Relational Groundness Analysis of Logic Programs

(Extended Abstract)

Kirack Sohn [*]
University of California, Santa Cruz
ksohn@cse.ucsc.edu

March 23, 1994

Abstract

One of the most attractive features of logic programs is that arguments may be used bidirectionally, as input or output at run-time. However, the compiler must generate code for various alternatives, hence considerable slowdown of execution. In this paper we provide static analysis for *groundness*; that is, inference of whether the arguments are instantiated to ground terms through run-time.

Though groundness analysis has been studied in numerous papers, we present yet another method to resolve globally the problems due to aliased variables based on *relational abstract lattices*.

Relational groundness analysis has been considered to be impractical since tables representing the relationship are usually very big, hence hardness to handle. In our formalism, the relationships are represented in the form of simple computable boolean OR constraints.

Since in practice groundness constraints are almost always in simple computable forms, our method is considered to run without combinatorial blowup. Our abstract domain is exemplified with bottom-up abstract interpretation finding success patterns of predicates.

1 Introduction

Logic programming has been successfully used as a tool for several areas including compiler writing, expert system design, natural language processing, hardware design, and knowledge-base design. One of the most attractive features of Prolog is bidirectional use of arguments for input, output, or both; however, in practical programs most procedures do not make this sophisticated use of arguments. Using mode information allows compilers to produce more specific code which results in substantial speedup [Mel85]. With this regard, Warren introduced explicit mode declarations to help the compiler produce better code [War77]. But mode declarations must be verified since wrong annotation may introduce subtle errors in program executions.

Another approach is to infer mode declarations automatically via *abstract interpretation* [Mel87, BJCB87, MU87, DW88]. Abstract interpretation has been used as standard means for dataflow analysis since it was placed on a solid semantic basis by Cousot and Cousot [CC77] (See [CC92] for the theory and applications to logic programming).

[*]Work partially supported by NSF grants CCR-89-58590 and IRI-9102513

Early work done by Mellish [Mel81] produced erroneous results since aliasing effects resulting from unification was not considered, which was corrected later [Mel87].

Mannila and Ukkonen [MU87] used simple two-valued abstract domains $\{ground, any\}$, hence $free$ mode [1] cannot be inferred ($ground, free, any$ are abstractions of ground terms, free variables, all (ground or nonground) terms, respectively). In addition, their method could not handle the problem with aliased variables accurately.

Bruynooghe *et al.* [BJCB87] suggested multi-passes algorithm (repeat previous call strategy) to resolve aliasing problem, which are considered to be costly if the strategy is applied globally.

Debray and Warren [DW88] presented an algorithm to give a sound and efficient treatment of aliasing. However their method produces less precise mode information since they use conservative local analysis in which all unsafe instantiations are replaced by any.

In this paper we study relational abstract domains which describe possible groundness relationships among arguments. Like [MU87], we only analyze groundness, however, our method provides great accuracy with respect to groundness. It is also noted that success of correctness proofs like program termination relies on preciseness of groundness information [UVG88, Plü90, SVG91].

Let us consider a call `p(A, f(A,B))` in which we have no instantiation information on `A` and `B`. Hence the goal is abstracted as $p(any, any)$ in the previous cited methods. But more careful look at the call gives us the possible combinations of call patterns $\{p(0,0), p(0,1), p(1,1)\}$ where 0 and 1 are $ground$ and $nonground$ respectively to keep notations simple. That is, if the second argument is $ground$, then the first has no other choice but $ground$. Suppose we have a clause `p(U, f(a,b)).` With the call pattern $p(any, any)$, the success pattern after the call to the clause is $p(any, ground)$ whereas with our call pattern, the success pattern is $p(ground, ground)$. Preciseness of success patterns affect the subsequent call patterns immediately. It should be mentioned that if we use a table of possible groundness relationships among arguments as shown above, we will end up with combinatorial blowup.

We now introduce simple boolean constraints to denote groundness relation. Roughly, a term can be abstracted as a logical disjunction of variables occurring in the term. For example, $p(a, a \vee b))$ is the abstraction of the goal `p(A, f(A,B))` where a and b are boolean variables corresponding to `A` and `B` and $\vee$ is boolean OR.

Main contribution of our study is to introduce novel concepts on boolean cones and develop efficient operations to accomplish effectively relational groundness analysis and to provide bottom-up groundness analysis using boolean constraints.

In Section 2, we describe notations used through the paper and introduce groundness abstraction of logic programs, and concepts, properties, and operations on boolean cones.

Section 3 describes some simplification procedures to remove redundancy in groundness constraints.

Section 4 presents bottom-up groundness analysis mimicking immediate consequence operator.

Section 5 concludes the paper.

The framework similar to our relational bottom-up analysis has been investigated on real arithmetic domain to derive constraints among argument sizes by Van Gelder [VG91].

Throughout the paper, we will use standard Edinburgh style syntax for logic programs [CM81].

[1] In their paper, *nonground* means possibly nonground, which means *any*

2 Basic Concepts

2.1 Abstraction

Now we describe abstraction of terms, atoms, and clauses. Logic programs can be abstracted by groundness relationships among terms. Informally speaking, groundness abstraction of a term carries the information that if all the variables in the term are ground, then the term is ground. We choose the boolean value 0 to denote *ground*. So $x \vee y$ is the groundness abstraction of `f(X,Y)`. That is, if x (`X`) and y (`Y`) are 0 (*ground*), then $x \vee y$ (`f(X,Y)`) is 0 (*ground*). Let $vars(t)$ denote a set of variables in a term t. We shall use lowercase letters for the boolean variables corresponding to logical variables.

For notational convenience, we shall use p_i for a boolean variable representing groundness abstraction of i-th argument of predicate p. Boolean terms are terms built from boolean variables and connective boolean OR $\vee$. Note that we do not use NOT and AND in boolean terms. Boolean equalities are built from boolean terms and boolean equality $\approx$. We often use vector notation to denote a vector of boolean terms. For example, $(a_1, a_2) \vee (b_1, b_2) \approx (c_1, c_2)$ denotes: $a_1 \vee b_1 \approx c_1$ and $a_2 \vee b_2 \approx c_2$. Substitution $[x_1/e_1, \ldots, x_n/e_n]$ is defined in an usual way; that is, a boolean variable x_i is replaced by a boolean term e_i. Using vector notation, it is denoted by $[\vec{x}/\vec{e}]$.

Definition 2.1: Let t be a logical term. Then $\alpha(t)$ is a groundness-abstracted term.

$$\alpha(t) = \begin{cases} 0 & \text{if } vars(t) \text{ is empty} \\ \vee_{x \in vars(t)} x & \text{otherwise} \end{cases}$$

Let p be n-ary predicate. Then groundness-abstracted atom is as follows.

$$\alpha(p(t_1, t_2, \ldots, t_n)) = p(\alpha(t_1), \alpha(t_2), \ldots, \alpha(t_n))$$

Abstraction of clauses is obtained by applying abstraction to each atom occurring in the clauses. □

Example 2.1: Let us consider the usual `append` procedure.

```
a([], U, U).
a([X|U], V, [X|W]) :- a(U, V, W).
```

Applying groundness abstraction to each clause transforms it to groundness-abstracted procedure.

$a(0, u, u)$.
$a(x \vee u, v, x \vee w) \leftarrow a(u, v, w)$.

Since terms are boolean expressions, unification must be replaced by boolean constraint solving in the execution of the groundness-abstracted procedure. □

2.2 Relational Abstract Domains

We now introduce boolean cones and their constraint representation. We also examine some operations like closure of union as least upper bound operation and equivalence of two boolean cones, which are useful in finding the fixpoint of a certain transformation concerning groundness relation.

Definition 2.2: Let $a, b \in B^n$. $C \subseteq B^n$ is a *boolean cone* if

1. $\mathbf{0} \in C$
2. If $a \in C$ and $b \in C$, then $a \vee b \in C$

$\square$

Example 2.1: $\{(0,0,0),(1,1,0),(1,0,1),(1,1,1)\}$ is a boolean cone whereas $\{(0,0,0),(1,1,0),(1,0,1)\}$ is not. $\square$

We use the word "boolean cones" since they have the properties similar to those of convex cones in R^n. We now examine some useful properties of boolean cones.

Lemma 2.1: Intersection of two boolean cones C_1 and C_2 is a boolean cone C_3. $\square$

Example 2.2: Let $C_1 = \{(0,0,0),(0,1,1),(1,0,1),(1,1,1)\}$ and $C_2 = \{(0,0,0),(1,1,0),(1,1,1)\}$. C_1 and C_2 are boolean cones. $C_1 \bigcap C_2 = \{(0,0,0),(1,1,1)\}$ is also a boolean cone. $\square$

Lemma 2.2: The projection of boolean cone in $\{0,1\}^n$ onto $\{0,1\}^m$ where $m \leq n$ is also a boolean cone. $\square$

We now introduce the concept similar to extreme rays of convex cones in R^n, which we call *generators*. Generator sets serve as unique, minimal representation of boolean cones. Therefore testing equivalence of two cones reduces to comparing two generator sets. Boolean sum of two vectors is a vector of componentwise sums.

Definition 2.3: Let C be a boolean cone C, and $\alpha \in C$. Then α is a *generator* of C if $\alpha \neq \mathbf{0}$ and cannot be the boolean sum of any other points in C. A *generator set* $gen(C)$ of C is a set of all generators in C.

Example 2.3: $gen(\{(0,0,0),(0,1,1),(1,0,1),(1,1,1)\} = \{(0,1,1),(1,0,1)\}$ $\square$

Lemma 2.3: There exists a unique generator set $gen(C)$ for any boolean cone C. $\square$

Let $\mathcal{U}$ be the set of all boolean cones in $\{0,1\}^n$. Equipped with the subset ordering $\subseteq$, $\mathcal{U}$ forms a complete lattice with an empty set as the least element, $\{0,1\}^n$ as the greatest element, set intersection $\bigcap$ as greatest lower bound operation, closure of union $\bigsqcup$ as least upper bound operation as defined below.

Definition 2.4: The *closure of union* of n cones $C_1, C_2, \ldots, C_n$ is the set of points which are the boolean sum of any points in C_i's; it is denoted by $\bigsqcup\{C_1, C_2, \ldots, C_n\}$. $\square$

Lemma 2.4: Let $C_1, C_2, \ldots, C_n$ be boolean cones. $\bigsqcup\{C_1, C_2, \ldots, C_n\}$ is the set generated by the union of the generator sets of $C_1, C_2, \ldots, C_n$. $\square$

Example 2.4: Let $C_1 = \{(0,0,0),(0,1,0),(0,1,1)\} C_2 = \{(0,0,0),(0,1,0),(1,1,0)\}$ and C be $\bigsqcup\{C_1, C_2\}$. Their generator sets are $gen(C_1) = \{(0,1,0),(0,1,1)\}$, and $gen(C_2) = \{(0,1,0),(1,1,0)\}$. Their union is, $gen(C) = \{(0,1,0),(0,1,1),(1,1,0)\}$. Hence $C = \{(0,0,0),(0,1,0),(0,1,1),(1,1,0),(1,1,1)\}$. $\square$

2.3 Groundness Constraints

The success of relational groundness analysis relies on whether we have efficiently computable form of relations. Groundness relationships among arguments with respect to a predicate can be represented in the form of a set of boolean constraints.

Definition 2.4: Let $\vec{e} = (e_1, e_2, \ldots, e_n)$ be boolean terms and $c_1, c_2, \ldots, c_m$ boolean equalities. Let $\vec{p} = (p_1, \ldots, p_n)$ be a vector of boolean variables corresponding to groundness abstraction of arguments of predicate p.

$$G = \{\vec{p} \approx \vec{e}, c_1, c_2, \ldots, c_m\}$$

is called a *groundness constraint* for a predicate p. If $m = 0$, it is called a *simplified groundness constraint* for p. □

Theorem 2.5: A groundness constraint defines a boolean cone. □

We now give a theorem on boolean union of two groundness constraints. It can be naturally extended to n groundness constraints.

Theorem 2.6: Let G_1, G_2 be groundness constraints w.r.t. a predicate p as shown below.

$$G_1 = \{\vec{t} \approx \vec{e}, c_1, \ldots, c_m\}$$

and

$$G_2 = \vec{t} \approx \vec{f}, d_1, \ldots, d_o\}$$

Then the groundness constraints corresponding to the closure of union of G_1 and G_2 is as follows:

$$\bigsqcup\{G_1, G_2\} = \{\vec{t} = \vec{e} \vee \vec{f}, c_1, \ldots, c_m, d_1, \ldots, d_o\}$$

□

3 Simplifications and Normal Forms

The most costly operation is to test equivalence of two groundness constraints, which must be performed at each iteration of transformation concerning abstract interpretation. This operation is tantamount to finding generators of projection of cones, since generators are unique, minimal representation of cones. In most cones associated with practical programs, generators can be obtained by the following *simplification rules*.

- $x \vee y \vee \cdots \vee z \approx 0$ is equivalent to $x \approx y \approx \ldots \approx z \approx 0$.
- If $x \vee y \vee \cdots \vee z \approx w$ is a constraint and w does not appear on the left-hand side substitute the left-hand side into every place where w appears. If w appears nowhere, delete the constraint.
- If x appears on the same side as y in every occurrence, x can be deleted.

A simplified groundness constraint corresponds to a set of points including generators, which reduces to a generator set by removing redundancy. This is explained in Example 3.1.

Example 3.1: Let $C = \{\vec{p} \approx (x \vee w, y \vee w, x \vee y \vee w)\}$ be a simplified groundness constraint. $(x \vee w, y \vee w, x \vee y \vee w) \approx x \cdot (1,0,1) \vee y \cdot (0,1,1) \vee w \cdot (1,1,1)$ where $\cdot$ can be viewed as scalar multiplication and $\vee$ as componentwise boolean OR. Then $(1,0,1), (0,1,1)$, and $(1,1,1)$ are candidates for generators, and $(1,1,1)$ is redundant since $(1,1,1) = (1,0,1) \vee (0,1,1)$. So $gen(C) = \{(0,1,1), (1,0,1)\}$. This example also explains how to generate groundness constraints from generators. $\Box$

Let us turn our attention to groundness constraints which cannot be simplified by the above simplification rules. One way to find generators is to transform equalities with boolean terms to ones with real arithmetic terms. That is,

$$x_1 \vee x_2 \vee \ldots x_n \approx y_1 \vee y_2 \vee \ldots y_m$$

can be transformed to

$$\begin{aligned}
& x_1 + x_2 + \ldots x_n \geq y_1 \\
& x_1 + x_2 + \ldots x_n \geq y_2 \\
& \ldots \\
& x_1 + x_2 + \ldots x_n \geq y_m \\
& y_1 + y_2 + \ldots y_n \geq x_1 \\
& y_1 + y_2 + \ldots y_n \geq x_2 \\
& \ldots \\
& y_1 + y_2 + \ldots y_n \geq x_n \\
& x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \in \{0,1\}
\end{aligned}$$

It is easy to prove that both have the same set of solutions, and extreme rays of the linear arithmetic systems are vectors of 0 or 1's after being normalized so that one of their positive components is 1.

Theorem 3.1: If x is a generator of a boolean cone represented by a groundness constraint, then it is a normalized extreme ray of a convex cone represented by a linear arithmetic constraint transformed from the groundness constraint. $\Box$

As there are well-known methods to find extreme rays of convex cones (same as finding extreme points of convex polytopes) [VG91, Las90], we can find candidates for generators for the corresponding boolean cones by removing redundancy.

4 Bottom-Up Groundness Analysis

In the preceding sections, we described the relational abstract domain and some useful operations on the domain. We now give a bottom-up abstract interpretation to get success patterns in the form of groundness constraints.

Abstract interpretation consists of abstract domain and abstract operations which mimics base semantics faithfully. Hence bottom-up abstract interpretation mimics fixpoint semantics. Formal framework of bottom-up abstract interpretation can be found in [CC92, MS88].

Now we define a transformation mimicking immediate consequence operator.

Definition 4.1: Suppose we have k predicates whose names are $p, q, \ldots, r$ in an abstract program and p_i is i-th clause having the head predicate p and so forth. Let $P, Q, \ldots, R$ be groundness constraints associated with those predicates, and $\mathcal{U}_p$ the set of all boolean cones determined by the arity of the predicate p and so forth. Let $\mathcal{U} = (\mathcal{U}_p, \mathcal{U}_q, \ldots, \mathcal{U}_r)$. Let x be a vector of groundness constraints $(P, Q, \ldots, R) \in \mathcal{U}_p, \mathcal{U}_q, \ldots, \mathcal{U}_r)$. *Recursive transformation* T is a mapping from $\mathcal{U}$ to $\mathcal{U}$ and T_p is a mapping from $\mathcal{U}$ to $\mathcal{U}_p$ as given below.

$$\begin{aligned} T(x) &= (T_p(x), T_q(x), \ldots, T_r(x)) \\ T_p(x) &= \bigsqcup\{T_{p^1}(x), T_{p^2}(x), \ldots, T_{p^m}(x)\} \\ &\vdots \\ T_{p^i}(x) &= \{\vec{p} \approx \vec{a}, Q[\vec{q}/\vec{b}], \ldots, R[\vec{r}/\vec{c}]\} \end{aligned}$$

where there are m clauses having the head predicate p and the i-th abstract clause of the predicate p is in the form of

$$p(\vec{a}) \leftarrow q(\vec{b}), \ldots, r(\vec{c})$$

□

Theorem 4.1: There exists the least fixpoint of T and it can be reached in finite number of iterations.

Proof: By its definition, T_{p_i} is clearly monotone, so T is monotone. The domain $(\mathcal{U}_p, \mathcal{U}_q, \ldots, \mathcal{U}_r)$ forms a finite complete lattice, equipped with componentwise subset ordering. □

In practice, it is more efficient to process strongly connected components separately in predicate dependency graphs, starting from leaves in a tree induced by SCCs.

Example 4.2: Continuing with Example 4.1,

$$\begin{aligned} T^1 &= T(\emptyset) = \{\vec{a} \approx (0, u, u)\} \\ T^2 &= T(T^1) = \bigsqcup\{T^1, \{\vec{a} \approx (x \vee u, v, x \vee w, u \approx 0, v \approx u', w \approx u'\}\} \\ &= \{\vec{a} \approx (x, v, x \vee v\} \end{aligned}$$

Interested readers are invited to verify $T^3 = T^2$, which is the least fixpoint of T. Note that we only use simplification rules to reduce to simplified groundness constraints. The resulting fixpoint shows that in the success pattern of `append` procedure, if the first and second argument are ground, then the third is ground, and vice versa. □

5 Conclusion

We presented a novel relational groundness analysis in this paper. To compute groundness relations effectively, we used boolean constraints capturing boolean cones. Including freeness analysis and developing efficient test of cone equivalence will be future research directions.

Acknowledgements

We thank A. Van Gelder for helpful discussions.

References

[BJCB87] M. Bruynooghe, G. Janssens, A. Callebaut, and Demoen B. Abstract interpretation: Towards the global optimisation of Prolog programs. In *Proceedings of the 1987 International Symposium on Logic Programming, IEEE Press*, 1987.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1981.

[DW88] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.

[Las90] J.-L. Lassez. Parametric queries, linear constraints and variable elimination. In *Proceedings of DISCO 90, Springer Verlag Lecture Notes in Computer Science*, 1990.

[Mel81] C. S. Mellish. The automatic generation of mode declaration for logic programs. Technical Report DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1981.

[Mel85] C. S. Mellish. Some global optimizations for a prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[Mel87] C. S. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horword, Chichester, U.K., 1987.

[MS88] K. Marriott and H. Sondergaard. Bottom-up abstract interpretation of logic programs. In S. K. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the Fifth International Conference*, pages 733–748, Cambridge, Massachusetts, 1988. MIT Press.

[MU87] H. Mannila and E. Ukkonen. Flow analysis of Prolog programs. In *Proceedings of the 1987 International Symposium on Logic Programming.* IEEE Press, 1987.

[Plü90] L. Plümer. Termination proofs for logic programs based on predicate inequalities. In *Proc. 7th Int'l Conf. on Logic Programming*, pages 634–648, Jerusalem, 1990.

[SVG91] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Tenth ACM Symposium on Principles of Database Systems*, 1991.

[UVG88] J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.

[VG91] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 3, 1991. Extended abstract appears in Ninth ACM Symposium on Principles of Database Systems, 1990.

[War77] D. H. D. Warren. Implementing prolog - compiling predicate logic programs. Technical Report DAI Research Paper 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1977.