause. The new goal derived from one step of SLD resolution is $A_1\theta, \dots, A_m\theta, B_2\theta, \dots, B_n\theta$. We must show that this new goal is also well-moded i.e., show that

- $\vdash A_1\theta : m_{A_1}^i$.
 Since $\vdash B_1 : m_{B_1}^i$ (the initial goal is well-moded)
 $\Rightarrow \vdash B_1\theta : m_{B_1}^i$ (by proposition 3)
 $\Rightarrow \vdash A\theta : m_A^i$ (since $B_1\theta = A\theta$)
 $\Rightarrow \exists \Gamma'$, such that $\Gamma' \vdash A : m_A^i$ and $\emptyset \vdash \Gamma'\theta$ (by lemma 5)
 $\Rightarrow \Gamma' \vdash A_1 : m_{A_1}^i$ (since $A \leftarrow A_1, \dots, A_m$ is a well-moded clause)
 $\Rightarrow \vdash A_1\theta : m_{A_1}^i$ (by lemma 4).
- for all Γ , $\Gamma \vdash A_1\theta : m_{A_1}^o \dots \Gamma \vdash A_j\theta : m_{A_j}^o \Rightarrow \Gamma \vdash A_{j+1}\theta : m_{A_{j+1}}^i$.
 From the antecedent, and since $\vdash A\theta : m_A^i \Rightarrow \Gamma \vdash A\theta : m_A^i$ (trivially).
 $\Rightarrow \exists \Gamma'$ such that $\Gamma' \vdash A : m_A^i$, $\Gamma' \vdash A_1 : m_{A_1}^o, \dots, \Gamma' \vdash A_j : m_{A_j}^o$ and $\Gamma \vdash \Gamma'\theta$ (by lemma 5).
 $\Rightarrow \Gamma' \vdash A_{j+1} : m_{A_{j+1}}^i$ (since $A \leftarrow A_1, \dots, A_m$ is a well-moded clause).
 $\Rightarrow \Gamma \vdash A_{j+1}\theta : m_{A_{j+1}}^i$ (by lemma 4).

- for all $\Gamma, \Gamma \vdash A_1\theta : m_{A_1}^o \dots \Gamma \vdash A_m\theta : m_{A_m}^o \Rightarrow \Gamma \vdash B_2\theta : m_{B_2}^i$, and so on.
 From the antecedent, and since $\vdash A\theta : m_A^i \Rightarrow \Gamma \vdash A\theta : m_A^i$.
 $\Rightarrow \Gamma \vdash A\theta : m_A^o$ (since $A \leftarrow A_1, \dots, A_n$ is a well-moded clause).
 $\Rightarrow \Gamma \vdash B_1\theta : m_{B_1}^o$ (since $A\theta = B_1\theta$)
 $\Rightarrow \exists \Gamma'$, such that $\Gamma' \vdash B_1 : m_{B_1}^o$ and $\Gamma \vdash \Gamma'\theta$ (by lemma 5)
 $\Rightarrow \Gamma' \vdash B_2 : m_{B_2}^i$ (since the original goal is well-moded).
 $\Rightarrow \Gamma' \vdash B_2\theta : m_{B_2}^i$ (by lemma 4). □

From the above theorem we see that LD resolution preserves well-modedness of goals.

Theorem 7 Correctness of Mode dependency Consider a well-moded atomic goal $p(t_1, \dots, t_n)$ with an assigned mode-dependency $p : m_p^i \rightarrow m_p^o$. If the LD resolution of this goal with respect to a well-moded logic program succeeds with a computed answer substitution θ then $\vdash p(t_1, \dots, t_n)\theta : m^o$

Proof: By induction on the length of the SLD refutation.

- length = 1. There exists a well-moded clause $p(u_1, \dots, u_n) : p(m_1^i, \dots, m_n^i) \rightarrow p(m_1^o, \dots, m_n^o)$ such that $p(t_1, \dots, t_n)\theta = p(u_1, \dots, u_n)\theta$. Thus,
 $\vdash p(t_1, \dots, t_n) : p(m_1^i, \dots, m_n^i) \Rightarrow \vdash p(t_1, \dots, t_n)\theta : p(m_1^i, \dots, m_n^i)$ (by proposition 3).
 $\Rightarrow \vdash p(u_1, \dots, u_n)\theta : p(m_1^i, \dots, m_n^i) \Rightarrow \vdash p(u_1, \dots, u_n)\theta : p(m_1^o, \dots, m_n^o)$ (since $p(u_1, \dots, u_n)$ is well-moded).
- length > 1. There exists a well-moded clause $A \leftarrow A_1, \dots, A_n$ such that the goal $p(t_1, \dots, t_n)$ unifies with the head of the clause. The new goal is $A_1\theta, \dots, A_n\theta$ which is well-moded (by Lemma 6). By induction hypothesis, each of the subgoals $A_1\theta, \dots, A_n\theta$ succeed with the results $(A_i\sigma)$ in the corresponding output mode. $\vdash A_1\sigma : m_{A_1}^o, \dots, \vdash A_n\sigma : m_{A_n}^o$. Hence, there exists a Γ' such that $\Gamma' \vdash A_1 : m_{A_1}^o, \dots, \Gamma' \vdash A_n : m_{A_n}^o$ and $\emptyset \vdash \Gamma'\theta$. (by lemma 5) Furthermore, $\Gamma' \vdash A : m_A^i$ (since the goal is well-moded, i.e. $\vdash A : m_A^i$). Hence, $\Gamma' \vdash A : m_A^o$ (since the clause is well-moded). Hence, $\vdash A\theta : m_A^o$ (by lemma 4) $\Rightarrow \vdash p(t_1, \dots, t_n)\theta : m_p^o$ □

4 Characterizing orderings to prove termination

4.1 Simplifying terms with mode-annotations

To prove termination it is not enough to show that the head of a clause is “greater than” the body since goals that unify with the head of a clause might be more general than the head itself. Consequently, we need to compare *generalized* versions of the head and the body. These generalizations depend on the mode-dependency of the predicates in the clause, and are obtained by the transformation τ defined below.

Definition 7 Given a term t over a set of function symbols \mathcal{F} a set of variables \mathcal{V} , and a mode-annotation a , the transformation $\tau : \mathcal{T} \times \mathcal{A} \rightarrow \mathcal{T}(\mathcal{F} \cup (\mathcal{V} \times \mathcal{A}) \cup \{\square, \perp\})$ which generalizes t to the most general term in the mode-annotation a , is defined recursively as follows:

$$\begin{aligned}
\tau(X, a) &= \begin{cases} \square & \text{if } a = ? \\ X^a & \text{otherwise} \end{cases} \\
\tau(f(t_1, \dots, t_n), ?) &= \square \\
\tau(f(t_1, \dots, t_n), \text{ground}) &= f(\tau(t_1, \text{ground}), \dots, \tau(t_n, \text{ground})) \\
\tau(f(t_1, \dots, t_n), g(a_1, \dots, a_m)) &= \begin{cases} f(\tau(t_1, a_1), \dots, \tau(t_n, a_n)) & \text{if } g = f \text{ and } n = m \\ \perp & \text{otherwise} \end{cases} \\
\tau(f(t_1, \dots, t_n), \mathbf{fix}\alpha.a) &= \tau(f(t_1, \dots, t_n), a[\mathbf{fix}\alpha.a/\alpha]) \\
\tau(f(t_1, \dots, t_n), a_1 + a_2) &= \text{sup}(\tau(f(t_1, \dots, t_n), a_1), \tau(f(t_1, \dots, t_n), a_2))
\end{aligned}$$

where the binary operator sup on terms over $\mathcal{F} \cup (\mathcal{V} \times \mathcal{A}) \cup \{\square\}$ is defined as follows:

$$\begin{aligned}
\text{sup}(t_1, t_2) &= \square \text{ if } t_1 = \square \text{ or } t_2 = \square \\
\text{sup}(t, \perp) &= t \\
\text{sup}(\perp, t) &= t \\
\text{sup}(f(t_1, \dots, t_n), f(u_1, \dots, u_n)) &= f(\text{sup}(t_1, u_1), \dots, \text{sup}(t_n, u_n))
\end{aligned}$$

The value \square denotes arbitrary terms (i.e., a lack of any structure information) while the value \perp denotes an erroneous application of τ , i.e., an application $\tau(t, a)$ where no instance of t can be in the mode a . Furthermore, we require τ to be strict in \perp . Note that the binary operator sup operates only on terms such that either one of the two terms is a \square or a \perp , or the two terms have the same outermost function symbol.

For a term t and a mode-annotation a , the transformation τ produces the “generalization” of t compatible with the mode-annotation a . This means that subterms of t with corresponding mode-annotations $?$ are replaced by the symbol \square , and that variables are tagged with the corresponding mode-annotation. For example, $\tau(1.X, \text{list}(?)) = \square.X^{\text{list}(?)}$, and $\tau(f(X, X), f(g(?), \text{ground})) = f(X^{g(?)}, X^{\text{ground}})$.

In the following, we identify τ with its (obvious) extension to pairs of atom and mode. We also usually omit the tag on the variables after the application of the transformation τ , unless, as in the last example above, such an omission would be ambiguous. Notice, however, that if $\emptyset \vdash t : a$, then $\tau(t, a)$ does not contain any tagged variables.

Lemma 8 Let t be a term in a mode-annotation a . Then, for every substitution σ , $\tau(t, a) = \tau(t\sigma, a)$.

Proof: (by induction on the construction of $\tau(t, a)$) Consider the possible cases:

- a is *ground*, then t must be a ground term, so $t = t\sigma$.
- a is $?$, then $\tau(t, a) = \square = \tau(t\sigma, a)$.
- a is $f(a_1, \dots, a_n)$, then since t is in mode a , we must have $t = f(t_1, \dots, t_n)$ and, naturally, $t\sigma = f(t_1\sigma, \dots, t_n\sigma)$; by the inductive hypothesis $\tau(t_i, a_i) = \tau(t_i\sigma, a_i)$, so, the result follows.

- a is $\mathbf{fix}\alpha.a$, then by the inductive hypothesis $\tau(t, a[\mathbf{fix}\alpha.a/\alpha]) = \tau(t\sigma, a[\mathbf{fix}\alpha.a/\alpha])$, and the result follows trivially.
- a is $a_1 + a_2$, then again by the inductive hypothesis the result follows. \square

Lemma 9 Given a substitution σ , the substitution $\nu \equiv_{def} \{\{X^a \rightsquigarrow \tau(X\sigma, a) \mid X \in \text{dom}(\sigma) \wedge a \in \mathcal{A}\}\}$ is such that for any term t and any mode-annotation a (t does not have to be in the mode-annotation a)

$$\tau(t\sigma, a) = \tau(t, a)\nu$$

Proof: (by induction on the definition of $\tau(t, a)$) First, it is easy to see (by induction) that if $\tau(t, a) = \perp$ then for any σ , $\tau(t\sigma, a) = \perp$, so $\tau(t\sigma, a) = \perp = \tau(t, a)\nu$. Similarly, if $\tau(t, a) = \square$ then $\tau(t\sigma, a) = \square = \tau(t, a)\nu$.

Next, if $t = X$, and $a_1 \neq ?$ then $\tau(t, a)\nu = X^a\nu = \tau(X\sigma, a_1) = \tau(t\sigma, a)$. If X is not in the domain of σ then $X^a\nu = X^a = \tau(X, a) = \tau(X\sigma, a)$. Let us now consider the cases where $t = f(t_1, \dots, t_n)$:¹¹

1. if a is *ground*, then

$$\begin{aligned} \tau(t, \text{ground})\nu &= f(\tau(t_1, \text{ground})\nu, \dots, \tau(t_n, \text{ground})\nu) \\ &=_{Ind} f(\tau(t_1\sigma, \text{ground}), \dots, \tau(t_n\sigma, \text{ground})) = \tau(t\sigma, \text{ground}) \end{aligned}$$

2. if a is $f(a_1, \dots, a_n)$, then

$$\begin{aligned} \tau(t, f(a_1, \dots, a_n))\nu &= f(\tau(t_1, a_1)\nu, \dots, \tau(t_n, a_n)\nu) \\ &=_{Ind} f(\tau(t_1\sigma, a_1), \dots, \tau(t_n\sigma, a_n)) = \tau(t\sigma, f(a_1, \dots, a_n)) \end{aligned}$$

3. if a is $\mathbf{fix}\alpha.a$, then

$$\tau(t, \mathbf{fix}\alpha.a)\nu = \tau(t, a[\mathbf{fix}\alpha.a/\alpha])\nu =_{Ind} \tau(t\sigma, a[\mathbf{fix}\alpha.a/\alpha]) = \tau(t\sigma, \mathbf{fix}\alpha.a)$$

4. if a is $a_1 + a_2$, first, by a simple induction on the definition of *sup* we can show that $\text{sup}(t_1, t_2)\nu = \text{sup}(t_1\nu, t_2\nu)$. Now we have

$$\begin{aligned} \tau(t, a_1 + a_2)\nu &= \text{sup}(\tau(t, a_1), \tau(t, a_2))\nu = \text{sup}(\tau(t, a_1)\nu, \tau(t, a_2)\nu) \\ &=_{Ind} \text{sup}(\tau(t\sigma, a_1), \tau(t\sigma, a_2)) = \tau(t, a_1 + a_2) \end{aligned}$$

\square

¹¹The symbol $=_{Ind}$ denotes equality by the inductive hypothesis

4.2 Simple Termination Theorem

Theorem 10 A logic program P universally terminates if there exists a well-founded stable ¹² ordering \succ , such that for each clause $A \leftarrow B_1, \dots, B_n$ (with mode-dependencies: $A : a^i \rightarrow a^o, B_1 : b_1^i \rightarrow b_1^o, \dots, B_n : b_n^i \rightarrow b_n^o$)

$$\forall j. \tau(A, a^i) \succ \tau(B_j, b_j^i)$$

The proof of this theorem follows from the proof of the more complex version of this theorem given in subsection 4.4.

This theorem can be used to prove the termination of simple programs such as `append` or `split`. However, for many programs, such as `quicksort`, we need to use inter-argument relationships to complete the termination proof.

4.3 Inter-argument Relationships

Given a predicate p with a mode dependence $p(a_1^i, \dots, a_n^i) \rightarrow p(a_1^o, \dots, a_n^o)$, an inter-argument relationship on p is expressed by the syntax:

$$p(X_1, \dots, X_n) : R(\tau(X_1, a_1^o), \dots, \tau(X_n, a_n^o))$$

and has the interpretation:

$$\vdash p(u_1, \dots, u_n) : p(a_1^o, \dots, a_n^o), \text{ and } p(u_1, \dots, u_n) \text{ succeeds, implies } R(\tau(u_1, a_1^o), \dots, \tau(u_n, a_n^o))$$

In defining an inter-argument relationship, the transformation τ is needed because we must allow inter-argument relationships to hold on non-ground terms.

For example, the semantic property that in the output tuple of `append` the length of the first argument is less than or equal to the length of the third can be expressed by:

$$\text{append}(X, Y, Z) : X \leq Z$$

Inter-argument relationships can also be defined for (conjunction of) literals with the syntax

$$L_1, \dots, L_n : R(\tau(t_1, a_1^o), \dots, \tau(t_k, a_k^o))$$

where t_1, \dots, t_k are terms appearing in the literals and a_1, \dots, a_k are the corresponding mode-annotations. The interpretation of this relationship is: for all substitutions σ ,

$$\vdash L_1\sigma : m_{L_1}^o, \dots, L_n\sigma : m_{L_n}^o, \text{ and } L_1\sigma, \dots, L_n\sigma \text{ succeed, implies } R(\tau(t_1\sigma, a_1^o), \dots, \tau(t_k\sigma, a_k^o))$$

As an example consider the following relationship:

$$\text{append}(X, [A|L], Y), \text{append}(X, L, Z) : Y > Z$$

In general a set of inter-argument relationships can be expressed as a single relationship on a conjunction of the literals. Consequently, in the following, we assume that all inter-argument relationships have been appropriately defined on conjunctions of literals.

¹²An ordering \succ is stable if $t_1 \succ t_2$ implies for all substitutions σ , $\sigma(t_1) \succ \sigma(t_2)$.

We define an ordering relation \succ via an inference system \vdash_{\succ} . Such an inference system can use an inter-argument relationship $(L_1, \dots, L_n : R)$ as a hypothesis for inferences. Thus, derivations have the form:

$$R \vdash_{\succ} t_1 \succ t_2.$$

For example, we define in the next section an inference system that establishes an ordering relation \succ , by combining subterm ordering, inter-argument monotonicity relationships, and lexicographic ordering.

Definition 8 We say that an inference system \vdash_{\succ} is *stable* if

$$R \vdash_{\succ} t_1 \succ t_2 \text{ implies } \forall \sigma. R\sigma \vdash_{\succ} t_1\sigma \succ t_2\sigma$$

Lemma 11 If \vdash_{\succ} is stable then given an inter-argument relationship $L_1, \dots, L_n : R$ such that $R \vdash_{\succ} \tau(t_1, a_1) \succ \tau(t_2, a_2)$ and given a substitution σ such that $L_1\sigma, \dots, L_n\sigma$ are in their output mode and succeed, then $\tau(t_1\sigma, a_1) \succ \tau(t_2\sigma, a_2)$.

Proof: By the definition of inter-argument relationships, if $L_1\sigma, \dots, L_n\sigma$ are well-moded and true, then $R(\dots, \tau(t_i\sigma, a_i), \dots)$ is true as well. By lemma 9 there exists a substitution ν such that for any t and a , $\tau(t\sigma, a) = \tau(t, a)\nu$. Hence, $R(\dots, \tau(t_i\sigma, a_i), \dots) = R(\dots, \tau(t_i, a_i)\nu, \dots) = R\nu$. By stability, $R \vdash_{\succ} t_1 \succ t_2$ implies $R\nu \vdash_{\succ} t_1\nu \succ t_2\nu$. Since $R\nu$ is true, the result follows from the correctness of \vdash_{\succ} . \square

4.4 Termination Theorem

Definition 9 Given a stable inference system defining an ordering relation, a clause $A \leftarrow B_1, \dots, B_n$ is τ -*decreasing* if

$$\forall i, R_{i-1} \vdash_{\succ} \tau(A, a) \succ \tau(B_i, b_i) \tag{1}$$

where R_i is defined by the following inter-argument relationship $B_1, \dots, B_i : R_i$.

Theorem 12 A logic program \mathcal{P} terminates if there exists a stable inference system \vdash_{\succ} , defining a well-founded ordering relation \succ , such that each clause in \mathcal{P} is τ -decreasing.

Proof: Suppose, on the contrary that there exists some infinite derivation: $G_1 \vdash G_2 \vdash \dots$. Consider the (infinite) proof tree represented by this derivation. By Koenig's lemma, since the branching factor is finite, there must be some infinite branch. Consider adjacent nodes A, B along this branch. The subtree below A was obtained by resolving A with a clause $A' \leftarrow B_1, \dots, B_{i-1}, B', B_i, \dots, B_n$ such that $A\sigma = A'\sigma$ ($\sigma = mgu(A, A')$) and $B = B'\sigma\gamma$ where γ is the computed answer substitution obtained from the resolution of $B_1\sigma, \dots, B_{i-1}\sigma$.

Therefore, each of $B_1\sigma\gamma, \dots, B_{i-1}\sigma\gamma$ succeed and are in their output mode. Since the clause is τ -decreasing, $R_i \vdash_{\succ} \tau(A', a) \succ \tau(B', b)$. By lemma 11, we obtain that $\tau(A'\sigma\gamma, a) \succ \tau(B'\sigma\gamma, b)$.

By lemma 8, $\tau(A, a) = \tau(A\sigma, a) = \tau(A'\sigma, a) = \tau(A'\sigma\gamma, a)$. Similarly, we have $\tau(B, b) = \tau(B'\sigma\gamma, b)$. Since $\tau(A'\sigma\gamma, a) \succ \tau(B'\sigma\gamma, b)$, we can conclude that this infinite branch represents an infinite descending sequence. This contradicts the well-foundedness of \succ . \square

The following alternative definition of τ -decreasiveness (adapted from [10]) is consistent with the termination theorem above and has proven to be useful in practice.

Definition 10 Given a stable inference system defining a well-founded ordering relation, a clause $A \leftarrow B_1, \dots, B_n$ is said to be τ -decreasing also if

1. $\tau(A, a^i) \succ \tau(B_1, b_1^i)$
2. $\forall j, R_{j-1} \vdash_{\succ} \tau(B_{j-1}, b_{j-1}^o) \succeq \tau(B_j, b_j^i)$
3. $R_n \vdash_{\succ} \tau(B_n, b_n^o) \succeq \tau(A, a^o)$ or $R_n \vdash_{\succ} \tau(A, a^i) \succ \tau(A, a^o)$

where R_j is the inter-argument relationship of B_1, \dots, B_j .

The advantage of this alternative definition of τ -decreasiveness is that sometimes using this definition and a carefully chosen ordering, we can avoid the need for inter-argument relationships. The example 3 illustrates this possibility. Using the following proposition we can easily show that Theorem 12 still holds with this characterization of τ -decreasing clause.

Proposition 13 Given a stable inference system defining a well-founded ordering relation $>$. For all τ -decreasing clauses $A \leftarrow B_1, \dots, B_n$ and for all atoms A' and substitution σ such that $A' : a^i$ and $A'\sigma = A\sigma$, we have

- $\tau(A', a^i) > \tau(B_j\sigma\alpha_{j-1}, b_j^i)$ and $\tau(A', a^i) > \tau(B_j\sigma\alpha_j, b_j^o)$ where α_j is the computed answer substitution resulting from the evaluation of $B_1\sigma, \dots, B_j\sigma$.
- $\tau(A', a^i) > \tau(A\sigma\alpha_n, a^o)$ where α_n is the computed answer substitution resulting from the evaluation of $B_1\sigma, \dots, B_n\sigma$.

Proof: (by induction on $>$) First note that by Lemma 8 $\tau(A', a^i) = \tau(A'\sigma, a^i) = \tau(A\sigma, a^i)$. Now consider the following two cases:

- There is no B_j . In that case, if the clause is τ -decreasing we have $\tau(A, a^i) > \tau(A, a^o)$. By Lemma 11 and the stability of $>$ this implies $\tau(A\sigma, a^i) > \tau(A\sigma, a^o)$, as desired.
- There are some B_j 's. Let us do a secondary induction on n , the number of B_j 's, to prove the first part of the property.
 - $n = 1$. Since the clause is τ -decreasing we must have $\tau(A, a^i) > \tau(B_1, b_1^i)$, so $\tau(A\sigma, a^i) > \tau(B_1\sigma, b_1^i)$. Since the clause is well-moded, $\vdash B_1\sigma : b_1^i$, so the inductive hypothesis is applicable and we have $\tau(B_1\sigma, b_1^i) > \tau(B_1\sigma\alpha_1, b_1^o)$ (for α_1 the substitution generated by the execution of $B_1\sigma$). Hence, $\tau(A\sigma, a^i) > \tau(B_1\sigma\alpha_1, b_1^o)$.
 - $n > 1$. Assume $\tau(A\sigma, a^i) > \tau(B_{n-1}\sigma\alpha_{n-1}, b_{n-1}^o)$. By τ -decreasingness we have $R_{n-1} \vdash \tau(B_{n-1}, b_{n-1}^o) > \tau(B_n, b_n^i)$, so, by stability, $R_{n-1}\sigma\alpha_{n-1} \vdash \tau(B_{n-1}\sigma\alpha_{n-1}, b_{n-1}^o) > \tau(B_n\sigma\alpha_{n-1}, b_n^i)$. By the correctness of the inter-argument relationship, $R_{n-1}\sigma\alpha_{n-1}$ holds, so using the inductive hypothesis, $\tau(A\sigma, a^i) > \tau(B_n\sigma\alpha_{n-1}, b_n^i)$. By the induction argument on $>$, we can assume that $\tau(B_n\sigma\alpha_{n-1}, b_n^i) > \tau(B_n\sigma\alpha_n, b_n^o)$. From this, the result follows.

Since the clause is τ -decreasing, we must have either $R_n \vdash \tau(B_n, b_n^o) \geq \tau(A, a^o)$, so, by stability, $R_n \sigma \alpha_n \vdash \tau(B_n \sigma \alpha_n, b_n^o) \geq \tau(A \sigma \alpha_n, a^o)$. Therefore, since $R_n \sigma \alpha_n$ must hold, $\tau(A \sigma, a^i) > \tau(A \sigma \alpha_n, a^o)$. We can also have $R_n \vdash \tau(A, a^i) > \tau(A, a^o)$, so, by stability, $R_n \sigma \alpha_n \vdash \tau(A \sigma \alpha_n, a^i) > \tau(A \sigma \alpha_n, a^o)$. From this the result follows. \square

5 Orderings for Logic Programs

It follows from Theorem 12 that to show termination we need inference systems that define well-founded stable orderings and allow the use of inter-argument relationships. We now present one such system inspired by [16, 5, 7] which is based on the lexicographic ordering ([8]).

Given a precedence ordering \succ_p on predicate symbols and a term ordering \succ_t (discussed below), we can define the following lexicographic ordering $>$ among literals:

$$\frac{P \approx_p Q \quad \{t_i\}_i (\succ_t)_{lex} \{s_i\}_i}{P(t_1, \dots, t_n) > Q(s_1, \dots, s_m)} \quad \frac{P \succ_p Q}{P(t_1, \dots, t_n) > Q(s_1, \dots, s_m)}$$

In general, the term ordering, \succ_t , must be well-founded and defined by a stable inference system. Moreover, it is essential that the ordering induced by the inter-argument relationships be *consistent* with the term ordering. While different term orderings are possible, in practice, the subterm ordering is sufficient for most programs. The set of inference rules for the subterm ordering is

- Subterm relation

$$\frac{t_i \succeq_t s}{f(\dots, t_i, \dots) \succ_t s} \quad \frac{t \succeq_t g(s_1, \dots, s_m)}{t \succ_t s_i}$$

- Structural rules : transitivity, reflexivity of \succeq_t , irreflexivity of \succ_t , ...

The correctness of these rules is straightforward as is the stability of the ordering. With the appropriate structural rules, this set is complete as well. Other terms orderings, such as the lexicographic path ordering, could easily be adapted to form sets of inference rules such as the above ([7]). It is even possible to define inference rules that use mode information for simplifications, but such inference rules do not seem to help significantly in practice.

Example 2 (Quicksort)

```

split(nil, A, nil, nil).
split(B.X, A, B.L, H) ← B ≤ A, split(X, A, L, H).
split(B.X, A, L, B.H) ← B > A, split(X, A, L, H).
quicksort(nil, nil).
quicksort(A.X, Y) ← split(X, A, L, H), quicksort(L, L1),
                    quicksort(H, H1), append(L1, A.H1, Y).

```

% Mode-dependence :

```

split(list(int), int, ?, ?) → split(list(int), int, list(int), list(int))
quicksort(list(int), ?) → quicksort(list(int), list(int))

```

To show that this program terminates, we use the lexicographic ordering $>$ described earlier, with the following precedence ordering on predicate symbols:

$$\text{quicksort} \succ_p \text{split} \succ_p \text{append} \succ_p \text{“}\leq\text{”} \succ_p \text{“}\gt\text{”}$$

The term ordering \succ_t is simply the subterm relation.

We can show the termination of `split` as follows:

$\{\text{split}(B.X, A, B.L, H)\} > B \leq A, > (B > A)$, and $> \text{split}(X, A, L, H)\}$, since

1. $\text{split} \succ_p \text{“}\leq\text{”}$, and $\text{split} \succ_p \text{“}\gt\text{”}$
2. $\text{split} \approx_p \text{split}$ and $(\square.X, A, \square, \square) (\succ_t)_{lex} (X, A, \square, \square)$.

To show the termination of `quicksort`, we need the inter-argument relationships $\text{split}(X, A, L, H) : X \geq L$ and $\text{split}(X, A, L, H) : X \geq H$, and then we must prove:

$$\begin{array}{lcl} & \text{quicksort}(A.X, \square) & > \text{split}(X, A, \square, \square) \\ X \geq L, X \geq H & \Rightarrow & \text{quicksort}(A.X, \square) > \text{quicksort}(L, \square) \\ X \geq L, X \geq H & \Rightarrow & \text{quicksort}(A.X, \square) > \text{quicksort}(H, \square) \\ X \geq L, X \geq H & \Rightarrow & \text{quicksort}(A.X, \square) > \text{append}(L1, H1, \square) \end{array}$$

The first and fourth case are trivially true. The second and third case can be seen to follow from the inter-argument relationships. For example, in the second case, the following deduction is possible:

$$\frac{\frac{X \geq L}{A.X \succ_t L}}{\text{quicksort} \approx \text{quicksort} \quad \frac{\{A.X, \square\} (\succ_t)_{lex} \{L, \square\}}{\text{quicksort}(A.X, \square) > \text{quicksort}(L, \square)}} \quad \square$$

5.1 Recursive term orderings for literals

When a term ordering is defined recursively using a precedence ordering on function symbols one can, in addition to the precedence, attach a *status* to each function symbol indicating whether its arguments should be compared lexicographically or as a multiset [8]. This notion of recursive ordering is easily adapted to orderings for literals. Moreover, the notion of status can be expanded for predicate symbols. For a predicate symbol, different status (and precedences) can be given for each mode of the predicate, and, furthermore, one can even specify, as part of the status, that only certain arguments of the predicate should be involved in further comparisons. The resulting notion of recursive orderings provides a powerful class of orderings. For example, in some cases, such orderings obviate the need for inter-argument relationships.

Consider terms $t = f(t_1, \dots, t_n)$ and $s = g(s_1, \dots, s_m)$ where the status of f and g are written s_f and s_g respectively. If t and s are compared by a recursively defined ordering $>$, three cases can occur

- the arguments of t are compared with s in accordance with the status of f . This is written $\{t_1, \dots, t_n\}(>)_{\langle s_f, \emptyset \rangle} s$. For example, if s_f indicates that only the second and third arguments should be involved in further comparison, this is equivalent to $t_2 > s \wedge t_3 > s$. (Whether the status indicate multiset or lexicographic does not have an effect in this case.)
- the arguments of s are compared with t . This is written $t (>)_{\langle \emptyset, s_g \rangle} \{s_1, \dots, s_m\}$.
- the arguments of s are compared with the arguments of t . This is written $\{t_1, \dots, t_n\} (>)_{\langle s_f, s_g \rangle} \{s_1, \dots, s_m\}$. For example, if both s_f and s_g specify that a lexicographic order should be used and s_f specifies that only the second and third arguments should be used, then this is equivalent to $\{t_2, t_3\} (>)_{lex} \{s_1, \dots, s_m\}$. Note that s_f and s_g must both indicate a lexicographic order or both indicate a multiset order.

Definition 11 (Recursive Ordering) Assume given a precedence ordering $>_p$ on function symbols and on *moded* predicate symbols (i.e., on predicate symbols with an associated mode) and assume that each function and each moded predicate symbols has associated with it some status. Note that symbols of equal precedence must have the same lexicographic or multiset status.

Besides the usual structural rules (transitivity, ...), the inference rules specific to the recursive ordering are as follows: let $A = p(t_1, \dots, t_n)$ and $B = q(s_1, \dots, s_m)$

$$\frac{p >_p q \quad A (>)_{\langle \emptyset, s_q \rangle} \{s_1, \dots, s_m\}}{A > B} \quad \frac{\{t_1, \dots, t_n\} (>)_{\langle s_p, \emptyset \rangle} B}{A > B}$$

$$\frac{p \approx_p q \quad \{t_1, \dots, t_n\} (>)_{\langle s_p, s_q \rangle} \{s_1, \dots, s_m\} \quad A (>)_{\langle \emptyset, s_q \rangle} \{s_1, \dots, s_m\}}{A > B}$$

Proposition 14 The recursive ordering is a well-founded order.

Proof:

- Transitivity: Let $A = p(t_1, \dots, t_n)$, $B = q(s_1, \dots, s_m)$, and $C = o(r_1, \dots, r_l)$. Assume $A > B$ and $B > C$. We want to show $A > C$. We proceed by induction in the size of the terms A, B, C . According to the definition of recursive orderings, the possible cases are
 - $\{t_1, \dots, t_n\} (>)_{\langle s_p, \emptyset \rangle} B$, i.e., some selected subterms of A is greater than B . Hence, by the inductive hypothesis, this subterm is greater than C , so A is greater than C . The same reasoning applies if $\{s_1, \dots, s_m\} (>)_{s_q} C$.
 - $p >_p q \geq_p o$, in this case we must have $B (>)_{\langle \emptyset, s_o \rangle} \{r_1, \dots, r_l\}$, so, by the inductive hypothesis, $A (>)_{\langle \emptyset, s_o \rangle} \{r_1, \dots, r_l\}$. Hence, $A > C$. The same reasoning applies if $p \geq_p q >_p o$.
 - $p \approx_p q \approx_p o$. By the same reasoning as above, we can show $A (>)_{\langle \emptyset, s_o \rangle} \{r_1, \dots, r_l\}$. Moreover, in this case $A > B > C$ implies $\{t_1, \dots, t_n\} (>)_{\langle s_p, s_q \rangle} \{s_1, \dots, s_m\} (>)_{\langle s_q, s_o \rangle} \{r_1, \dots, r_l\}$. Hence, $\{t_1, \dots, t_n\} (>)_{\langle s_p, s_o \rangle} \{r_1, \dots, r_l\}$ by the transitivity of the multiset or lexicographic ordering, accordingly.

- Well-foundedness : Suppose, there is an infinite descending sequence of literals. Consider an infinite subsequence where all the literals use with the same predicate symbol. By transitivity, this is an infinite descending sequence. By definition of the recursive ordering, the multisets of arguments from these literals form an infinite descending sequence of multisets of fixed size of *terms* (since a literal cannot be a subterm of another literal). But this contradicts the well-foundedness of the recursive ordering for terms ([8]).

Example 3 ([10]) Consider the following program computing the transitive closure of a predicate p :

```

p(a, b).
p(b, c).
tc(X, X).
tc(X, Y) ← p(X, Z), tc(Z, Y).

```

% Mode-dependencies :

```

p(ground, ?) → p(ground, ground)
tc(ground, ?) → tc(ground, ground)

```

This program can be shown to terminate *without inter-argument relationships* by using the recursive ordering defined above with the precedence ordering: $a \succ_p b \succ_p c \succ_p p : p(\text{ground}, \text{ground}) \succ_p tc : tc(\text{ground}, ?) \succ_p tc : tc(\text{ground}, \text{ground}) \succ_p p : p(\text{ground}, ?)$, and status information specifying that $p : p(\text{ground}, ?)$ and $tc : tc(\text{ground}, ?)$ consider only their first argument for further comparisons and $p : p(\text{ground}, \text{ground})$ and $tc : tc(\text{ground}, \text{ground})$ consider only their second argument for further comparisons. Then the clauses defining the program can be shown to be τ -decreasing as follows:

- For the first clause, we must show

$$p(a, b) : p(\text{ground}, ?) > p(a, b) : p(\text{ground}, \text{ground})$$

which follows from $a \succ_p p : p(\text{ground}, \text{ground})$ and $a \succ_p b$.

- The second and third clause are similarly shown to be τ -decreasing.
- For the fourth clause, we must show

$$\begin{aligned}
&tc(X, Y) : tc(\text{ground}, ?) > p(X, Z) : p(\text{ground}, ?) \\
&p(X, Z) : p(\text{ground}, \text{ground}) \geq tc(Z, Y) : tc(\text{ground}, ?) \text{ and} \\
&tc(Z, Y) : tc(\text{ground}, \text{ground}) \geq tc(X, Y) : tc(\text{ground}, \text{ground})
\end{aligned}$$

All these can be shown by our recursive ordering. Therefore the program terminates.

6 Inductive Reasoning and Symbolic Evaluation

6.1 Deriving inter-argument monotonicity relationships

Inter-argument monotonicity relationships can be derived as *partial correctness properties* by assuming that they hold for the recursive call to predicates (or of the goal) and using inference

systems for orderings, like the ones discussed earlier, to test that they hold for the original predicates. This is the classical Hoare’s technique to prove partial correctness properties; moreover, if the relevant literals are terminating then the inter-arguments relationships become *total correctness* properties and this technique can be seen as a special case of Noetherian induction.

Example 4 Given the usual definition of `append` (with both modes: `append(?, ?, list(?)) → append(list(?), list(?), list(?))` and `append(list(?), list(?), ?) → append(list(?), list(?), list(?))`), suppose we want to prove the inter-argument relationship:

$$\text{append}(L_1, A.L_2, Y), \text{append}(L_1, L_2, Z) : Y > Z$$

We have to consider two cases: $L_1 = []$ and $L_1 = [B|L']$. In the first case, after simplification, we are left to prove $A.L_2 > L_2$ which follows by the subterm property of $>$. In the second case, the result of the symbolic evaluation is:

$$\text{append}(L', A.L_2, Y'), \text{append}(L', L_2, Z') : B.Y' > B.Z'$$

By the assumption that the inter-argument relationship holds for the recursive call of the goal, we have $Y' > Z'$. It is now trivial to show that $Y' > Z' \Rightarrow B.Y' > B.Z'$.

Moreover, having proved that `append` in the above given modes terminates, we can conclude that this inter-argument relationship is a total correctness property. Using this, we could, for example, easily prove the termination of the permutation program ([17]):

```
perm(nil, nil).
perm(A.X, Y) ← append(L1, A.L2, Y), append(L1, L2, Z), perm(X, Z).
% Mode-dependency :
perm(? , list(?)) → perm(list(?), list(?))
```

6.2 Mutually recursive predicates

For mutually recursive predicate definitions a formal inductive argument is sometimes necessary to prove termination. The technique adopted is similar to the one advocated above for deriving inter-argument relationships. It uses *symbolic evaluation* with the termination of the predicates as the inductive hypotheses and syntactic orderings to justify the use of the inductive hypotheses. Moreover, once the termination of some literals is established (e.g. by the inductive hypothesis), we can use the relevant inter-argument relationships as a total correctness property to propagate ordering information, henceforth, allowing more uses of the inductive hypotheses. The following example (from [17, 26]) illustrates these issues.

Example 5 (Parser) The following rules specify an arithmetic expression parser. The mode-dependence is $e(\text{list}(\text{ground}), ?) \rightarrow e(\text{list}(\text{ground}), \text{list}(\text{ground}))$ The first argument to `e` is the

ground input list to be parsed and the second argument is the unparsed suffix.

$$\begin{aligned} e(L, T) & : - \quad t(L, '+' . C), e(C, T). \\ e(L, T) & : - \quad t(L, T). \end{aligned}$$

$$\begin{aligned} t(L, T) & : - \quad n(L, '*' . C), t(C, T). \\ t(L, T) & : - \quad n(L, T). \end{aligned}$$

$$\begin{aligned} n(' . A, T) & : - \quad e(A, ') . T). \\ n(L . T, T) & : - \quad \text{constant-or-identifier}(L). \end{aligned}$$

The technique described in subsection 6.1 can be used to obtain the inter-argument relationships:

$$e(L, T) : L > T, \quad t(L, T) : L > T, \quad n(L, T) : L > T$$

To prove the termination of this program we prove by induction the following property:

Any goal of the form $e(t_1, t_2)$ or $t(t_1, t_2)$ or $n(t_1, t_2)$ terminates.

We use the simple lexicographic ordering on literals defined earlier. The precedence ordering on predicates is $e \approx_p t \approx_p n \succ_p \text{constant-or-identifier}$.

Symbolic evaluation of the possible goals $\{e(L, T), t(L, T), n(L, T)\}$ generates a large number of cases like $e(' . A, T), e(L . '*' . C, T), t(' . '*' . C, T), \dots$. Let us consider, for example, the case $t(' . A, T)$ whose symbolic evaluation produces:

$$t(' . A, T) \Leftrightarrow \begin{aligned} & e(A, ') . '*' . C), t(C, T) \\ & \vee e(A, ') . T). \end{aligned}$$

It is possible to use the inductive hypothesis since

$$\begin{aligned} & t(' . A, T) \succ e(A, ') . '*' . C) \\ \text{and} \quad & t(' . A, T) \succ e(A, ') . T) \end{aligned}$$

Hence, by the inductive hypothesis, $e(A, ') . '*' . C)$ and $e(A, ') . '*' . T)$ terminate. Moreover, the inter-argument relationship holds, i.e $A > ') . '*' . C$, so $A > C$ and $) . A > C$. Hence

$$t(' . A, T) \succ t(C, T).$$

This allows the use of the inductive hypothesis again and we conclude that $t(C, T)$ terminates. All the other cases can be treated similarly, so the program universally terminates.

Notice, however, that we could show the termination of this program in a more straightforward way, without inter-argument relationships, if we use the recursive ordering discussed earlier with the precedence ordering “.” $\succ_p n : n(\text{list}(\text{ground}), \text{list}(\text{ground})) \succ_p t : t(\text{list}(\text{ground}), \text{list}(\text{ground})) \succ_p e : e(\text{list}(\text{ground}), \text{list}(\text{ground})) \succ_p e : e(\text{list}(\text{ground}), ?) \succ_p t : t(\text{list}(\text{ground}), ?) \succ_p n : n(\text{list}(\text{ground}), ?)$ and the status: $e : e(\text{list}(\text{ground}), ?)$ consider only its first arguments while $e : e(\text{list}(\text{ground}), \text{list}(\text{ground}))$ consider only its second arguments, and similarly for t and n .

7 Extensions

The techniques for showing termination presented here can be adapted to more powerful notions of mode-annotations. We can indeed extend mode-annotations to regular trees as described in [19]. An extension is in fact necessary to handle partially instantiated structures which none of the existing approaches to termination are able to handle. Consider the following motivating example: a tail-recursive program to flatten an input list by one level.

Example 6 (Tail-recursive one level Flatten)

```

flatten(nil, nil).
flatten(A.X, Y) ← mlist(A), append(A, Z, Y), flatten(X, Z).
flatten(A.X, Y) ← constant(A), append([A], Z, Y), flatten(X, Z).
constant(X) ← atom(X), integer(X).
mlist([A|X]).

```

% Mode-dependencies :

```

flatten(list(ground + list(?)), ?) → flatten(list(ground + list(?)), list(?))
constant(?) → constant(ground)
mlist(?) → mlist(list(?))

```

With the definition of mode-annotations given in section 2 we are not able to assign a mode-dependency for `append` such that the program is well-moded. If we assign the following mode-dependency to `append`

$$\text{append}(\text{list}(\?), \?, \?) \rightarrow \text{append}(\text{list}(\?), \text{list}(\?), \text{list}(\?))$$

then the first clause defining `append` is not well-moded. However, we clearly need the mode-dependency $\text{append}(\text{list}(\?), \?, \?) \rightarrow m^o$ if `flatten` is to be well-moded. For m^o all that is really necessary is that when Z is in the mode $\text{list}(\?)$ then Y is also in that mode. Informally this suggests a mode-dependency of the form

$$\exists \alpha, \beta (\text{append}(\text{list}(\?), \?, \?) \rightarrow \text{append}(\text{list}(\?), \alpha, \beta) \ \& \ \alpha = \text{list}(\?) \Rightarrow \beta = \text{list}(\?)).$$

After the subsequent execution of the (recursive) call to `flatten`, since Z is in the output mode of `flatten`, Z is instantiated to a term in the mode $\text{list}(\?)$. Hence, we can use the implication in the mode-dependency of `append` to conclude that Y must also be in mode $\text{list}(\?)$. This concludes proving the well-modedness of the program. The termination proof is then straightforward. \square

Thus, to define well-modedness of programs such as `flatten` above, we need to extend the syntax of mode-annotations by allowing a -terms containing existentially quantified mode variables and mode-dependency defined by logical implication. The extended mode-annotations are similar to regular trees¹³ as described in [19].

¹³We note in passing that the alternative prescriptive approach to types as defined in [13] is not able to handle partially instantiated arguments.

Mode-dependency for predicates can also be used to specify control in concurrent logic programs. For instance, a consumer process may have to suspend until an argument gets sufficiently instantiated by a producer process to unify with the head of a clause. Mode-annotations can be used to specify the exact structure of the arguments that would allow the consumer process to resume. Furthermore, in parallel evaluation of logic programs, the granularity of subtasks (i.e., if a subtask warrants a sub-computation large enough to outweigh the cost of forking) can be specified via the richer notion of modes.

8 Conclusions

This paper presents a rich framework of directionality for logic programs and its use in a proof method for proving universal termination of logic programs. Notions of mode-annotations (defined via a combination of modes and types) and mode-dependencies are used to give a directionality to logic programs. Well-founded orderings on the atoms in a clause are then used to show the finiteness of LD-derivations. Due to the generality of mode-annotations, the termination methodology can handle programs and goals with non-ground inputs and non-ground outputs.

We believe that the richness of mode-annotations can also be used in applications other than termination proofs, such as specifying control for concurrent logic languages.

References

- [1] K. R. Apt, R. N. Bol, and J. W. Klop. On the safe termination of Prolog programs. In G. Levi and M. Martelli, editors, *Proc. Intl. Conf. on Logic Programming*, volume 6, Pisa, 1989.
- [2] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. Technical Report CS-R9048, Centrum voor Wiskunde en Informatica, September 1990.
- [3] M. Baudinet. Proving termination properties of Prolog programs : A semantic approach. *Proc. Symp. LICS*, 3:336 – 348, 1988.
- [4] M. Bezem. Characterising Termination of logic programs with level mappings. In E. L. Lusk and R. A. Overbeek, editors, *Proc. North American Conf. on Logic Programming*, volume 1, pages 60 – 89. MIT Press, 1989.
- [5] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *ACM Symp. on Principles of Database Systems*, volume 10. ACM Press, 1991.
- [6] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *First Intl. Logic Programming Conf.*, pages 129–133, 1982.
- [7] H. Comon. Solving inequations in term algebras. In *LICS*, pages 62–69. IEEE, June 1990.
- [8] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69 – 116, 1987.
- [9] Y. Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming. Addison Wesley, 1990.
- [10] H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In M. Rusinowitch and J.L. Remy, editors, *CTRS*, pages 216–222, July 1992.

- [11] A. Van Gelder. Deriving constraints among argument sizes in logic programs. In *ACM Symp. on Principles of Database Systems*, volume 9, pages 47 – 60. SIGACT-SIGMOD-SIGART, ACM Press, 1990.
- [12] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Programming*, 13(2):205–258, July 1992.
- [13] T. K. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O’Keefe type system. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 202 – 217, Cambridge, Mass., 1991. MIT Press.
- [14] P. Mishra. Towards a theory of types in Prolog. In *IEEE International symposium on logic programming*, pages 289 – 298, 1984.
- [15] P. Mishra and U. S. Reddy. Declaration-free type checking. In *ACM Symp. on Principles of Programming Languages*, pages 7 – 21, 1985.
- [16] Gerald Peterson. Solving term inequalities. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 258–263, Boston, MA, July 1990.
- [17] L. Plumer. Termination proofs for logic programs based on predicate inequalities. In D.H.D. Warren and P. Szeredi, editors, *Proc. Intl. Conf. on Logic Programming*, volume 7, pages 634 – 648. MIT Press, 1990.
- [18] L. Plumer. Automatic termination proofs for prolog programs operating on non-ground terms. In *Proc. Intl. Logic Programming Symp.* MIT Press, 1991.
- [19] C. Pyo and U. S. Reddy. Inference of polymorphic types for logic programs. In E. L. Lusk and R.A. Overbeek, editor, *Logic Programming: Proceedings of the North American Conf.*, pages 1115 – 1134. MIT Press, 1989.
- [20] M.R.K. Krishna Rao, Deepak Kapur, and R.K. Shyamasundar. A transformational methodology for proving termination of logic programs. In *5th Conference on Computer Science Logic*. LNCS, 1991. (to appear).
- [21] U. S. Reddy. Transformation of logic programs into functional programs. In *Proc. Intl. Symp. on Logic Programming*, pages 187 – 196, New Jersey, USA, 1984. IEEE.
- [22] U. S. Reddy. Notions of polymorphism for predicate logic programs. In *Proc. Intl. Conf. on Logic Programming*, volume 5, 1988.
- [23] U. S. Reddy. A typed foundation for directional logic programming. In *Proc. Workshop on Extensions to Logic Programming, University of Bologna, Italy*, pages 199 – 222, 1992.
- [24] B. J. Ross. Using algebraic semantics for proving Prolog termination and transformation. In *Proc. Assoc. Logic Programming, UK*, 1991.
- [25] Y. Sagiv. A termination test for logic programs. In *Proc. Intl. Logic Programming Symp.* MIT Press, 1991.
- [26] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *ACM Symp. on Principles of Database Systems*, pages 216 – 226, 1991.
- [27] T. Vassak and J. Potter. Characterization of terminating logic programs. In *IEEE Symposium on Logic Programming*, Vancouver, 1986.
- [28] K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In *Proceedings ICLP’91*, pages 301–315, Paris, June 1991. MIT Press.

- [29] B. Wang and R. K. Shyamasundar. Towards a characterization of termination of logic programs. In *ACM Conference on Programming Languages Implementation and Logic programming*, 1990.

Appendix A: Mode Inference Rules

Mode judgements of the form $\Gamma \vdash t : a$ are derived using the the following mode inference rules:

Introduction Rules

In an mode context Γ , mode judgements $\Gamma \vdash t : a$ are inferred using the the following introduction rules:

$$\begin{array}{ll}
 \Gamma \vdash X : a & \text{if } (X : a) \in \Gamma \\
 \Gamma \vdash X : ? & \text{if } (X \notin \text{dom}(\Gamma)) \\
 \Gamma \vdash c : \textit{ground} & \text{if } c \text{ is a constant} \\
 \frac{\Gamma \vdash t_1 : a_1 \quad \dots \quad \Gamma \vdash t_k : a_k}{\Gamma \vdash F(t_1, \dots, t_k) : F(a_1, \dots, a_k)} & \text{where } F \text{ is a function or predicate symbol} \\
 \frac{\Gamma \vdash t : a_1}{\Gamma \vdash t : a_1 + a_2} & \\
 \frac{\Gamma \vdash t : a[\mathbf{fix}\alpha.a/\alpha]}{\Gamma \vdash t : \mathbf{fix}\alpha.a} &
 \end{array}$$

Derived Rules

The following rule expresses inclusion relationship between well-moded terms.

$$\frac{\Gamma \vdash t : a}{\Gamma \vdash t : b} \quad \text{if } b \succeq a$$

Elimination Rules

The mode-context Γ generated by a sequence of well-moded literals (each in its output mode) is determined using the introduction rules backward and the following elimination rule:

$$\frac{\Gamma \vdash t : a_1 + a_2 \quad \Gamma \cup \{t : a_1\} \vdash A \quad \Gamma \cup \{t : a_2\} \vdash A}{\Gamma \vdash A}$$

Notice that because of the implicit non-determinism involved in the backward use of the inclusion rule, in practice we might fail to effectively generate a mode-context Γ .

The soundness of these rules is shown as follows.

Proposition 15

If $\Gamma \vdash t : a$ then for all ζ which respect Γ , $\zeta(t) \in \llbracket a \rrbracket \eta$ where η is any mode valuation.

Proof: By induction on the length of the derivation of $\Gamma \vdash t : a$.

- $\Gamma \vdash X : a$ if $(X : a) \in \Gamma$. For all ζ which respects Γ , $\zeta(X) \in \llbracket \Gamma(X) \rrbracket \eta$; i.e., $\zeta(X) \in \llbracket a \rrbracket \eta$.
- $\Gamma \vdash X : ?$ if $(X : a) \notin \Gamma$. For all ζ which respects Γ , $\zeta(X) \in \llbracket ? \rrbracket \eta$ which is true for all ζ .
- $\Gamma \vdash c : \text{ground}$. For all ζ which respects Γ , $\zeta(c) = c \in \llbracket \text{ground} \rrbracket \eta$ which is true for all ζ .
- $\Gamma \vdash f(t) : f(a)$. For all ζ which respects Γ , $\zeta(f(t)) = f(\zeta(t))$. By induction hypothesis, $\zeta(t) \in \llbracket a \rrbracket \eta$. Hence, $f(\zeta(t)) \in f(\llbracket a \rrbracket \eta)$ i.e., $\zeta(f(t)) \in \llbracket f(a) \rrbracket \eta$.
- $\Gamma \vdash t : a_1 + a_2$. By induction hypothesis, for all ζ which respects Γ , $\zeta(t) \in \llbracket a_1 \rrbracket \eta$. Now, since $\llbracket a_1 \rrbracket \eta \subseteq \llbracket a_1 + a_2 \rrbracket \eta$, hence $\zeta(t) \in \llbracket a_1 + a_2 \rrbracket \eta$.
- $\Gamma \vdash t : \mathbf{fix}\alpha.a$. By induction hypothesis, for all ζ which respect Γ , $\zeta(t) \in \llbracket a[\mathbf{fix}\alpha.a / \alpha] \rrbracket \eta$. Hence, $\zeta(t) \in \llbracket a \rrbracket (\eta[\mathbf{fix}\alpha.a / \alpha])$. Hence, $\zeta(t) \in$ least S such that $S = \llbracket a \rrbracket (\eta[S/\alpha])$. By the definition of \mathbf{fix} , this means that $\zeta(t) \in \llbracket \mathbf{fix}\alpha.a \rrbracket \eta$.
- $\Gamma \vdash t : a_2$ (using the inclusion rule). By the induction hypothesis, for all ζ which respect Γ , $\zeta(t) \in \llbracket a_1 \rrbracket \eta$. Hence, $\zeta(t) \in \llbracket a_2 \rrbracket \eta$ (since $a_1 \preceq a_2 \Rightarrow \llbracket a_1 \rrbracket \eta \subseteq \llbracket a_2 \rrbracket \eta$ by proposition 16).

□

Appendix B: Inclusion Checking Rules

The following inference rules (which have been adapted from [15]) are used to infer inclusion (covering) of mode-annotations: $a_1 \preceq a_2$.

$$\begin{array}{c}
 a \preceq a \qquad \textit{reflexivity} \\
 \\
 \frac{a_1 \preceq a_2, \quad a_2 \preceq a_3}{a_1 \preceq a_3} \qquad \textit{transitivity} \\
 \\
 a \preceq ? \\
 \\
 \frac{a_1 \preceq \textit{ground}, \dots, a_n \preceq \textit{ground}}{f(a_1, \dots, a_n) \preceq \textit{ground}} \\
 \\
 \frac{a_1 \preceq b_1, \dots, a_n \preceq b_n}{f(a_1, \dots, a_n) \preceq f(b_1, \dots, b_n)} \\
 \\
 \frac{a_1 \preceq a, \quad a_2 \preceq a}{a_1 + a_2 \preceq a} \\
 \\
 \frac{a \preceq a_1}{a \preceq a_1 + a_2} \\
 \\
 \frac{a[\mathbf{fix}\alpha.a/\alpha] \preceq a_2}{\mathbf{fix}\alpha.a \preceq a_2} \\
 \\
 \frac{a_1 \preceq a[\mathbf{fix}\alpha.a/\alpha]}{a_1 \preceq \mathbf{fix}\alpha.a}
 \end{array}$$

The above rules can be transformed into an algorithm for checking inclusion $a_1 \preceq a_2$. Further note that the inference system is deterministic but for the rule for $+$ in a_2 and the unfolding of the \mathbf{fix} . The rule for $+$ can be made deterministic by requiring that a_2 be discriminative.

The soundness of these rules is shown below.

Proposition 16 If $a_1 \preceq a_2$, then $\llbracket a_1 \rrbracket \eta \subseteq \llbracket a_2 \rrbracket \eta$ where η is any mode valuation.

Proof: By induction on the length of the derivation of $a_1 \preceq a_2$.

- $a \preceq ?$. Since $\llbracket ? \rrbracket \eta = \mathcal{T}$, hence $\llbracket a_1 \rrbracket \eta \subseteq \llbracket ? \rrbracket \eta$
- If the last rule used to derive $a_1 \preceq a_2$ is

$$\frac{a_1 \preceq \textit{ground}, \dots, a_n \preceq \textit{ground}}{f(a_1, \dots, a_n) \preceq \textit{ground}}$$

By the induction hypothesis, we can assume that $\llbracket a \rrbracket \eta \subseteq \llbracket \textit{ground} \rrbracket \eta$.

Now, $\llbracket f(a) \rrbracket \eta = \{f(t) \mid t \in \llbracket a \rrbracket \eta\} \subseteq \{f(t) \mid t \in \llbracket \textit{ground} \rrbracket \eta\} \subseteq \llbracket \textit{ground} \rrbracket \eta$

- If the last rule used to derive $a_1 \preceq a_2$ is

$$\frac{a_1 \preceq b_1, \dots, a_n \preceq b_n}{f(a_1, \dots, a_n) \preceq f(b_1, \dots, b_n)}$$

By the induction hypothesis, we can assume that $\llbracket a_1 \rrbracket \eta \subseteq \llbracket b_1 \rrbracket \eta, \dots, \llbracket a_n \rrbracket \eta \subseteq \llbracket b_n \rrbracket \eta$.
 Now, $\llbracket f(a_1, \dots, a_n) \rrbracket \eta = \{f(t_1, \dots, t_n) \mid t_1 \in \llbracket a_1 \rrbracket \eta, \dots, t_n \in \llbracket a_n \rrbracket \eta\}$
 $\subseteq \{f(t_1, \dots, t_n) \mid t_1 \in \llbracket b_1 \rrbracket \eta, \dots, t_n \in \llbracket b_n \rrbracket \eta\} \subseteq \llbracket f(b_1, \dots, b_n) \rrbracket \eta$

- If the last rule used to derive $a_1 \preceq a_2$ is

$$\frac{a_1 \preceq a, \quad a_2 \preceq a}{a_1 + a_2 \preceq a}$$

By the induction hypothesis, we can assume that $\llbracket a_1 \rrbracket \eta \subseteq \llbracket a \rrbracket \eta$ and $\llbracket a_2 \rrbracket \eta \subseteq \llbracket a \rrbracket \eta$.
 Now, $\llbracket a_1 + a_2 \rrbracket \eta = \llbracket a_1 \rrbracket \eta \cup \llbracket a_2 \rrbracket \eta \subseteq \llbracket a \rrbracket \eta$

- If the last rule used to derive $a_1 \preceq a_2$ is

$$\frac{a \preceq a_1}{a \preceq a_1 + a_2}$$

By the induction hypothesis, we can assume that $\llbracket a \rrbracket \eta \subseteq \llbracket a_1 \rrbracket \eta \subseteq \llbracket a_1 + a_2 \rrbracket \eta$

- If the last rule used to derive $a_1 \preceq a_2$ is

$$\frac{a[\mathbf{fix}\alpha.a/\alpha] \preceq a_2}{\mathbf{fix}\alpha.a \preceq a_2}$$

By the induction hypothesis, we can assume that $\llbracket a[\mathbf{fix}\alpha.a/\alpha] \rrbracket \eta \subseteq \llbracket a_2 \rrbracket \eta$. Hence,
 $\llbracket \mathbf{fix}\alpha.a \rrbracket \eta \subseteq \llbracket a_2 \rrbracket \eta$ (by the semantics of \mathbf{fix}).

- If the last rule used to derive $a_1 \preceq a_2$ is

$$\frac{a_1 \preceq a[\mathbf{fix}\alpha.a/\alpha]}{a_1 \preceq \mathbf{fix}\alpha.a}$$

By the induction hypothesis we can assume that $\llbracket a_1 \rrbracket \eta \subseteq \llbracket a[\mathbf{fix}\alpha.a/\alpha] \rrbracket \eta$. Hence
 $\llbracket a_1 \rrbracket \eta \subseteq \llbracket \mathbf{fix}\alpha.a \rrbracket \eta$ (by the semantics of \mathbf{fix}).

□