Precise and Efficient Groundness Analysis for Logic Programs

Kim Marriott
Dept. of Computer Science
Monash University
Clayton Vic. 3168
Australia

marriott@cs.monash.edu.au

Harald Søndergaard
Dept. of Computer Science
The University of Melbourne
Parkville Vic. 3052
Australia

harald@cs.mu.oz.au

Abstract

We show how precise groundness information can be extracted from logic programs. The idea is to use abstract interpretation with Boolean functions as "approximations" to groundness dependencies between variables. This idea is not new, and different classes of Boolean functions have been used. We argue, however, that one class, the *positive* functions is more suitable than others. Positive Boolean functions have a certain property which we (inspired by A. Langen) call "condensation." This property allows for rapid computation of groundness information.

Categories and Subject Descriptors: D.1.6 [**Programming Languages**]: Logic Programming; D.3.4 [**Programming Languages**]: Processors—*compilers*, *optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*, *invariants*, *logics of programs*

General Terms: Languages, Theory

Additional Key Words and Phrases: Abstract interpretation, condensation, groundness analysis, propositional logic.

1 Introduction

Groundness analysis is arguably the most important dataflow analysis for logic programs. The question: "At a given program point, is variable x always bound to a term that contains no variables?" is not only important for an optimizing compiler attempting to speed up unification, but for practically every programming tool which applies some kind of dataflow analysis. The reason is that most other analyses, such as independence analysis (whether instantiation of x indirectly instantiates other variables) or occur-check analysis (whether unification can safely be performed without the occur-check) employ groundness analysis to improve accuracy. For example, if x is ground (a terminological abuse we consistently use for "bound to a ground term"), then x cannot possibly share with other variables, and this is useful information for independence, occur-check, and many other dataflow analyses.

Dataflow analysis of logic programs is different from the analysis of imperative or functional programming languages. This is partly because the language is nondeterministic, but, more importantly, because dataflow in a sense is bidirectional, owing to the use of unification. The point is that a variable in a logic programming language is very different from a variable in the other language paradigms. It is sometimes referred to as a "logical variable" and characterized as "constrain-only" since the execution of a logic program proceeds by steps that continually narrow the set of possible values that a variable may take.

While this characteristic of logic program execution makes dataflow analyses harder in some ways, it also opens up new views of some analysis problems. For example it suggests the possibility of propagating conditional invariants of the form "*From this point on*, if variable x has (ever gets) property p, then variable y has (will have) property r." In this note we show how this applies to groundness. The idea is to let the statement "program variable x is ground" be represented by the propositional variable x. A groundness dependency such as "whenever y becomes ground, so does x" may then be represented by a Boolean function, in this case the function denoted by $y \rightarrow x$. If we use Boolean functions as approximations to runtime states, then abstract interpretation gives a natural way of specifying a very precise groundness analysis (Sections 2 and 3).

This idea is not new, and different classes of Boolean functions have been used for the purpose. The main purpose of this note is to show that exactly one of the classes, the *positive* functions, has a very useful property which we shall refer to as "condensation" (Section 4). Owing to this property, a relatively simple and efficient implementation is possible, yielding an extremely precise analysis. The analysis is efficient partly because it is query-independent. We describe the analysis in Section 5 and give a novel program transformation which makes it straightforward to extract information about runtime call patterns.

Furthermore, the analysis can be performed in an incremental fashion without loss of accuracy (Section 6). That is, dataflow information pertinent to a module may be computed independently of the predicates imported by that module. This is because the class of positive functions is not only condensing but also closed under existential quantification. Unfortunately quantifier elimination is expensive in the presence of Boolean functions that are unknown, so the precision-preserving extension of our analysis to modular programs may be of mainly theoretical interest.

2 Abstract interpretation of logic programs

Consider the idea of a logic program interpreter which answers queries by returning not only a set of answer constraints, but also a thoroughly annotated version of the program: For each program point it lists the constraints that are obtained there at some stage during evaluation of the given query. Since control may return to a program point many times during evaluation, each annotation is naturally a (possibly infinite) set of constraints.

Properly formalized, this idea leads to the notion of a *collecting semantics*, a semantics which gives very precise dataflow information, but which is of course not finitely computable in general. However, if we replace the possibly infinite sets of constraints by more crude "approximations" or "descriptions" of sets of constraints, then we may obtain a dataflow analysis which terminates in

finite time. This is the idea behind abstract interpretation of logic programs.

Example 2.1 For the program

$$q(u,v) \quad \leftarrow \quad ⓐ\, q(f(u),v)\, ⓑ$$

and query $\leftarrow q(a,x)$, the collecting semantics associates $\{u = a, u = f(a), u = f(f(a)), \ldots\}$ with ⓐ and $\emptyset$ with ⓑ. A more crude approximation to this associates with ⓐ and ⓑ sets that are greater than or equal to those given by the collecting semantics. If we introduce the notation $\{u\}$ for "the set of constraints that ground u" then associating $\{u\}$ with both ⓐ and ⓑ is a correct approximation. Though not very precise, it still tells us that every spawned query will have a ground first argument. ■

This leads to the view that a dataflow analysis is an approximation to the collecting semantics. P. and R. Cousot [6] have formalized that idea as follows. The standard domain Y (the powerset of constraints in the example) and the description domain X (the powerset of variables in the example) should be complete lattices with the monotonic $\gamma : X \to Y$ giving the meaning of the descriptions in X. Furthermore, $\gamma\ X$, the image of X under γ, should be a *Moore family*, that is, closed under the greatest lower bound operation on Y. In Example 2.1 we have

$$\gamma\ V = \{e \mid e \text{ grounds every } v \in V\}.$$

The ordering on the sets of variables is superset ordering in this case. This is because a larger set of constraints will have fewer variables made ground by *all* constraints in the set. In general the ordering on the lattice of descriptions is opposite to that normally used in denotational semantics: a "higher" description applies to a larger set of constraints, that is, it has less information contents. So "higher" means "less precise."

The notion of approximation is made exact as follows: $x \in X$ approximates $y \in Y$ iff $y \leq \gamma\ x$. We write this $x \propto y$. For *syntactic objects*, such as programs and primitive constraints, $S \propto S'$ iff $S = S'$. Finally we extend $\propto$ to function spaces as follows. Consider $f : X \to X'$ and $g : Y \to Y'$. We define $f \propto g$ iff $\forall\, x \in X\ .\ \forall\, y \in Y\ .\ x \propto y \Rightarrow (f\ x) \propto (g\ y)$.

A formal semantics can be given by laying down the appropriate semantic domains and specifying certain operators on these domains. For groundness analysis there is only one important operation, *add*, which takes a term equation and a set of constraints and forms the possible conjunctions:

$$add\ e\ E = \{e \wedge e' \mid e' \in E\} \setminus \{false\}.$$

An important feature of P. and R. Cousot's "adjoined" framework is that given an operation on the standard domain, there is a least, that is, most precise, operation on the description domain which approximates it. This is because descriptions form a Moore family. We shall refer to such "most precise" operations as the *induced* operations. In general, the unique induced approximation to $g : Y \to Y'$ is

$$\lambda\, x\ .\ \bigsqcap\{x' \mid g\ (\gamma\ x) \leq \gamma\ x'\}.$$

When referring to an *induced groundness analysis* we mean the analysis obtained by using the induced version of *add*.

3 Groundness analysis using Boolean functions

As we have seen, a very natural description domain for groundness analysis are sets of variables which are definitely ground at a particular program point. Such descriptions were introduced by Mellish and used in early mode and groundness analyzers [11, 18]. We regard the set $\{x_1, ..., x_n\}$ as representing the n-place Boolean function $x_1 \wedge .... \wedge x_n$ to which we attach the meaning: at this point all of the (program) variables $x_1, \ldots, x_n$ are bound to ground terms. We let *Con* be the set of all such conjunctions of variables. *Con* is the simplest useful description domain for groundness analysis, as anything less expressive cannot capture the type of statements that we require from any groundness analysis.

However, *Con* does not lead to very precise groundness analysis because it ignores "aliasing" and other dependencies between variables, witness the following example (from here on we only label program points of interest).

Example 3.1 Consider the following program and query $q(x, y)$.

$q(x, y) \quad :- \quad p(x, y), \textcircled{a}\ r(x) \textcircled{b}$
$p(u, u)$
$r(a)$

An analysis based on *Con* will deduce that x will be ground at ⓑ, but fail to deduce that y will also be ground. The fact that x and y are aliases is not captured. ■

Aliasing information can be captured by more complex Boolean functions in which implication is allowed. As an example, if the constraint $x = f(y, z)$ is generated during query evaluation, the description $x \leftrightarrow (y \wedge z)$ can be deduced. Informally the formula says that if x is (becomes) ground then so is (does) both of y and z, and *vice versa*. Such Boolean formulas, called *dependency formulas*, were introduced by Dart and used for groundness analysis in deductive databases [7, 8]. We denote the set of dependency formulas by *Def*, since they correspond to the Boolean functions that can be written as *definite* propositional formulas. If the program in Example 3.1 is analyzed using dependency formulas as descriptions, ⓐ will have the description $x \leftrightarrow y$, and therefore ⓑ will have the description $x \wedge y$, as desired.

Dependency formulas give rise to very precise groundness analyses. However, an even more precise groundness analysis is possible with *Pos*, the set of *positive* Boolean functions. These extend *Def* by allowing disjunction. *Pos* was suggested for groundness analysis by Marriott and Søndergaard [14] (under the less suggestive name '*Prop*') and further studied by Cortesi *et al.* [5].

Example 3.2 Consider the following program and query $q(x, y)$.

$q(x, y) \quad :- \quad p(x, y), \textcircled{a}\ r(x, y) \textcircled{b}$
$p(a, u)$
$p(u, b)$
$r(u, u)$

An analysis based on *Def* cannot deduce that either x or y are ground at ⓑ. The best description it can produce is $x \leftrightarrow y$. However an analysis using *Pos* will find that ⓐ has the description $x \vee y$, and so, by conjoining this information with $x \leftrightarrow y$ (from r) it finds that at ⓑ both x and y are ground. ■

In practice it is not common for programs to require the extra precision of *Pos*. Nonetheless we feel that *Pos* is the better description domain for groundness analysis. This is because *Pos* has an important property which allows it to be more efficiently implemented. This is somewhat surprising, because one would expect extra precision to come at the cost of efficiency. The reason is that *Pos* is closed under disjunction and so is "condensing," a property we discuss in Section 4. In the remainder of this section we will formalize *Pos* and the other description domains.

We note that *Pos* is the largest class of Boolean functions which it makes sense to use as descriptions in a groundness analysis. Since groundness is undecidable, a dataflow analysis operating in finite time can only give approximate groundness information. The statements that it produces will carry a modality, as in "x is *inevitably* ground." For this reason, only the *positive* Boolean functions are useful. If we associate the meaning "x is inevitably ground" with the formula x, then $\neg x$ would mean "x may not always be ground" and this contains no information at all, that is, exactly the information conveyed by the constant function *true*.

Definition. A *Boolean function* is a function $F : Bool^n \rightarrow Bool$. We call the set of all n-ary Boolean functions $Bfun_n$ and let it be ordered by logical consequence ($\models$). The function F is *positive* iff $F(true, \ldots, true) = true$. We denote the set of positive Boolean functions of n variables by Pos_n. ■

For simplicity we will assume that we have some fixed number n of variables and leave out subscripts and the phrase "of n variables." We shall also use propositional formulas as representations of Boolean functions without worrying about the distinction. Thus we may speak of a formula as if it were a function and in any case denote it by F. As a reminder of the fact that a propositional formula is only one of a class which all represent a given Boolean function, we put square brackets around propositional formulas and think of the result as the class of equivalent formulas. By a slight abuse of notation we sometimes apply logical connectives to these classes of equivalent formulas.

It is well known that *Bfun* is a Boolean lattice and in fact *Pos* forms a Boolean sublattice of *Bfun*. In terms of propositional formulas, meet and join are given by conjunction and disjunction, respectively.

Example 3.3 The Boolean functions $[x \rightarrow y]$ and $[x \vee y]$ are in *Pos*, but $[\neg x]$ is not. ■

It is convenient to include the (non-positive) Boolean function *false* as an approximation to the empty set of constraints. So we will in fact be dealing with the complete lattice $Pos_\perp = Pos \cup \{false\}$, ordered by logical consequence.

Let *Sub* denote the set of substitutions and let *Eqn* denote the set of constraints. In the Prolog setting of this note, constraints are possibly existentially quantified term equations, but the ideas

generalize naturally to constraint logic languages. A substitution θ is a *unifier* of the constraint e iff $\models (\theta\ e)$. Let *unif* e denote the set of unifiers of e. The idea with using $Pos_\perp$ is that a constraint e is described by $\phi \in Pos_\perp$ exactly in case that, for every unifier θ of e, the truth assignment corresponding to the variables ground by θ satisfies ϕ.

Definition. For a substitution θ, let *grounds* θ be the truth assignment which maps a variable to true if θ grounds the variable and to false otherwise. That is, *grounds* : $Sub \to Var \to Bool$ is defined by

$$grounds\ \theta\ V \Leftrightarrow vars\ (\theta\ V) = \emptyset$$

where *vars* S is the set of variables in syntactic object S. The function $\gamma : Pos_\perp \to (\mathcal{P}\, Eqn)$ is defined by

$$\gamma\ \phi = \{e \in Eqn \mid \forall\, \theta \in unif\ e\ .\ (grounds\ \theta) \models \phi\}.\ \blacksquare$$

Thus γ maps $\phi \in Pos_\perp$ to the set of constraints e which have the property that, no matter how they further become constrained to some e', the Boolean function corresponding to e' will still satisfy ϕ.

Example 3.4 The Boolean function $[x \leftrightarrow y]$ describes both of the constraints $x = a \wedge y = b$ and $\exists\, u'\ .\ (x = u' \wedge y = u' \wedge u = u')$, but not the constraint $x = a$. However, $[x \leftrightarrow y]$ is not the *best* description for $x = a \wedge y = b$. For this constraint the best description is $[x \wedge y]$ which in turn has $[x \leftrightarrow y]$ as a logical consequence. That is, $[x \leftrightarrow y]$ is a less precise approximation than $[x \wedge y]$.

As a further example, let $\phi = [x \wedge (y \leftrightarrow z)]$. Then $(x = a \wedge y = f(z))$ is approximated by ϕ but $(x = a \wedge y = f(a))$ is not. $\blacksquare$

Proposition 3.1 [17] $\gamma\ Pos_\perp$ is a Moore family. $\blacksquare$

Let us define *Con* and *Def* precisely:

Definition. Let *Def* be the largest class of positive Boolean functions whose models are closed under intersection and let *Con* be the monotonic functions in *Def*. $\blacksquare$

Like $\gamma\ Pos_\perp$, both $\gamma\ Con_\perp$ and $\gamma\ Def_\perp$ are Moore families, so *Con* and *Def* are applicable as domains for groundness analysis. It is well known that existential quantifiers can be eliminated from Boolean functions, as $\exists\, x\ .\ f(\ldots x \ldots) = f(\ldots false \ldots) \vee f(\ldots true \ldots)$. What is perhaps less obvious is the fact that *Con*, *Def*, and *Pos* are all closed under existential quantification. This simplifies their use in groundness analysis, since an important operation is the "projection" away from variables that have become uninteresting, and the projection operation is exactly existential quantification.

4 Condensation

Sometimes optimizations performed by a compiler are only applicable for certain types of queries. For this reason, most dataflow analyses that have been suggested for logic programs are *query directed*. That is, they compute an approximation to the collecting semantics for a particular query description.

As we have seen, the collecting semantics gives information about program points. In particular it gives information about *calls* to a predicate and the *answers* to a predicate. For instance, in Example 3.2, ⓐ collects information about the calls to $r(x, y)$ and ⓑ collects information about the answers to $q(x, y)$.

Information about the calls and answers may also be computed in a *query-independent* fashion. In this case, the analysis is performed for all possible queries. This has the advantage that only one analysis is performed, even when information about many different queries is needed, since information about a particular query can be efficiently derived from the result of a query-independent analysis.

Unfortunately the result from such a two-step process may well be less precise than that obtained from separate query-directed analysis. This will be apparent from Example 4.1 below. However, one point of this note is that there are cases in which no precision is lost by using a query-independent analysis. This happens when a domain is "condensing." The name comes from the "condensing procedures" introduced by Langen [12] and used by Jacobs and Langen [10] in their work on dataflow analysis of logic programs.

We can think of a query-directed dataflow analysis as a function which takes a program, a query, and possibly a statement about the initial state of variables in the query, yielding a statement about the answers. Fixing the program P and the query Q, we may write the function as $F^{\Phi}_{P,Q} : \Phi \to \Phi$, where Φ is the domain of descriptions, or statements, available. We put the superscript Φ as a reminder of the fact that the dataflow analysis is the one *induced* by Φ. A precise denotational definition of $F^{\Phi}_{P,Q}$ for parametric Φ can be found in [17].

Definition. The description domain Φ is *condensing* iff

$$F^{\Phi}_{P,Q}\ (\phi \wedge \phi') = \phi \wedge (F^{\Phi}_{P,Q}\ \phi')$$

for all programs P, queries Q, and statements $\phi, \phi' \in \Phi$. ■

Note that in particular $F^{\Phi}_{P,Q}\ \phi = \phi \wedge (F^{\Phi}_{P,Q}\ \mathit{true})$. This means that an analysis with respect to program P and query Q need not be repeated for several different ϕ. It only needs to be performed in the most general case (true) where we make no assumptions about the instantiation of variables whatsoever. If this yields ϕ' then the required answer is $\phi \wedge \phi'$. This applies not just to the top-level query, but to every call made during the execution of P.

Also note that the definition of condensing only considers the *answers* to a query. However, as we shall see, condensation also allows us to compute information pertaining to arbitrary program points.

Example 4.1 Consider the program P and query Q from Example 3.2. We have

$$\begin{array}{lcl} F^{Con}_{P,Q}\,[true] & = & [true] \\ F^{Def}_{P,Q}\,[true] & = & [x \leftrightarrow y] \\ F^{Pos}_{P,Q}\,[true] & = & [x \wedge y]. \end{array}$$

This illustrates the relative accuracy of the three domains. Furthermore,

$$\begin{array}{lcccl} F^{Con}_{P,Q}\,[x] & = & [x \wedge y] & \neq & [x] \wedge (F^{Con}_{P,Q}\,[true]) \\ F^{Def}_{P,Q}\,[x \leftrightarrow y] & = & [x \wedge y] & \neq & [x \leftrightarrow y] \wedge (F^{Def}_{P,Q}\,[true]) \end{array}$$

so *Con* and *Def* are not condensing. ■

Our use of "condensing" is somewhat different from that of Jacobs and Langen. Our notion is stronger, as we consider only the *induced* analysis $F^{\Phi}_{P,Q}$. This is what makes it possible to consider "condensing" a property of the domain Φ. Jacobs and Langen allow for sub-optimal analyses, the idea being that sometimes condensation can be achieved by introducing loss of accuracy in the analysis. Even though the notions differ, the following result can be obtained by combining Theorems 14, 65, 67, and 68 of Langen [12] since the operations Langen uses with *Pos* are in fact the induced operations (this is not the case for other domains in Langen's treatment).

Theorem 4.1 $Pos_{\perp}$ is condensing. ■

We saw that $Con_{\perp}$ and $Def_{\perp}$ are not condensing. In fact since closure under both $\vee$ and $\rightarrow$ are necessary for condensation, we have:

Theorem 4.2 $Pos_{\perp}$ is the smallest extension of $Con_{\perp}$ which is condensing. ■

To summarize: As $Pos_{\perp}$ is condensing, groundness analysis using *Pos* can be modular. Once we have a closed form for the result of calling predicate p, we need not worry about the context in which p is called, or compute p's result for different environments.

5 Efficient groundness analysis

Debray [9] has shown that an analysis using *Pos* is in effect solving an EXPTIME-complete problem, where complexity is measured in terms of program size. However, under the reasonable assumption that the number of variables in any clause is bounded, an analysis using *Pos* is in PTIME. In fact, a number of independent implementations of *Pos* indicate that accurate groundness analysis is perfectly practical for "real-world" programs. A natural representation for Boolean functions, used for example by Le Charlier and Van Hentenryck [13], is by means of ordered binary-decision graphs [1]. Le Charlier and Van Hentenryck report good results from instantiating a generic "abstract interpreter" with *Pos*.

But precise groundness analysis can be performed in an even more efficient way, as *Pos* is condensing. While what we want is information about the *calls* that take place in a top-down execution, it suffices to compute only "bottom-up" information which gives an approximation to the answers of the program [16]. We first explain how to compute answer information.

Example 5.1 Let P be the program

$$\begin{array}{lcl} overlap(x,y) & \leftarrow & x = u:z, member(u,y) \\ overlap(x,y) & \leftarrow & x = u:z, overlap(z,y) \\ member(u,y) & \leftarrow & y = u:z \\ member(u,y) & \leftarrow & y = v:z, member(u,z) \end{array}$$

As a first step we form the Clark completion [2] of P:

$$\begin{array}{lcl} overlap(x,y) & \leftrightarrow & \exists\, uz\,.\,(x = u:z \wedge (member(u,y) \vee overlap(z,y))) \\ member(u,y) & \leftrightarrow & \exists\, vz\,.\,(y = u:z \vee (y = v:z \wedge member(u,z))). \end{array}$$

A straightforward translation yields the following recursive definitions of the *Boolean functions* *overlap* and *member*:

$$\begin{array}{lcl} overlap(x,y) & = & \exists\, uz\,.\,([x \leftrightarrow (u \wedge z)] \wedge (member(u,y) \vee overlap(z,y))) \\ member(u,y) & = & \exists\, vz\,.\,([y \leftrightarrow (u \wedge z)] \vee ([y \leftrightarrow (v \wedge z)] \wedge member(u,z))). \end{array}$$

Solving these (for *member* first, then for *overlap*), using Kleene iteration, we get

$$\begin{array}{lcl} member_0(u,y) & = & [false] \\ member_1(u,y) & = & \exists\, vz\,.\,[(y \leftrightarrow (u \wedge z)) \vee ((y \leftrightarrow (v \wedge z)) \wedge false)] \\ & = & \exists\, z\,.\,[y \leftrightarrow (u \wedge z)] \\ & = & [y \rightarrow u] \end{array}$$

$$\begin{array}{lcl} overlap_0(x,y) & = & [false] \\ overlap_1(x,y) & = & \exists\, uz\,.\,[(x \leftrightarrow (u \wedge z)) \wedge ((y \rightarrow u) \vee false)] \\ & = & \exists\, uz\,.\,[(x \leftrightarrow (u \wedge z)) \wedge (y \rightarrow u)] \\ & = & \exists\, u\,.\,[(x \rightarrow u) \wedge (y \rightarrow u)] \\ & = & [true]. \end{array}$$

It is easy to verify that $member_1$ and $overlap_1$ make up a fixpoint and so are least solutions to the system of recursive Boolean equations. ■

These results tell us for example that if $member(u,y)$ is called with a groundness statement ϕ then the result will be $\phi \wedge [y \rightarrow u]$. As far as *overlap* is concerned, the analysis cannot say anything about groundness or groundness dependencies. This is adequate, since *overlap* does not in fact introduce any groundness.

As a second example, let us compute the groundness information for the well-known append program and the naive list reversal program. This is more interesting than the example above, as it requires more iteration steps.

Example 5.2 The programs are

$$\begin{array}{lcl} app(x,y,z) & \leftarrow & x = nil, y = z \\ app(x,y,z) & \leftarrow & x = u' : x', y = y', z = u' : z', app(x', y', z') \\ \\ rev(x,y) & \leftarrow & x = nil, y = nil \\ rev(x,y) & \leftarrow & x = z : v, rev(v,w), u = z : nil, \textcircled{a}\ app(w,u,y) \end{array}$$

The corresponding Boolean functions are defined by

$$\begin{array}{lcl} app(x,y,z) & = & [x \wedge (y \leftrightarrow z)] \vee \\ & & \exists\, u'x'y'z' \,.\, ([x \leftrightarrow (u' \wedge x')] \wedge [y \leftrightarrow y'] \wedge [z \leftrightarrow (u' \wedge z')] \wedge app(x', y', z')) \\ rev(x,y) & = & [x \wedge y] \vee \\ & & \exists\, uvwz \,.\, ([x \leftrightarrow (z \wedge v)] \wedge rev(v,w) \wedge [z \leftrightarrow u] \wedge app(w,u,y)). \end{array}$$

The equation for *app* is solved by the iteration

$$\begin{array}{lcl} app_0(x,y,z) & = & [false] \\ app_1(x,y,z) & = & [x \wedge (y \leftrightarrow z)] \vee \\ & & \exists\, u'x'y'z' \,.\, ([x \leftrightarrow (u' \wedge x')] \wedge [y \leftrightarrow y'] \wedge [z \leftrightarrow (u' \wedge z')] \wedge [false]) \\ & = & [x \wedge (y \leftrightarrow z)] \\ app_2(x,y,z) & = & [x \wedge (y \leftrightarrow z)] \vee \\ & & \exists\, u'x'y'z' \,.\, [(x \leftrightarrow (u' \wedge x')) \wedge (y \leftrightarrow y') \wedge (z \leftrightarrow (u' \wedge z')) \wedge x' \wedge (y' \leftrightarrow z')] \\ & = & [(x \wedge y) \leftrightarrow z]. \end{array}$$

This is the least solution to the recursive definition of the Boolean function *app*. Using this we can now solve for *rev*:

$$\begin{array}{lcl} rev_0(x,y) & = & [false] \\ rev_1(x,y) & = & [x \wedge y] \\ rev_2(x,y) & = & [x \leftrightarrow y] \end{array}$$

which is the least solution to the recursive definition of *rev*. ■

Condensation also allows the efficient computation of (query-directed) information about *calls* occurring at runtime. The idea is to use a program transformation which augments the program with predicates whose answers are the calls occurring in the original program. Given a program point ⓐ in a clause

$$\ldots \leftarrow \ldots \textcircled{a}\ p(y_1, \ldots, y_n), \ldots$$

we define a new predicate $q\textcircled{a}(x_1, \ldots, x_m, y_1^{\#}, \ldots, y_n^{\#})$ which holds if and only if a call to $q(x_1, \ldots, x_m)$ spawns the call to $p(y_1^{\#}, \ldots, y_n^{\#})$ immediately after ⓐ. As before, this gives rise to a recursive definition of a Boolean function $q\textcircled{a}$ which we can then compute once and for all. We illustrate this with two examples.

Example 5.3 We repeat part of the program from Example 5.1 for convenience:

$$\begin{array}{lcl} overlap(x,y) & \leftarrow & x = u:z, \textcircled{a}\ member(u,y) \\ overlap(x,y) & \leftarrow & x = u:z, overlap(z,y). \end{array}$$

Assume we are interested in the calls to *member* at point ⓐ in the context of a query to *overlap*. As a first step we construct the new predicate

$$\begin{array}{lcl} overlap\textcircled{a}(x,y,u^{\#},y^{\#}) & \leftarrow & x = u:z, (u,y) = (u^{\#},y^{\#}) \\ overlap\textcircled{a}(x,y,u^{\#},y^{\#}) & \leftarrow & x = u:z, overlap\textcircled{a}(z,y,u^{\#},y^{\#}) \end{array}$$

The two clauses come from the two original clauses, respectively. Continuing as before we get:

$$\begin{array}{l} overlap\textcircled{a}(x,y,u^{\#},y^{\#}) = \\ \qquad \exists\, uz\,.\,([x \leftrightarrow (u \wedge z)] \wedge ([(u \leftrightarrow u^{\#}) \wedge (y \leftrightarrow y^{\#})] \vee overlap\textcircled{a}(z,y,u^{\#},y^{\#}))). \end{array}$$

Solving this equation, we get

$$\begin{array}{lcl} overlap\textcircled{a}_0(x,y,u^{\#},y^{\#}) & = & [false] \\ overlap\textcircled{a}_1(x,y,u^{\#},y^{\#}) & = & \exists\, uz\,.\,([x \leftrightarrow (u \wedge z)] \wedge ([(u \leftrightarrow u^{\#}) \wedge (y \leftrightarrow y^{\#})] \vee false)) \\ & = & [(x \rightarrow u^{\#}) \wedge (y \leftrightarrow y^{\#})]. \end{array}$$

This is the least fixpoint for the equation that defines $overlap\textcircled{a}$. It tells us for example, that at ⓐ, *member* will be called with a ground second argument if *and only if* *overlap* is called with a ground second argument. ■

Example 5.4 Now assume we are interested in the calls to *app* at point ⓐ in the program in Example 5.2, in the context of a query to *rev*. The transformation step gives the recursive predicate:

$$\begin{array}{lcl} rev\textcircled{a}(x,y,w^{\#},u^{\#},y^{\#}) & \leftarrow & x = z:v, rev(v,w), u = z:nil, (w,u,y) = (w^{\#},u^{\#},y^{\#}) \\ rev\textcircled{a}(x,y,w^{\#},u^{\#},y^{\#}) & \leftarrow & x = z:v, rev\textcircled{a}(v,w,w^{\#},u^{\#},y^{\#}) \end{array}$$

The corresponding Boolean function is defined by

$$\begin{array}{lcl} rev\textcircled{a}(x,y,w^{\#},u^{\#},y^{\#}) & = & \exists\, uvwz\,.\,([x \leftrightarrow (z \wedge v)] \wedge rev(v,w) \wedge [z \leftrightarrow u] \wedge [w \leftrightarrow w^{\#}] \wedge \\ & & [u \leftrightarrow u^{\#}] \wedge [y \leftrightarrow y^{\#}]) \vee \\ & & \exists\, vwz\,.\,([x \leftrightarrow (z \wedge v)] \wedge rev\textcircled{a}(v,w,w^{\#},u^{\#},y^{\#})). \end{array}$$

The equation for $rev\textcircled{a}(x,y,w^{\#},u^{\#},y^{\#})$ is solved by the iteration

$$\begin{array}{lcl} rev\textcircled{a}_0(x,y,w^{\#},u^{\#},y^{\#}) & = & [false] \\ rev\textcircled{a}_1(x,y,w^{\#},u^{\#},y^{\#}) & = & [x \leftrightarrow (u^{\#} \wedge w^{\#})] \wedge [y \leftrightarrow y^{\#}] \\ rev\textcircled{a}_2(x,y,w^{\#},u^{\#},y^{\#}) & = & [x \rightarrow (u^{\#} \wedge w^{\#})]. \end{array}$$

This tells us that if $rev(x,y)$ is called with x ground then calls to *app* at program point ⓐ will have the first and second arguments ground. ■

In short, our query-independent analysis gives information about calls and answers. Owing to condensation, the result contains sufficient information that the result *with respect to a specific query* can be derived efficiently and with no loss of accuracy. The positive Boolean functions contain sufficient information about groundness dependencies to handle every possible call context by simply conjoining functions. This speeds up computation and is simpler than more general query-directed implementations of *Pos*.

The transformation given here generalizes the well-known magic-set transformation, which can be used to compute information about the calls for a single query, see for example [19]. Codish and Demoen [4] have independently investigated query-independent groundness analysis based on *Pos*. While they do not consider precision, Codish and Demoen have implemented their analysis. Their experimental results are most encouraging, especially as in the Codish/Demoen approach, the computation of query-independent information about call patterns requires the computation of a fixpoint, whereas in our approach, only a single conjunction is required for each program point.

6 Incremental groundness analysis

Groundness analysis using *Pos* may be incremental, that is, the dataflow information pertinent to a module may be computed for the module, independently of the predicates it imports. We illustrate the basic idea with two examples.

Example 6.1 Consider again the definition of *overlap*:

$$\begin{array}{lcl} overlap(x,y) & \leftarrow & x = u:z, member(u,y) \\ overlap(x,y) & \leftarrow & x = u:z, overlap(z,y) \end{array}$$

but now assume that the definition of *member* is not available. However, we can still analyze *overlap* by solving the recursive equation

$$overlap(x,y) = \exists\, uz\,.\,([x \leftrightarrow (u \wedge z)] \wedge (member(u,y) \vee overlap(z,y))).$$

Kleene iteration gives:

$$\begin{array}{lcl} overlap_0(x,y) & = & [false] \\ overlap_1(x,y) & = & \exists\, uz\,.\,([x \leftrightarrow (u \wedge z)] \wedge (member(u,y) \vee [false]) \\ & = & \exists\, uz\,.\,([x \leftrightarrow (u \wedge z)] \wedge member(u,y)) \\ & = & \exists\, u\,.\,([x \rightarrow u] \wedge member(u,y)) \\ & = & ([\neg x] \wedge member(false,y)) \vee member(true,y) \\ & = & (member(false,y) \rightarrow [x]) \rightarrow member(true,y). \end{array}$$

This is the least solution to the equation for *overlap*.

Later, when the result for *member* arrives it can be plugged in, giving the result for *overlap*. For example, if $member(u,y)$ turns out to have the description $[u \wedge y]$, that is, it always grounds its arguments, then the description for $overlap(x,y)$ will be $[y]$. Actually, we know from Section 5

that $member(u, y)$ is approximated by the Boolean function $[y \rightarrow u]$. When this is substituted in the formula for *overlap*, we obtain $[true]$, as expected. ■

Example 6.2 Now consider the definition of the ancestor relationship which is defined in terms of the parent relation.

$$\begin{array}{lcl} ancestor(x, y) & \leftarrow & parent(x, y) \\ ancestor(x, y) & \leftarrow & parent(x, z), ancestor(z, y). \end{array}$$

We can analyze *ancestor* by solving the recursive equation

$$ancestor(x, y) = \exists\, z\, .\, (parent(x, y) \vee (parent(x, z) \wedge ancestor(z, y))).$$

Kleene iteration gives:

$$\begin{array}{lcl} ancestor_0(x, y) & = & [false] \\ ancestor_1(x, y) & = & parent(x, y) \end{array}$$

which is the least solution to the equation for *ancestor*. Later, the result for *parent* can be substituted, giving the result for *ancestor*. For example, if $parent(x, y)$ turns out to have the description $[x \wedge y]$, then the description for $ancestor(x, y)$ will be $[x \wedge y]$. ■

There are three observations to make about the method illustrated in these examples.

1. It is generally applicable, even in the context of mutually recursive modules. Once the equations are combined then, if they are mutually recursive, Kleene iteration can be used to solve the system. In the case they are not mutually recursive, simple substitution suffices.

2. It will always terminate after a finite number of iterations with a finite approximation. This is because the descriptions for the missing predicates are functions in *Pos*, and *Pos* is closed under existential quantification. Thus, we can perform quantifier elimination to remove the local variables, leaving only the constants *true* and *false* or non-quantified variables.

3. Performing the analysis incrementally does not change the result, that is, no accuracy is lost. This is because *Pos* is condensing.

Unfortunately, although the analysis will always terminate, it is not clear that it can be implemented efficiently, as for some practical programs analyzed by hand, the resulting equations are very large. This blow-up is caused by applying quantifier elimination to the missing descriptions, and it is difficult to see how it can be reduced.

Codish *et al.* [3] have given a general framework for incremental analysis based on an "unfolding" semantics. The particular analysis sketched here differs from analyses in their framework in two ways. First, because of the use of quantifier elimination, there is no need to introduce "star abstractions" as our analysis will always terminate. Second, performing the analysis incrementally never loses precision. This is because *Pos* is condensing, and because star abstraction is not used. In the more general case of [3] no assumption of condensation can be made, so that performing an analysis incrementally may lose precision.

7 Conclusion

Groundness analysis is an important component of most dataflow analyses for (constraint) logic programs. In general, combination of dataflow analyses is not easy, but for most composite analyses it turns out that the groundness component may be computed separately, first, and the groundness results may then be used by other analyses, without this staging having negative effects on accuracy. For this reason, it is important that groundness analysis be accurate while still efficient.

Pos gives rise to one of the most precise groundness analyses that we know of. In particular it achieves more accuracy than more complex analyses based on "reexecution" or "propagation" [15]. By utilizing condensation it is possible to implement groundness analysis based on *Pos* efficiently. This is empirically supported by the recent implementation of Codish and Demoen [4]. A further advantage of *Pos* is that the analysis may be incremental, that is, the dataflow information pertinent to a module may be computed for the module, independently of the predicates it imports and with no loss of accuracy.

Acknowledgements

We thank Mike Codish and an anonymous referee for useful suggestions that have improved this paper.

References

[1] Bryant, R. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* **24** (3): 293–318, 1992.

[2] Clark, K. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[3] Codish, M., Debray, S., and Giacobazzi, R. Compositional analysis of modular logic programs. *Proc. Twentieth ACM Symp. Principles of Programming Languages*, pages 451–464. Charleston, South Carolina, 1993.

[4] Codish, M., and Demoen, B. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. To appear in *Proc. Int. Logic Programming Symp.*, Vancouver, Canada, 1993.

[5] Cortesi, A., Filé, G., and Winsborough, W. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. *Proc. Sixth Ann. IEEE Symp. Logic in Computer Science*, pages 322–327. Amsterdam, The Netherlands, 1991.

[6] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proc. Sixth Ann. ACM Symp. Principles of Programming Languages*, pages 269–282. San Antonio, Texas, 1979.

[7] Dart, P. *Dependency Analysis and Query Interfaces for Deductive Databases.* PhD thesis, The University of Melbourne, Australia, 1988.

[8] Dart, P. On derived dependencies and connected databases. *Journal of Logic Programming* **11** (2): 163–188, 1991.

[9] Debray, S. On the complexity of dataflow analysis of logic programs. In W. Kuich, editor, *Proc. Nineteenth Int. Coll. Automata, Languages and Programming* (Lecture Notes in Computer Science 623), pages 509–520. Springer-Verlag, 1992.

[10] Jacobs, D., and Langen, A. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming* **13** (2&3): 291–314, 1992.

[11] Jones, N. D., and Søndergaard, H. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood, 1987.

[12] Langen, A. *Advanced Techniques for Approximating Variables in Logic Programs.* PhD Thesis, University of Southern California, Los Angeles, California, 1991.

[13] Le Charlier, B., and Van Hentenryck, P. Groundness analysis for Prolog: Implementation and evaluation of the domain Prop. *Proc. ACM Symp. Partial Evaluation and Semantics-Based Program Manipulation*, pages 99–110, Copenhagen, Denmark, 1993.

[14] Marriott, K., and Søndergaard, H. Notes for a tutorial on abstract interpretation of logic programs. Presented to North American Conf. Logic Programming, Cleveland, Ohio, 1989.

[15] Marriott, K., and Søndergaard, H. On propagation-based analysis of logic programs. Technical Report 93/8. Dept. of Computer Science, University of Melbourne, Australia, 1993.

[16] Marriott, K., and Søndergaard, H. Semantics-based dataflow analysis of logic programs. In G. X. Ritter, editor, *Information Processing 89*, pages 601–606. North-Holland, 1989.

[17] Marriott, K., Søndergaard, H., and Jones, N. D. Denotational abstract interpretation of logic programs. To appear in *ACM Trans. Programming Languages and Systems.*

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

[19] Nilsson, U. Abstract interpretation: A kind of magic. In J. Małuszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming* (Lecture Notes in Computer Science 528), pages 299-309. Springer-Verlag, 1991.

End of Paper; Total pages = 15. Words = 4953 plus abstract (81 words) plus bibliography.