On Propagation-Based Analysis of 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

Notions such as "reexecution" and "propagation" have recently attracted attention in dataflow analysis of logic programs. Both techniques promise more accurate dataflow analysis without requiring more complex description domains. Propagation, however, has never been given a formal definition. It has therefore been difficult to discuss properties such as correctness, precision, and termination of propagation.

We suggest a definition of propagation. Comparing propagation-based analysis with the more conventional approach based on abstract interpretation, we find that propagation involves a certain inherent loss of precision when dataflow analyses are based on description domains which are not "downwards closed" (including mode analysis). In the archetypical downwards closed case, groundness analysis, we contrast approaches using Boolean functions as descriptions with those using propagation or reexecution .

1 Introduction

In abstract interpretation of logic programming, two notions of repetition have lately gained interest [1, 2, 13]. The motivation comes from studying naive dataflow analysis of programs such as the following:

$$\begin{array}{lll} q(x,y) & \leftarrow & p(x,y), \textcircled{a}\ r(x), \textcircled{b}\ s(y) \\ p(u,u) & & \\ r(a) & & \\ s(v) & & \end{array}$$

Suppose we are interested in determining which variables are inevitably ground at the various program points. A naive analysis will deduce that x will become ground, but fail to deduce that s will therefore be called with a ground argument. Assuming the naive analysis works by annotating the program with sets of variables that are inevitably ground, the annotations will be: At point ⓐ $\emptyset$ and at point ⓑ $\{x\}$. The fact that x and y are aliases is hidden.

However, assume that at point ⓑ we decide to repeat the analysis of $p(x,y)$. The fact that x is ground can now easily be used to deduce that y is ground, and so we can improve the result at point ⓑ and make the annotation $\{x,y\}$. Even though $p(x,y)$ will in fact never be invoked with a ground first argument, this "repetition" based reasoning is valid. We discuss the idea in more detail in Section 3.

M. Bruynooghe originally suggested the repetition principle in the context of logic programming, referring to it as the "repeat previous call" strategy [2, 3]. Le Charlier and Van Hentenryck have explained the method more precisely and implemented dataflow analyses employing what they prefer to call "reexecution" [13], and recently Bruynooghe and Janssens [4] have suggested a notion of "propagation."

The idea behind propagation is that one can repeat[1] any "abstract unification" between the root and the current node of the execution tree. Bruynooghe and Janssens observe that in this setting the "abstract operations" need not be correct in the classical sense. By repeating an incorrect operation sufficiently often, the result may well end up being correct! This idea is very interesting and in this paper we attempt to make it precise by giving a continuations semantics based on closure operators which captures propagation. This allows us to address the correctness conjecture made by Bruynooghe and Janssens. We give general conditions under which propagation is correct.

[1] We use the term "repetition" for the act of repeating analysis of a literal, so "reexecution" and "propagation" will always have their technical meanings: they are two *methods* that both employ "repetition."

Bruynooghe and Janssens say of propagation:

> "The advantage is that one can use simpler abstract domains and simpler operations, which, when only applied locally, would yield incorrect results. ... For example, one can perform mode analysis with a very simple abstract domain not including any information about sharing and obtain results which appear as precise as with [approaches using more complex domains]."

We later show that the last statement is true only in certain circumstances: We exhibit programs for which the analysis proposed by Bruynooghe and Janssens is not as accurate as those produced by the other approaches referred to. A certain loss of precision is inherent in dataflow analyses using propagation when the description domains are not "downwards closed." Freeness and mode analyses are in this category.

The loss of precision is avoided when domains are downwards closed. We show that in this case, propagation can only improve accuracy. The most important examples of downwards closed domains are those used for groundness analysis. We investigate classical approaches to groundness analysis based on Boolean functions and compare their power with approaches using propagation. Contrary to statements in the literature, an analysis based on *positive* Boolean functions is strictly more precise than one using propagation and the simple groundness descriptions from the example above. We show that the latter corresponds to using the less expressive class of *definite* Boolean functions.

In Section 2 we recapitulate the idea of abstract interpretation of (constraint) logic programs. In Section 3 we discuss "repetition" and propose a precise definition of propagation. In Section 4 we offer a critique of these techniques. In Section 5 we consider various domains for groundness analysis and compare the classical approaches to groundness analysis with those based on propagation.

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 obtained at ⓧ 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. Let us denote the set of all constraints by Eqn.

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" then we may obtain a dataflow analysis which terminates in finite time. This is the idea behind abstract interpretation of logic programs [2, 3, 7, 8, 11, 16].

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 maps ⓐ to $\{u = a, u = f(a), u = f(f(a)), \ldots\}$ and ⓑ to $\emptyset$. A cruder approximation to this maps to ⓐ 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 $\{ⓐ \mapsto \{u\}, ⓑ \mapsto \{u\}\}$ is a correct approximation. Though not very precise, it still tells us that every spawned query will have a ground first argument. ■

From this point of view, a dataflow analysis is an approximation to the collecting semantics. P. and R. Cousot have formalized that idea as follows. The standard domain Y ($\mathcal{P}\ Eqn$ in the example) and the description domain X ($\mathcal{P}\ Var$ in the example) should be complete lattices related by a pair of *adjoined* functions. That is, there should be monotonic functions $\alpha : Y \to X$ and $\gamma : X \to Y$ such that

$$\forall\, x \in X\ .\ \forall\, y \in Y\ .\ \alpha\ y \leq_X x \Leftrightarrow y \leq_Y \gamma\ x.$$

The intention is that $\alpha\ E$ is the most precise approximation that correctly describes every constraint in E, while γ is the "semantic function" for approximations. In Example 2.1 we have

$$\begin{array}{lcl} \alpha\ E & = & \{v \mid \text{every } e \in E \text{ grounds } v\} \\ \gamma\ V & = & \{e \mid e \text{ grounds every } v \in V\}. \end{array}$$

A formal semantics is given by laying down the appropriate semantic domains and specifying certain operators on these domains. For example, in our case an important operation is $comb : \mathcal{P}\ Eqn \to \mathcal{P}\ Eqn \to \mathcal{P}\ Eqn$ which takes two sets of constraints and forms the possible conjunctions:

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

The notion of approximation is made precise 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 $\forall\, x \in X\ .\ \forall\, y \in Y\ .\ x \propto y \Rightarrow (f\ x) \propto (g\ y)$.

An important feature of P. and R. Cousot's "adjoined" framework is that given an operation on the standard domain, there is a (unique) best operation on the description domain which approximates it. Here "best" is with respect to precision. We shall refer to such "best" operations as the *induced* operations. To follow up the *comb* example, the best approximation to *comb* is $\lambda\ V\ V'\ .\ V \cup V'$.

We now give a precise definition of the standard semantics. In Section 3 we give a similar definition of the "propagation" semantics. Interpreters and compilers for (constraint) logic programming languages are usually based on SLD resolution, and for this reason, both of the semantic definitions capture this style of top-down, left-to-right atom reduction.

Predicates in a (constraint) logic program are divided into two classes: the *primitive constraints*, *Prim*, and the programmer-defined *atoms*, *Atom*. Primitive constraints are predefined in the sense that they have an intended meaning or interpretation which, for efficiency, is built into the solver for the language. We will typically use Horn clause programs in examples, and in this case, the primitive constraints are term equations. For simplicity we require that atoms have the form $p(x_1, .., x_n)$ where the x_i are distinct variables. A *literal* is an atom or a primitive constraint. A *constraint* is a (possibly existentially quantified) conjunction of primitive constraints. We only consider constraints modulo logical equivalence. We let *Eqn* denote the set of all constraints. We let $\overline{\exists}_S e$ denote the constraint e restricted to the variables in S. That is, $\overline{\exists}_S e$ is $\exists\, V_1 \exists\, V_2 ... \exists\, V_n e$ where $\{V_1, V_2, ..., V_n\} = vars\ e \setminus vars\ S$ and the function *vars* takes a syntactic object and returns the set of (free) variables occurring in it.

A *(constraint logic) program* is a finite set of clauses of the form $H \leftarrow B$, where the *head*, H, is an atom and the *body*, B, is a sequence of literals. A *goal* is a (possibly empty) sequence of literals. Let *Goal* be the set of goals, *Clause* the set of clauses, *Lit* the set of literals and *Prog* the set of constraint logic programs.

A *renaming* is an involution on variables. We naturally extend renamings to mappings between atoms, clauses, and constraints. The set of renamings is denoted by *Ren*. Syntactic objects s and s' are said to be *variants* if there is a renaming ρ such that $\rho\ s = s'$. The *definition of an atom A in program P with respect to variables W*, $defn_P\ A\ W$, is the set of variants of clauses in P such that each variant has A as a head and has variables disjoint from $W \setminus vars\ A$.

The operational semantics of a program is in terms of "answers" to its "derivations" which are reduction sequences of "states" where a state is a tuple consisting of the current constraint and the current literal sequence, or "goal". A *derivation* of state s for program P is a sequence of states $s_0 \to \cdots \to s_n$ where $s = s_0$ and there is a reduction from s_i to s_{i+1} where state $\langle e, L : G\rangle$ can be *reduced* as follows:

1. If $L \in Prim$ and $(L \wedge e)$ is satisfiable, then $\langle e, L : G\rangle \to \langle L \wedge e, G\rangle$;
2. If $L \in Atom$, then $\langle e, L : G\rangle \to \langle e, B :: G\rangle$, where $[\![L \leftarrow B]\!] \in defn_P\ L\ (vars(G) \cup vars(\theta))$

(We let :: denote concatenation of sequences.)

A derivation is *successful* when the last state has an empty sequence of atoms. Let e be the constraint in the last state of a successful derivation from a state s. The constraint e restricted to the variables in s is said to be an *answer* to state s. We denote the set of answers to state s by $answer(s)$. By the answers to a goal G we mean the answers to the state $\langle true, G\rangle$.

The following definition captures the essence of SLD refutation-based analyses using a standard computation rule and a parallel search rule. It is best understood by first considering the case when the description domain X is simply $\mathcal{P}\ Eqn$. It is defined in terms of three auxiliary functions which are induced from the following "concrete" functions:

Definition. As already mentioned, the function $comb : \mathcal{P}\ Eqn \to \mathcal{P}\ Eqn \to \mathcal{P}\ Eqn$ is defined by

$comb\ E\ E' = \{e \wedge e' \mid e \in E \text{ and } e' \in E'\} \setminus \{false\}$.

$add : Prim \to \mathcal{P}\ Eqn \to \mathcal{P}\ Eqn$ is defined by

$add\ p\ E = comb\ \{p\}\ E$.

$echo : Atom \to Atom \to \mathcal{P}\ Eqn \to \mathcal{P}\ Eqn$ is defined by

$echo\ A\ H\ E =$
$\{\rho\ (\overline{\exists}_H e) \mid \rho \in Ren, \rho\ H = A, \text{and } e \in E\}$. ■

The reasons for the names are: The function *comb* "combines" two sets of constraints by producing all possible conjuncts. The set $add\ p\ E$ is the result of conjoining the primitive constraint p to each of the set E of constraints. The function $echo\ A\ H$ takes a set of constraints E, each of which constrains variables in H and "echoes" these in terms of the variables in A.

Example 2.2 Let $A = p(x, y)$, $H = p(u, v)$, and $E = \{x = z \wedge y = z\}$. Then

$$echo\ H\ A\ E = \{u = v\}. \quad \blacksquare$$

Definition. A *standard semantics* for description domain X has semantic domain:

$$Den \quad = \quad Atom \to X \to X$$

and semantic functions:

$$\begin{array}{lcl} \mathbf{P}_X & : & Prog \to Den \\ \mathbf{C}_X & : & Clause \to Den \to Den \\ \mathbf{B}_X & : & Body \to Den \to X \to X \\ \mathbf{L}_X & : & Lit \to Den \to X \to X. \end{array}$$

It is defined by:

$$\begin{array}{l} \mathbf{P}_X\ [\![P]\!] = lfp\ (\bigsqcup_{C \in P}(\mathbf{C}_X\ [\![C]\!])) \\ \mathbf{C}_X\ [\![H \leftarrow B]\!]\ d\ A\ x = \\ \quad comb_X\ x\ (echo_X\ A\ H\ (\mathbf{B}_X\ [\![B]\!]\ d\ (echo_X\ H\ A\ x))) \\ \mathbf{B}_X\ [\![nil]\!]\ d\ x = x \\ \mathbf{B}_X\ [\![L : B]\!]\ d\ x = \mathbf{B}_X\ [\![B]\!]\ d\ (\mathbf{L}_X\ [\![L]\!]\ d\ x) \\ \mathbf{L}_X\ [\![L]\!]\ d\ x = d\ L\ x \quad \text{when } L \in Atom \\ \mathbf{L}_X\ [\![L]\!]\ d\ x = add_X\ L\ x \quad \text{when } L \in Prim \end{array}$$

in terms of the monotonic auxiliary functions

$$\begin{array}{lcl} comb_X & : & X \to X \to X \\ add_X & : & Prim \to X \to X \\ echo_X & : & Atom \to Atom \to X \to X. \quad \blacksquare \end{array}$$

$\mathbf{P}_X$ is well-defined [16]. The standard semantics is *safe* if the auxiliary functions $comb_X$, add_X and $echo_X$ approximate the functions *comb*, *add* and *echo* respectively. The standard semantics *induced* for X is the semantics obtained by inducing the auxiliary functions from *comb*, *add*, and *echo* respectively. Note that the induced semantics is completely defined by the description domain X. Furthermore, safeness suffices to ensure correctness of the resulting dataflow analysis [16]:

Theorem 2.1 Let $\mathbf{P}_X$ be a safe standard semantics. Let x be a constraint description and s be the state $\langle e, [A] \rangle$. If $x \propto \{e\}$, then

$$(\mathbf{P}_X\ [\![P]\!]\ A\ x) \propto (answer\ s). \quad \blacksquare$$

The above semantic equations specify a dataflow analysis. The standard way to implement the analysis is by means of memoization or tabulation in which the least fixpoint is reached via the Kleene sequence and in which only those values of a denotation actually required are computed. In such an implementation, termination is guaranteed, provided the constraint descriptions are "almost Noetherian."

Definition. The description domain X is *almost Noetherian* iff there do not exist infinite chains $\{E_1, E_2, \ldots\} \subseteq \mathcal{P}\ Eqn$ and $\{x_1, x_2, \ldots\} \subseteq \mathcal{P}\ X$ such that $vars\ (\bigcup E_i)$ is finite and for all i, $x_i \propto E_i$. $\blacksquare$

3 Propagation

Consider the program

$$\begin{array}{l} q(u, v) \leftarrow p(u, v), r(u) \\ p(u, u) \\ r(a) \end{array}$$

together with the query $\leftarrow q(x, y)$ and suppose that we are interested in which variables will inevitably be ground. A simplistic analysis will proceed as follows: The call to q means that p will be called with neither variable bound. Thus p returns with neither variable ground. Then r binds u to a constant and we conclude that u is ground.

In fact an execution will ground both u and v since they are aliases by the time one is ground. If we repeat, that is, consider the query $\leftarrow q(x, y), q(x, y)$ rather than its single-atom version, then we will find that both variables must in fact be ground. This is because p is now called with u being ground. This motivated Bruynooghe [2, 3] to propose repetition (or "repeat previous call") as a device for improving the accuracy of dataflow analyses for logic programs.

Bruynooghe and Janssens [4] subsequently suggested that the idea could be generalized. With *propagation* repetition is not restricted to literals in the current clause body: any abstract operation on the path to the current node in the execution tree may be redone. Surprisingly, propagation relaxes the demand that abstract operations be "safe," that is, they only need to be correct under an assumption that there is no sharing of variables. It then relies on the repetition of (some of) the unifications already performed on the current execution path to compensate for this lack of correctness.

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

$$\begin{array}{lcl} q(x,y) & \leftarrow & x = y, p(x,y) \\ p(u,v) & \leftarrow & u = a^{ⓐ} \\ p(u,v) & \leftarrow & v = a^{ⓑ} \end{array}$$

A groundness analysis which disregards aliasing will arrive at the annotation $\{ⓐ \mapsto \{u\}, ⓑ \mapsto \{v\}\}$ and hence be unable to infer anything about the groundness of x and y. However, if propagation is applied at ⓐ, the repetition of $x = y, x = u, y = v$ will result in the annotation $\{u, v\}$ at ⓐ and similarly at ⓑ. Thus it will be concluded that both of x and y will become ground. ■

The "abstract unification" was correct in the classical sense, and so the first result at ⓐ—which was later improved—was correct. The next example shows how even incorrect "abstract unification" may work. Call a substitution θ a *unifier* of e iff $\models (\theta\ e)$ and let $unif\ e$ denote the set of unifiers of e.

Example 3.2 Consider the query and program from before, but this time let us perform a *freeness* analysis. An annotation will be a set of variables, those known to be *free*, that is, bound to a variable. Let us define "abstract unification" as follows:

$$add_{Free}\ p\ x = x \cap \{V \in Var \mid (\theta\ V) \text{ is a variable when } \theta \in unif\ p\}.$$

We now initially get the annotation $\{v\}$ at ⓐ, which is not correct. However, after (repeated) repetition of $x = y, x = u, y = v$ we get the correct annotation, $\emptyset$. ■

Bruynooghe and Janssens [4] conjecture that the propagation approach is valid. Here we offer conditions under which this is true. While they sketch the underlying idea, Bruynooghe and Janssens do not give a formal definition of propagation. A contribution of this paper is a denotational definition of the propagation semantics. Propagation is captured in terms of a "continuation" which is the semantic function associated with the literals previously met in a derivation. The idea is that literals and clauses are now continuation transformers rather than simple constraint transformers.

Repetition is captured by requiring that the continuations are closure operators—repetition corresponds to computation of the closure. A technical subtlety is that the continuations are closure operators with respect to an "instantiation" ordering $\preceq$ rather than the usual "is more general than" ordering $\leq$ on the abstract domain.

Definition. Let $(X, \preceq)$ be a poset. A function $F : X \to X$ is *monotonic* (with respect to $\preceq$) iff $\forall\, x, x' \in X\ .\ x \preceq x' \Rightarrow F\ x \preceq F\ x'$. F is *idempotent* iff $F = F \circ F$. F is *increasing* iff $\forall\, x \in X\ .\ x \preceq F\ x$. We let $X \xrightarrow{\text{inc}} X$ denote the set of increasing functions in $X \to X$. F is a *closure operator* iff F is monotonic, idempotent, and increasing.

Let $F : X \to X$ be an operator on a complete lattice X. Then $F^\star : X \to X$ is defined by

$$F^\star\ x = \bigsqcap \{x' \mid x' = F\ x' \text{ and } x \preceq x'\}.$$ ■

Proposition 3.1 $F^\star$ is a closure operator. ■

The relevance for repetition is that for increasing F, $F^\star = F \circ \ldots \circ F$. Notice that if X is a complete lattice then $X \xrightarrow{\text{inc}} X$ is a complete sublattice of $X \to X$.

Definition. Let X be a complete lattice. With $F : X \to X$ we define

$$F^\alpha = \begin{cases} \bigsqcup_{\alpha' < \alpha} F^{\alpha'} & \text{if } \alpha \text{ is a limit ordinal} \\ F \circ F^{\alpha - 1} & \text{if } \alpha \text{ is a successor ordinal.} \end{cases}$$ ■

Note that for $F : X \xrightarrow{\text{inc}} X$, $F^0\ x = x$.

Proposition 3.2 If $F : X \xrightarrow{\text{inc}} X$ is monotonic then there is an ordinal α such that $F^\star = f^\beta$ for all $\beta \geq \alpha$. ■

Definition. Let X be a complete lattice with respect to $\leq$ and also with respect to $\preceq$. The *propagation semantics* for description domain X has semantic domains:

$$\begin{array}{lcl} Cont & = & X \to X \\ Den' & = & Atom \to Cont \to Cont \end{array}$$

and semantic functions:

$$\begin{array}{lcl} \mathbf{Q}_X & : & Prog \to Atom \to Cont \\ \mathbf{P}'_X & : & Prog \to Den' \\ \mathbf{C}'_X & : & Clause \to Den' \to Den' \\ \mathbf{B}'_X & : & Body \to Den' \to Cont \to Cont \\ \mathbf{L}'_X & : & Lit \to Den' \to Cont \to Cont. \end{array}$$

It is defined by:

$$\begin{array}{l} \mathbf{Q}_X\ [\![P]\!]\ A = \mathbf{P}'_X\ [\![P]\!]\ A\ id_x \\ \mathbf{P}'_X\ [\![P]\!] = lfp\ (\bigsqcup_{C \in P} (\mathbf{C}'_X\ [\![C]\!])) \\ \mathbf{C}'_X\ [\![H \leftarrow B]\!]\ d\ A\ \kappa = \\ \quad comb'_X\ \kappa\ (echo'_X\ A\ H\ (\mathbf{B}'_X\ [\![B]\!]\ d\ (echo'_X\ H\ A\ \kappa))) \\ \mathbf{B}'_X\ [\![nil]\!]\ d\ \kappa = \kappa \\ \mathbf{B}'_X\ [\![L : B]\!]\ d\ \kappa = \mathbf{B}'_X\ [\![B]\!]\ d\ (\mathbf{L}'_X\ [\![L]\!]\ d\ \kappa) \\ \mathbf{L}'_X\ [\![L]\!]\ d\ \kappa = d\ L\ \kappa \text{ when } L \in Atom \\ \mathbf{L}'_X\ [\![L]\!]\ d\ \kappa = add'_X\ L\ \kappa \text{ when } L \in Prim. \end{array}$$

The auxiliary functions

$$\begin{array}{lcl} comb'_X & : & Cont \to Cont \to Cont \\ add'_X & : & Prim \to Cont \to Cont \\ echo'_X & : & Atom \to Atom \to Cont \to Cont \end{array}$$

are defined by

$$\begin{array}{lcl} comb'_X\ \kappa\ \kappa' & = & (\lambda\, x\, .\, comb_X\ (\kappa\ x)\ (\kappa'\ x))^\star \\ add'_X\ p\ \kappa & = & ((add_x\ p) \circ \kappa)^\star \\ echo'_X\ A\ H\ \kappa & = & (echo_X\ A\ H) \circ \kappa \circ (echo_X\ H\ A). \end{array}$$

Here $id_X : X \to X$ denotes the identity function. The least fixpoint operator is with respect to $\leq$ but the closure $F^\star$ is with respect to $\preceq$. The functions $comb_X$, add_X and $echo_X$ are required to be monotonic with respect to both $\leq$ and $\preceq$ and increasing with respect to $\preceq$. ■

Proposition 3.3 $\mathbf{Q}_X$ is well-defined. ■

When discussing safeness it is useful to think of continuations as implicitly describing the sets of equations, rather than the closure operators over sets of equations which they explicitly describe.

Definition. The closure $\kappa : X \to X$ *implicitly approximates* $E \in \mathcal{P}\, Eqn$, written $\kappa\ \tilde{\propto}\ E$ iff $\kappa \propto (comb\ E)$. We extend the relation to function spaces in the usual way: $f : X \to X'\ \tilde{\propto}\ g : Y \to Y'$ iff $\forall\, x \in X\, .\, \forall\, y \in Y\, .\, x\ \tilde{\propto}\ y \Rightarrow (f\ x)\ \tilde{\propto}\ (g\ y)$.

The propagation semantics is *safe* iff $comb'_X\ \tilde{\propto}\ comb$, $add'_X\ \tilde{\propto}\ add$, and $echo'_X \propto echo$. ■

The requirements for safeness are less demanding than in the standard semantics, as correctness of add_X and $comb_X$ is only required after taking the closure. However this generality comes at a cost. We can no longer prove correctness of add_X and $comb_X$ directly, but must rather prove that the more complex functions add'_X and $comb'_X$ are correct.

Theorem 3.4 Let $\mathbf{P}_{\mathcal{P}\,Eqn}$ and $\mathbf{Q}_{\mathcal{P}\,Eqn}$ be the standard semantics and propagation semantics induced from $\mathcal{P}\, Eqn$. Then

$$(\mathbf{P}_{\mathcal{P}\,Eqn}\ [\![P]\!]\ A\ E) \subseteq (\mathbf{Q}_{\mathcal{P}\,Eqn}\ [\![P]\!]\ A\ E).$$ ■

Proposition 3.5 Let $\mathbf{P'}_{\mathcal{P}\,Eqn}$ be induced from $\mathcal{P}\, Eqn$. If $\mathbf{P'}_X$ is safe then $\mathbf{P'}_X \propto \mathbf{P'}_{\mathcal{P}\,Eqn}$. ■

Theorem 3.6 Let $\mathbf{Q}_X$ be a safe propagation semantics for X. Let x be a constraint description and s be the state $\langle e, [A]\rangle$. If $x \propto \{e\}$, then $(\mathbf{Q}_X\ [\![P]\!]\ A\ x) \propto (answer\ s)$. ■

$\mathbf{Q}_X$ specifies a dataflow analysis which uses propagation. The analysis could be implemented using memoization or tabulation in which computation of the least fixpoint is by way of the Kleene sequence and in which only those values (and closures) of a function actually required are computed. Such a "call-by-need" implementation we believe corresponds to the operational semantics sketched by Bruynooghe and Janssens.

In such an implementation, termination is guaranteed if the description domain is "almost Noetherian" for the usual ordering, as required in the standard semantics, and, in addition, "almost Noetherian" for the "lifted instantiation ordering." This is the Egli-Milner ordering on $\mathcal{P}\, Eqn$ where the underlying ordering on Eqn is given by logical consequence:

Definition. The *lifted instantiation ordering*, $\preceq_{inst}$, on $\mathcal{P}\, Eqn$ is defined by $X \preceq_{inst} Y$ iff $\forall\, x \in X\, .\, \exists\, y \in Y\, .\, x \models y$ and $\forall\, y \in Y\, .\, \exists\, x \in X\, .\, x \models y$. ■

Definition. The description domain X is *almost Noetherian for the lifted instantiation ordering* iff there do not exist infinite sets $\{E_1, E_2, \ldots\} \subseteq \mathcal{P}\, Eqn$ and $\{x_1, x_2, \ldots\} \subseteq \mathcal{P}\, X$ such that

- for all i, $E_{i+1} \preceq_{inst} E_i$,
- for all i, $x_{i+1} \preceq x_i$,
- $vars\ (\bigcup E_i)$ is finite, and
- for all i, $x_i \propto E_i$. ■

Using Theorem 3.6 and this termination condition one can prove that the mode analysis given by Bruynooghe and Janssens [4] is correct and terminating.

The freeness analysis is well-defined because the ordering on descriptions is coherent with the usual instantiation ordering. For an example where this is not the case, assume that we are interested in *linearity*, that is, whether the terms a variable may be bound to are guaranteed to have no repeated variable occurrences. Consider the query

$$\leftarrow \textcircled{a}\, x = f(y, y), \textcircled{b}\, x = f(a, a) \textcircled{c}$$

At ⓐ x is linear, but at ⓑ x has changed status to "any." At ⓒ x has changed back to "linear." Because of the change, we should repeat $x = f(y, y)$, but then we can only assume that x becomes "any." It would not be correct to assume that a linear x would remain linear after a unification $x = f(y, y)$. But then, at ⓒ, x oscillates back to "linear," and so on, without limit.

4 A critique of propagation

We now compare the accuracy of analyses based on the propagation semantics with those based on the standard semantics.

We might hope that the inclusion relationship in Theorem 3.4 is in fact equality. But this is not the case, as shown in the following example; propagation may actually *decrease* precision, and is in a sense inherently less precise. Consider the program

$p(u, u)$
$p(a, u)$

The query $\leftarrow p(x, y)$ has two answers: $x = y$ and $x = a$. However, the query $\leftarrow p(x, y), p(x, y)$ has an additional answer, namely $x = a \wedge y = a$. Propagation therefore does not preserve a semantics that involves answer substitutions.

In terms of dataflow analysis, the same example shows how propagation may decrease precision. Suppose we want to investigate which variables are *free* after the query $\leftarrow p(x, y)$ has been executed. A simple analysis based on abstract interpretation will tell us that x may be bound, but y will remain free. However, the same analysis technique applied to the query $\leftarrow p(x, y), p(x, y)$ will give as result that both variables may be bound to constants. This loss of precision is an unfortunate aspect of a technique which was suggested as a means for *improving* the precision of dataflow analyses.

It could be objected that one should not perform propagation unconditionally, but only when the input to previous unifications have *changed* [4]. That does not, however, remove the possibility of unnecessary imprecision. The following examples show two different ways in which precision can be lost in cases where absence of propagation would have preserved it. Both examples use the "term descriptions" *ground*, *free*, and *any* with the ordering $t \leq u$ iff $t = u \vee u = any$ and assume a query $\leftarrow q(x, y)$.

Example 4.1 Consider the program

$q(x, y) \leftarrow p(x, y, z), ⓐ\ z = b, ⓑ\ r(y)$
$p(u, u, v)$
$p(b, u, v)$

Initially we get the (correct and precise) annotation

ⓐ : $\{x \mapsto any, y \mapsto free, z \mapsto free\}$
ⓑ : $\{x \mapsto any, y \mapsto free, z \mapsto ground\}$

Since z is in the call to p and its mode has changed, $p(x, y, z)$ must be repeated. We then get

ⓑ : $\{x \mapsto any, y \mapsto any, z \mapsto ground\}$

which is less precise than necessary, since r will in fact always be called with y free. ■

Example 4.1 is arguably somewhat pathological. However, the following example is not, and suggests that propagation is inadequate for mode analysis. It shows that the apparent power of propagation (that it allegedly compares with, say, EXP [5]) is only apparent.

Example 4.2 Consider the program

$q(x, y) \leftarrow x = f(y, z), ⓐ\ z = b, ⓑ\ r(y)$

Again we initially get

ⓐ : $\{x \mapsto any, y \mapsto free, z \mapsto free\}$
ⓑ : $\{x \mapsto any, y \mapsto free, z \mapsto ground\}$

As before the mode of z has changed, and propagation of $x = f(y, z)$ yields

ⓑ : $\{x \mapsto any, y \mapsto any, z \mapsto ground\}$

which is inaccurate, as r will always be called with y free. ■

A more subtle problem with propagation is that a Prolog construct such as $var(\ldots)$ becomes difficult to handle. Even though $var(\ldots)$ is non-logical, the standard approaches to mode analysis can easily handle a program such as

$q(x) \leftarrow p(x), ⓐ\ x = f(y)$
$p(x) \leftarrow var(x)$

The usual approach is simple: At point ⓐ x must be free, since the meaning of $var(x)$ is a function that takes any annotation and changes the x entry to $x \mapsto free$. Now, with propagation $p(x)$ should be repeated once $x = f(y)$ has been processed. But that would make the status of x oscillate from *free* to *any* to *free* which is not a correct result.

One way to remove the problem with the accuracy of propagation is to use a closure operator which uses the glb operator to combine the result of the repetition with the previous result. This technique is used in [13]. However, this means that for overall correctness the result at each stage in the repetition must be correct, and so is not in the spirit of Bruynooghe and Janssens' approach in which correctness is only guaranteed after performing the repetition exhaustively.

The most severe drawback of propagation, however, does not relate to precision. It is the fact that

there is no simple correctness criterion for the abstract operations. In classical abstract interpretation, the correctness problem is reduced to establishing correctness of a few basic operations. Their correctness will automatically ensure correctness of the composite dataflow analysis. This reduction is at the heart of abstract interpretation.

Propagation, however, does not seem to have this crucial feature. The freeness example taught us that operations need not be correct in the sense of abstract interpretation, and yet they must be "sufficiently correct" in some sense: If they alway return the smallest description, for example, no amount of propagation will yield correct results.

Looking at the freeness example, one may suspect that "sufficiently correct" means "correct under the assumption that no variables share." This criterion, however, is not generally valid. To see this, consider again the case of linearity analysis. If x, y, and z are all linear, then, under the assumption of no sharing, $z = f(x, y)$ should preserve linearity of all three. But an "abstract unification" which takes $\{x, y, z\}$ and returns $\{x, y, z\}$ is not correct, witness the query

$$\leftarrow x = y, z = f(x, y)$$

No amount of repetition would find that z becomes nonlinear, so the only correct approach to abstractly unifying $x = f(x, y)$ is to be pessimistic and say that z is no longer definitely linear. It appears that no simple local correctness criterion exists, and that proving correctness in the context of propagation is extremely complicated.

The objections listed above only apply to certain classes of dataflow analyses. There is one case where propagation may be useful: For "downwards closed" domains, propagation can only improve accuracy.

Definition. [16] A description domain X (with associated $\gamma : X \to \mathcal{P}\, Eqn$) is *downwards closed* iff for all $x \in X$ and $e, e' \in Eqn$, $(e \in \gamma\ x \wedge e' \models e)$ implies $e' \in \gamma\ x$. ■

Examples of downwards closed domains are those typically used in groundness analysis, type analysis, definite aliasing and definite sharing analysis.

Theorem 4.1 Let $\mathbf{P}_X$ and $\mathbf{Q}_X$ be the standard semantics and propagation semantics induced from the downwards closed description domain X. Then $\mathbf{P}_X \propto \mathbf{Q}_X$. ■

In Section 5 we introduce some downwards closed domains and compare the corresponding groundness analyses with those obtained using propagation.

5 Groundness analysis using Boolean functions

A variable in a logic programming language is very different from a variable in an imperative or functional programming language. It is sometimes referred to as a "logical variable" and characterized as "constrain-only." This is because execution of a logic program proceeds by steps that continually narrow the set of possible values that a variable may take.

This suggests the possibility of propagating conditional invariants of the form "*From this point on*, if x has (ever gets) property p, then y has (will have) property r." A statement such as "x is ground" may be represented by a propositional variable x. Groundness (or other) dependencies may then be represented by Boolean functions, such as that denoted by $y \to x$.

5.1 The domain *Pos* (nee *Prop*)

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 conveys no information at all, that is, exactly the information conveyed by the function *true*.

Definition. A *Boolean function* is a function $F : Bool^n \to Bool$. 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." The set of propositional variables $\{x_1, \ldots, x_n\}$ will be referred to as $Pvar$. 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. We sometimes refer to the formulas as *groundness dependency 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. (Here and in the following we use the notation $\bigwedge\{\phi_1, \ldots, \phi_n\}$ for the formula $\phi_1 \wedge \ldots \wedge \phi_n$, and similarly for $\bigvee$.) *Pos* was suggested for groundness analysis by Marriott and Søndergaard [14] (under the less suggestive name '*Prop*') and further studied by Cortesi *et al.* [6]. *Pos* has the desirable property of being "condensing" which allows for a very fast, yet precise analysis [15].

One can imagine many types of properties of dataflow information for which dependency formulas is a useful formalism. Here we are concerned with groundness. As an example, if the constraint $x = f(y, z)$ is generated during query evaluation, the relationship $x \leftrightarrow (y \wedge z)$ is deduced. Informally the formula says that if x is (becomes) ground then so is (does) both of y and z, and *vice versa*.

Example 5.1 The Boolean functions $[x \leftrightarrow (y \wedge z)]$ and $[x \rightarrow y]$ are in *Pos*. The functions $[\neg x]$ (negation) and $[(x \vee y) \wedge \neg(x \wedge y)]$ (exclusive or) are not in *Pos*. ■

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 the domain $Pos_- = Pos \cup \{false\}$, ordered by logical consequence.

The idea with using Pos_- is that an constraint e is described by $\phi \in Pos_-$ 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 $grounds\ \theta$ be the truth assignment which maps a variable to true if θ grounds the variable and to false otherwise. That is, $grounds : Sub \rightarrow Var \rightarrow Bool$ is defined by

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

The function $\gamma : Pos_- \rightarrow (\mathcal{P}\ Eqn)$ is defined by

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

This γ maps $\phi \in Pos_-$ to the set of constraints e which have the property that, no matter how they further become constrained to some e', the groundness formula corresponding to e' satisfies ϕ.

Example 5.2 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. ■

Lemma 5.1 [16] Pos_- is downwards closed. ■

The function *comb* is best approximated by conjunction [16]. The two other auxiliary functions have best approximations as follows:

Definition. Let $comb_{Pos} : Pos_- \rightarrow Pos_- \rightarrow Pos_-$ be defined by

$$comb_{Pos}\ \phi\ \phi' = \phi \wedge \phi',$$

$addPos : Prim \rightarrow Pos_- \rightarrow Pos_-$ by

$$add_{Pos}\ [\![x = t]\!]\ \phi = [x \leftrightarrow \bigwedge(vars\ t)] \wedge \phi,$$

and $echoPos : Atom \rightarrow Atom \rightarrow Pos_- \rightarrow Pos_-$ by

$$echo_{Pos}\ A\ H\ \phi = \text{if } \rho\ H = A \text{ for some } \rho \in Ren \text{ then } \rho\ (\exists_H\ \phi) \text{ else } [false].$$ ■

Example 5.3 We have

$$\begin{array}{lcl} add_{Pos}\ [\![x = nil]\!]\ [false] & = & [false] \\ add_{Pos}\ [\![x = nil]\!]\ [true] & = & [x] \\ add_{Pos}\ [\![y = z]\!]\ [x] & = & [x \wedge (y \leftrightarrow z)] \\ add_{Pos}\ [\![x = u : x']\!]\ [true] & = & [x \leftrightarrow (u \wedge x')]. \end{array}$$ ■

Example 5.4 Let P be the *append* program

$$\begin{array}{lcl} append(x, y, z) & \leftarrow & x = nil, y = z \\ append(x, y, z) & \leftarrow & x = u : x', \\ & & z = u : z', \\ & & append(x', y, z') \end{array}$$

Consider the query $A = \leftarrow append(x, y, z)$. To compute $\mathbf{P}_{Pos}\ [\![P]\!]\ A\ [true]$, the analysis proceeds as follows. Let $d_P = \bigsqcup_{C \in P}(\mathbf{C}_{Pos}\ [\![C]\!])$. We then have

$$\begin{array}{lcl} (d_P \uparrow 0)\ A\ [true] & = & [false] \\ (d_P \uparrow 1)\ A\ [true] & = & [x \wedge (y \leftrightarrow z)] \vee [false] \\ & = & [x \wedge (y \leftrightarrow z)] \\ (d_P \uparrow 2)\ A\ [true] & = & [x \wedge (y \leftrightarrow z)] \vee [(x \wedge y) \leftrightarrow z] \\ & = & [(x \wedge y) \leftrightarrow z] \end{array}$$

and $(d_P \uparrow 3) = (d_P \uparrow 2) = lfp\ \ d_P$. Thus $\mathbf{P}_{Pos}\ [\![P]\!]\ A\ [true] = [(x \wedge y) \leftrightarrow z]$. Similarly one can show that $\mathbf{P}_{Pos}\ [\![P]\!]\ A\ [z] = [x \wedge y \wedge z]$. ■

5.2 The domains *Con*, *Mon*, and *Def*

A number of subclasses of *Pos* serve well for groundness analysis.

Definition. Let *Def* be the class of positive Boolean functions whose models are closed under intersection. Let *Mon* be the class of monotonic Boolean functions (with respect to the ordering $\models$). Let *Con* be the intersection of *Def* and *Mon*. ■

Con has been used for groundness analysis by Jones and Søndergaard [11]. *Mon* has been used extensively in the strictness analysis of lazy functional programs [17]. It has a compact representation in terms of so-called frontiers. *Def* has been used by Dart in his work on dependency analysis for deductive databases [9, 10]. The reason for the names *Con* and *Def* are: A Boolean function is in *Con* iff it can be written as a conjunction. It is in *Def* iff it can be written as a definite propositional formula. The monotonic functions are exactly those that can be written in CNF.

All three domains are downwards closed, like *Pos*, and applicable as domains for groundness analysis. In Section 5.3 we discuss their power when used with or without propagation.

Con is isomorphic with the domain used for simple groundness analysis in previous sections: The annotation V (a set of variables) corresponds directly to the Boolean function $\bigwedge V$.

In the following it is understood that all versions of $comb_X$, add_X, and $echo_X$ are the induced versions.

5.3 The power of propagation in groundness analysis

Le Charlier and Van Hentenryck have implemented both a groundness analysis using reexecution based on *Con* [13] and one using $\mathbf{P}_{Pos}$ [12] and compared them for efficiency. They state that a groundness analysis using *Pos* is both efficient and precise, compared with reexecution. They also notice that reexecution based on *Con* is not as precise as an analysis based on *Pos*. Consider the program

$$\begin{array}{lcl} q(x,y) & \leftarrow & x = y, p(x,y) \\ p(u,v) & \leftarrow & u = a \\ p(u,v) & \leftarrow & v = a \end{array}$$

Here $\mathbf{P}_{Pos}$ will derive $[x \wedge y]$ as groundness information but reexecution based on *Con* will derive $[true]$. Le Charlier and Van Hentenryck explain that this is "because non local literals are not reexecuted inside a clause." That is, when processing the clause $p(a,u)$, we cannot repeat $x = y$, as that literal is in a different clause.

This point is correct, but it fails to demonstrate the true expressiveness of *Pos*, as one could argue that to achieve the desired precision, all that is needed is to allow for unrestricted repetition, as is done with propagation. What *is* demonstrated by the above example is the fact that groundness analysis based on *Def* is more powerful than one using reexecution based on *Con*. Hence one based on *Pos* is also more powerful and replacing reexecution by propagation makes no difference in the example.

But *Pos* is strictly more powerful than *Def*, and a simple change to the program from before shows this:

$$\begin{array}{lcl} q(x,y) & \leftarrow & p(x,y), \textcircled{a}\ x = y \\ p(u,v) & \leftarrow & u = a \\ p(u,v) & \leftarrow & v = a \end{array}$$

Here $\mathbf{P}_{Def}$ will derive $[x \leftrightarrow y]$ as groundness information. This is because the best *Def* annotation at ⓐ is $[true]$. $\mathbf{P}_{Pos}$, however, will again derive the more precise $[x \wedge y]$ since the annotation at ⓐ is $[x \vee y]$, which conjoined with $[x \leftrightarrow y]$ yields $[x \wedge y]$.

Showing that $\mathbf{P}_{Def}$ is more precise than $\mathbf{P}_{Con}$ is also easy. Consider

$$q(x,y) \quad \leftarrow \quad x = y, x = a$$

Here $\mathbf{P}_{Con}$ will derive x as groundness information whereas $\mathbf{P}_{Def}$ will derive the more precise $x \wedge y$.

The above examples demonstrate how some approaches are more precise than others. The following definition enables us to make precise statements about approaches being comparable in power.

Definition. Semantic function F is *as precise as* G with respect to query language Q iff

$$F\ [\![P]\!]\ A\ x \leq G\ [\![P]\!]\ A\ x \text{ for all } x \in Q.$$

We write this $F \overset{Q}{\leq} G$. F and G are *equivalent* with respect to Q iff $F \overset{Q}{\leq} G$ and $G \overset{Q}{\leq} F$. We write this $F \overset{Q}{\simeq} G$. F is *more precise than* G with respect to Q, written $F \overset{Q}{<} G$ iff $F \overset{Q}{\leq} G$ and $F \overset{Q}{\not\simeq} G$. ■

The next theorem says that (1) propagation does not improve an analysis that uses *Pos*; (2,3) propagation using *Def* or *Mon* is equivalent to classical analysis using *Pos*, except that *Pos* allows more expressive

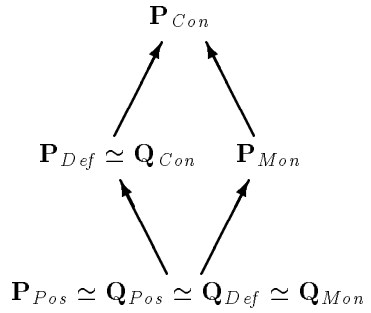

Figure 1: Groundness analyses: relative precision

queries; and (4) propagation using *Con* is equivalent to classical analysis using *Def*, except that *Def* allows more expressive queries.

Theorem 5.2 We have:

(1) $\mathbf{P}_{Pos} \overset{\text{Pos}}{\simeq} \mathbf{Q}_{Pos}$ (3) $\mathbf{P}_{Pos} \overset{\text{Mon}}{\simeq} \mathbf{Q}_{Mon}$

(2) $\mathbf{P}_{Pos} \overset{\text{Def}}{\simeq} \mathbf{Q}_{Def}$ (4) $\mathbf{P}_{Def} \overset{\text{Con}}{\simeq} \mathbf{Q}_{Con}$ ■

The following corollaries follow from Theorem 5.2 and previous examples.

Corollary 5.3 We have:

(1) $\mathbf{P}_{Def} \overset{\text{Con}}{<} \mathbf{P}_{Con}$ (5) $\mathbf{P}_{Pos} \overset{\text{Con}}{<} \mathbf{Q}_{Con}$

(2) $\mathbf{P}_{Mon} \overset{\text{Con}}{<} \mathbf{P}_{Con}$ (6) $\mathbf{Q}_{Con} \overset{\text{Con}}{<} \mathbf{P}_{Con}$

(3) $\mathbf{P}_{Pos} \overset{\text{Def}}{<} \mathbf{P}_{Def}$ (7) $\mathbf{Q}_{Def} \overset{\text{Def}}{<} \mathbf{P}_{Def}$

(4) $\mathbf{P}_{Pos} \overset{\text{Mon}}{<} \mathbf{P}_{Mon}$ (8) $\mathbf{Q}_{Mon} \overset{\text{Mon}}{<} \mathbf{P}_{Mon}$ ■

Note in particular (5): $\mathbf{P}_{Pos}$ is strictly more accurate than the (groundness component of) the mode analysis [4] using propagation. Figure 1 summarizes the results in the case of the query language *Con*, that is, an arrow $\mathbf{X} \rightarrow \mathbf{Y}$ reads $\mathbf{X} \overset{\text{Con}}{<} \mathbf{Y}$.

6 Conclusion

Propagation utilizes repetition to simplify the construction of dataflow analyses. However, if description domains are not "downwards closed," propagation may *decrease* accuracy unnecessarily. We have given examples showing that this may occur frequently. Furthermore, termination of propagation is not guaranteed, even for finite description domains.

Propagation generally does improve the accuracy of dataflow analyses based on "downwards closed" domains, such as those used for groundness analysis. Alternatively one can often incorporate the "continuation" capturing repetitions in the domain itself. In the (groundness) examples discussed here, this was done by modelling a continuation by implication in some fragment of propositional logic. We find the use of Boolean functions preferable, since propositional logic is so well understood, and the worst-case complexity of the resulting analysis seems much easier to determine.

Finally propagation excludes the principle of "reduction of correctness proof." In classical abstract interpretation one can prove a dataflow analysis correct by simply proving the basic "abstract" operations correct. This principle of proof reduction is at the heart of abstract interpretation, but lost with propagation in general.

References

[1] R. Barbuti and M. Martelli. A tool to check the non-floundering programs and goals. In P. Deransart, B. Lorho and J. Małuszyński, editors, *Programming Language Implementation and Logic Programming* (Lecture Notes in Computer Science 348), pages 58–67. Springer-Verlag, 1989.

[2] M. Bruynooghe. A framework for the abstract interpretation of logic programs. Report CW 62, Dept. of Computer Science, University of Leuven, Belgium, 1987.

[3] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming* **10** (2): 91–124, 1991.

[4] M. Bruynooghe and G. Janssens. Propagation: A new operation in a framework for abstract interpretation of logic programs. In A. Pettorossi, editor, *Meta-Programming in Logic* (Lecture Notes in Computer Science 649), pages 294–307. Springer-Verlag, 1992.

[5] A. Cortesi and G. Filé. Abstract interpretation of logic programs: An abstract domain for groundness, sharing, freeness and compoundness analysis. Proc. Symp. Partial Evaluation and Semantics-Based Program Manipulation, *Sigplan Notices* **26** (9): 52–61, 1991.

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

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Fourth Ann. ACM Symp. Principles of Programming Languages*, pages 238–252. Los Angeles, California, 1977.

[8] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming* **13** (2&3): 103–179, 1992.

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

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

[11] N. D. Jones and H. Søndergaard. 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] B. Le Charlier and P. Van Hentenryck. Groundness analysis for Prolog: Implementation and evaluation of the domain Prop. *Proc. Third ACM Symp. Partial Evaluation and Semantics-Based Program Manipulation*, pages 99–110. ACM Press, 1993.

[13] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog (extended abstract). In K. Apt, editor, *Logic Programming: Proc. Joint Int. Conf. Symp. Logic Programming*, pages 750–764. MIT Press, 1992.

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

[15] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. To appear in *ACM Letters on Programming Languages and Systems*.

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

[17] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph. D. Thesis, University of Edinburgh, Scotland, 1981.