A Program Transformation for Backwards Analysis of Logic Programs

Gallagher, John Patrick

Published in:
Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003

Publication date:
2003

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Gallagher, J. P. (2003). A Program Transformation for Backwards Analysis of Logic Programs. In M. Bruynooghe (Ed.), *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003* (pp. 92-105). Uppsala: Springer. Lecture Notes in Computer Science, Vol.. 3018

A Program Transformation for Backwards Analysis of Logic Programs

John P. Gallagher*

Roskilde University, Computer Science, Building 42.1
DK-4000 Roskilde, Denmark
e-mail: jpg@ruc.dk

Abstract. The input to backwards analysis is a program together with properties that are required to hold at given program points. The purpose of the analysis is to derive initial goals or pre-conditions that guarantee that, when the program is executed, the given properties hold. The solution for logic programs presented here is based on a transformation of the input program, which makes explicit the dependencies of the given program points on the initial goals. The transformation is derived from the *resultants* semantics of logic programs. The transformed program is then analysed using a standard abstract interpretation. The required pre-conditions on initial goals can be deduced from the analysis results without a further fixpoint computation. For the modes backwards analysis problem, this approach gives the same results as previous work, but requires only a standard abstract interpretation framework and no special properties of the abstract domain.

1 Introduction

The input to backwards analysis is a program together with properties that are required to hold at given program points. The purpose of the analysis is to derive initial goals or pre-conditions that guarantee that, when the program is executed, the given properties hold. Discussion of the motivation for backwards analysis is given by King and Lu [KL02] and Genaim and Codish [GC01]. For example, in a logic program, it is useful to know which instantiation modes of goals will definitely not produce run-time instantiation errors caused by calls to built-in predicates with insufficiently instantiated arguments [KL02], and which goals are sufficiently instantiated to ensure termination [GC01]. By contrast, program analysis frameworks usually start with given goals, and derive properties that hold at various program points, when those goals are executed.

An essential aspect of static analysis using abstractions or approximations is that the analysis results are *safe*. Backwards analysis algorithms have distinctive characteristics in this regard. The final result, namely (a description of) the set of initial goals that guarantee the establishment of the given properties, should

* Supported by European Framework 5 Project ASAP (IST-2001-38059)

be an *under* approximation of the actual set of goals that satisfy the requirements. Analyses usually yield an *over* approximation, this has led investigators to develop special abstract interpretations that give an under approximation.

In this paper we develop a method for using standard abstraction and over-approximation techniques, and still obtain valid results for backwards analysis. This is achieved by analysing not the original program, but rather a transformed program that makes explicit the dependencies between the given properties and initial goals.

The method is presented in terms of (constraint) logic programs. The essential idea is to transform a given program P into another program (or rather meta-program) whose semantics is a *dependency* relation $\langle A, B\rangle$, where B is a call at some specified program point, and A is an atomic goal for P. Analysis of this transformed program yields an over-approximation of the set of dependencies between A and B, which can then be examined to find goals A that guarantee some required property of B.

1.1 Making Derivations Observable

The transformation to be presented in Section 2 makes explicit the dependencies of program points on initial goals. The transformation can be viewed as the implementation of a more expressive semantics than usual. Standard semantics (such as least Herbrand models, c-semantics, s-semantics, call and success patterns for atomic goals, and so on) do not record explicitly the relationship between initial goals and specific program points. The *resultants semantics* [GLM96,GG94] provides a sufficiently expressive framework.

Resultants Semantics A *resultant* is a formula $Q_1 \leftarrow Q_2$ where Q_1, Q_2 are conjunctions of atoms[1]. If Q_1 is an atom the resultant is a *clause*. Variables occurring in Q_2 but not in Q_1 are implicitly existentially quantified. All other variables are free in the resultant.

Definition 1. $\mathcal{O}_L(P)$

Given a definite program P, the resultants semantics $\mathcal{O}_L(P)$ is the set of all resultants[2] *$p(\bar{X})\theta \leftarrow R$ such that $p(\bar{X})$ is a "most general" atom (that is, an atom of the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are distinct variables) for some predicate in P, and $\leftarrow p(\bar{X}), \ldots, \leftarrow R$ is an SLD- derivation (with a computation rule selecting the leftmost atom) of $P \cup \{\leftarrow p(\bar{X})\}$ with computed answer θ. Such a resultant represents a partial computation of the goal $p(\bar{X})$. We include the zero-length derivations of form $p(\bar{X}) \leftarrow p(\bar{X})$.*

From here on the leftmost computation rule is assumed and the subscript L in $\mathcal{O}_L(P)$ is omitted. There is also a fixpoint definition of $\mathcal{O}(P)$; abstract interpretation of the resultants and related semantics was considered in [CLM01].

[1] Standard terminology and notation for logic programming is used [Llo87].

[2] Strictly speaking $\mathcal{O}_L(P)$ contains equivalence classes of resultants with respect to variable renaming, rather than resultants themselves.

Other standard semantics can be derived as abstractions of $\mathcal{O}(P)$. The subset of elements $p(\bar{X})\theta \leftarrow R \in \mathcal{O}(P)$ where $R = \mathsf{true}$ is isomorphic to the s-semantics [BGLM94], from which in turn the c-semantics [Cla79] and the least Herbrand model [Llo87] can be derived by computing all instances and ground instances respectively. Calls generated by a given goal can also be derived from $\mathcal{O}(P)$. The set of calls that arise from a given atomic goal A in a leftmost SLD derivation is given by the set $\mathsf{calls}(P, A) = \{B_1\theta \mid H \leftarrow B_1, \ldots, B_n \in \mathcal{O}(P), \mathsf{mgu}(A, H) = \theta\}$. We assume as usual that A is standardised apart from the elements of $\mathcal{O}(P)$.

1.2 Backwards Analysis Based on the Resultants Semantics

The possibility of using the resultants semantics for backwards analysis does not seem to have been considered previously. The relation $B \in \mathsf{calls}(P, A)$ can be read backwards; given B, A is a goal that invokes a call B.

We can capture the essential information about the dependencies between calls and goals using the *downwards closure* of $\mathcal{O}(P)$, denoted $\mathcal{O}^+(P)$. That is, $\mathcal{O}^+(P)$ is $\mathcal{O}(P)$ extended with all the instances obtained by substitutions for free variables, which are variables occurring in the resultants' heads. Then define a relation $\mathcal{D}$, called the *goal dependency* relation for P.

$$\mathcal{D}(A, B) \equiv (A \leftarrow B, \ldots, B_n \in \mathcal{O}^+(P))$$

The goal dependency relation for a program is closely related to the binary clause semantics of Codish and Taboch [CT99] (but is downwards closed with respect to the free variables).

Proposition 1. *Let P be a program, and $\mathcal{D}$ be the goal dependency relation for P. Then (i) if $\mathcal{D}(A, B)$ then $B \in \mathsf{calls}(P, A)$, and (ii) for all goals A and $B \in \mathsf{calls}(P, A)$, there exists a substitution σ such that $\mathcal{D}(A\sigma, B)$.*

Proof. (i). If $\mathcal{D}(A, B)$ then $\mathcal{O}(P)$ contains $A' \leftarrow B'_1, \ldots, B'_n$ such that $A \leftarrow B, \ldots, B_n$ is an instance obtained by a substitution, say θ, for the variables in A'. Hence $\mathsf{mgu}(A, A') = \theta$ and $B = B'_1\theta$, and so $B \in \mathsf{calls}(P, A)$ (ii) If $B \in \mathsf{calls}(P, A)$ then $\mathcal{O}(P)$ contains $A' \leftarrow B'_1, \ldots, B'_n$, $\mathsf{mgu}(A, A') = \sigma$ and $B = B'_1\sigma$. The instance $A\sigma \leftarrow B, \ldots, B'_n\sigma$ is thus contained in the downwards closure $\mathcal{O}^+(P)$ and hence $\mathcal{D}(A\sigma, B)$ holds.

Definition 2. *Let P be a program and $\mathcal{D}$ be the goal dependency relation for P. Let Θ and Φ be properties of atoms; that is, for every atom A, $\Theta(A)$ and $\Phi(A)$ are either true or false. We say that a* call-dependency *$\Theta \rightarrow \Phi$ follows from $\mathcal{D}$ if there does not exist $\mathcal{D}(A, B)$ such that $\Theta(A) \wedge \neg\Phi(B)$.*

Definition 3. *A property Θ is called* downwards closed *if, whenever $\Theta(A)$ holds, $\Theta(A\varphi)$ holds for all substitutions φ.*

Proposition 2. *Let P be a program, and $\mathcal{D}$ be the goal dependency relation for P. Suppose $\Theta \rightarrow \Phi$ follows from $\mathcal{D}$, and that Θ is a downwards closed property. Then for all goals A, and $B \in \mathsf{calls}(P, A)$, $\Theta(A) \rightarrow \Phi(B)$.*

Proof. Let A be a goal, such that $\Theta(A)$ holds. For all $B \in \mathsf{calls}(P, A)$, we must establish that $\Phi(B)$ holds. For each such B there exists some instance $A\sigma$ such that $\mathcal{D}(A\sigma, B)$ by Proposition 1. $\Theta(A\sigma)$ holds since Θ is a downwards closed property. Hence $\Phi(B)$ holds since $\Theta \rightarrow \Phi$ follows from $\mathcal{D}$.

Proposition 2 establishes that we can use the goal dependency relation of a program in order to establish dependencies between goals and calls, provided that the properties on goals are downwards closed. The next proposition shows that we can use over-approximations of the goal dependency relation to deduce dependencies.

Proposition 3. *Let S be a goal dependency relation and let S' be a relation including S. Then, if the call-dependency $\Theta \rightarrow \Phi$ follows from S', it also follows from S.*

Proof. Suppose that $\Theta \rightarrow \Phi$ follows from S'. Then there does not exist $\mathcal{D}(A, B) \in S'$ such that $\Theta(A) \wedge \neg\Phi(B)$. Hence such a pair does not exist in S either, and so $\Theta \rightarrow \Phi$ follows from S.

We can also explain how our approach achieves the "under-approximations" of the conditions on initial goals discussed earlier. Given a call property Φ, suppose $\Theta \rightarrow \Phi$ follows from the goal dependency relation $\mathcal{D}$. In an over-approximation of $\mathcal{D}$, we will in general be able to establish dependencies $\Theta' \rightarrow \Phi$, such that $\Theta' \rightarrow \Theta$. Put another way, the larger the approximation is, the more chance there is of finding a counterexample $\mathcal{D}(A, B)$ such that $\Theta(A) \wedge \neg\Phi(B)$. The greater the over-approximation, the more restrictive are the properties Θ' for which $\Theta' \rightarrow \Phi$ can be shown.

The backwards analysis method can now be summarised in the following way. The concrete semantics on which we define properties is the goal dependency relation $\mathcal{D}$ for a given program. Given a program P we define a transformed program containing a predicate whose logical consequences contain the goal dependency relation $\mathcal{D}$. Using abstract interpretation of the transformed program, we compute approximations of $\mathcal{D}$, which can be used to establish dependencies between goals and calls, as proved in Propositions 2 and 3.

We shall also define an even more refined transformed program, whose semantics is restricted to a subset of the goal dependency relation $\mathcal{D}$, containing tuples $\mathcal{D}(A, B)$ where B is a call occurring at one of a specified set of program points.

Basing our approach on a downwards closed semantics allows a straightforward approach to implementation, using for example the framework presented in [GBS95]. Our analyses are based on the c-semantics [Cla79], which is the set of atomic logical consequences of a program. Given a program P, let $\mathcal{C}(P)$ be the c-semantics of P. As shown in [GBS95], $\mathcal{C}(P)$ can be given a least fixpoint form.

2 The Program Transformation

First, the resultants semantics is formulated as a program transformation.

2.1 Resultants Semantics by Program Transformation

A resultant $A \leftarrow Q$ is represented as a meta-predicate $\mathcal{R}(A, Q)$. Let P be a program. For each program clause $H \leftarrow D_1, \ldots, D_n$ $(n > 0)$ in P we produce n clauses.

$$\begin{array}{l} \mathcal{R}(H, (Q, D_2, \ldots, D_n)) \leftarrow \mathcal{R}(D_1, Q) \\ \mathcal{R}(H, (Q, D_3, \ldots, D_n)) \leftarrow D_1, \mathcal{R}(D_2, Q) \\ \ldots \\ \mathcal{R}(H, Q) \leftarrow D_1, \ldots, D_{n-1}, \mathcal{R}(D_n, Q) \end{array}$$

For each unit clause $H \leftarrow \mathsf{true}$ produce a single clause $\mathcal{R}(H, \mathsf{true}) \leftarrow \mathsf{true}$. Finally, for each predicate p we add a clause $\mathcal{R}(p(\bar{x}), p(\bar{x}))$ where $p(\bar{x})$ is a most general call to p.

In the bodies of the clauses for $\mathcal{R}$ there are calls to the original program atoms D_1, D_2 and so on, so it is assumed that the clauses for P are included in the transformed program. These object program calls could have been written $\mathcal{R}(D_1, \mathsf{true}), \mathcal{R}(D_2, \mathsf{true})$ respectively since A is in the minimal model of the program iff there is a ground instance of a resultant $A \leftarrow \mathsf{true}$ in the resultants semantics of the program. If this modification were made, the transformation corresponds closely to the fixpoint definition of the resultants semantics [GLM96]. We denote by Res_P the collection of clauses defining the predicate $\mathcal{R}$ as shown above, together with P itself.

Example 1. Let P be the "naive reverse" program. The transformed program is shown in Figure 1. The meta-predicate $\mathcal{R}$ is denoted `res` in the program.

```
res(rev([],[]),true) :- true.
res(rev([X|Xs],Zs),(Q,app(Ys,[X],Zs))) :-
  res(rev(Xs,Ys),Q).
res(rev([X|Xs],Zs),Q) :-
  rev(Xs,Ys), res(app(Ys,[X],Zs),Q).
res(app([],Ys,Ys),true) :- true.
res(app([X|Xs],Ys,[X|Zs]),Q) :-
  res(app(Xs,Ys,Zs),Q).
res(rev(X,Y),rev(X,Y)) :- true.
res(app(X,Y,Z),app(X,Y,Z)) :- true.
```

```
rev([],[]).
rev([X|Xs],Zs) :-
  rev(Xs,Ys),
  app(Ys,[X],Zs).
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :-
  app(Xs,Ys,Zs).
```

Fig. 1. Res_P where P is the naive reverse program

Proposition 4. *Let P be a program. Then for all resultants $A \leftarrow G \in \mathcal{O}^+(P)$, $\mathcal{R}(A, G) \in \mathcal{C}(\mathsf{Res}_P)$.*

Proof. (Outline). A derivation corresponding to a resultant can be represented as an AND-OR proof tree. The proof is by induction on the depth of AND-OR trees.

Note that $\mathcal{C}(\mathsf{Res}_P)$ contains more instances of resultants than does $\mathcal{O}^+(P)$. Specifically, local variables in resultants are also instantiated, as well as head variables. The transformed program thus represents an approximation of the dependency relation. In practice this is not a loss in precision, since clearly no dependencies will be derived between local variables in resultants and head variables.

2.2 From Resultants to Binary Clauses

The program above can be modified to yield (the downwards closure of) binary clauses [CT99]. Only the first call in the right-hand-side of the resultants is recorded, rather than the whole resultant. A resultant $A_1 \leftarrow A_2$ in which both A_1 and A_2 are atoms is called a *binary clause*. In the binary clause semantics, a resultant $A \leftarrow B_1, \ldots, B_n$ is abstracted to $A \leftarrow B_1$.

The transformed program corresponding to the binary clauses is as follows. A meta-predicate $\mathcal{B}(A_1, A_2)$ represents the binary resultant $A_1 \leftarrow A_2$.

$$\begin{array}{l}\mathcal{B}(H, Q) \leftarrow \mathcal{B}(D_1, Q).\\ \mathcal{B}(H, Q) \leftarrow D_1, \mathcal{B}(D_2, Q).\\ \ldots\\ \mathcal{B}(H, Q) \leftarrow D_1, \ldots, D_{n-1}, \mathcal{B}(D_n, Q).\end{array}$$

As before, we add a clause $\mathcal{B}(p(\bar{x}), p(\bar{x}))$ for each predicate p where $p(\bar{x})$ is a most general atom for p. Note that a unit clause in P produces no clauses for $\mathcal{B}$. Let Bin_P be the transformed program consisting of P together with the clauses defining the predicate $\mathcal{B}$ as shown above.

Example 2. Let P be the "naive reverse" program. The transformed program is shown in Figure 2. The meta-predicate $\mathcal{B}$ is denoted `bin` in the program.

```
bin(rev([X|Xs],Zs),Q) :-
  bin(rev(Xs,Ys),Q).
bin(rev([X|Xs],Zs),Q) :-
  rev(Xs,Ys), bin(app(Ys,[X],Zs),Q).
bin(app([X|Xs],Ys,[X|Zs]),Q) :-
  bin(app(Xs,Ys,Zs),Q).
bin(rev(X,Y),rev(X,Y)) :- true.
bin(app(X,Y,Z),app(X,Y,Z)) :- true.
```

```
rev([],[]).
rev([X|Xs],Zs) :-
  rev(Xs,Ys),
  app(Ys,[X],Zs).
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :-
  app(Xs,Ys,Zs).
```

Fig. 2. Bin_P where P is the naive reverse program

Proposition 5. *Let P be a program. Then for all resultants $A \leftarrow B_1, \ldots, B_n \in \mathcal{O}^+(P)$, $\mathcal{B}(A, B_1) \in \mathcal{C}(\mathsf{Bin}_P)$.*

$\mathcal{C}(\mathsf{Bin}_P)$ is an over approximation of the goal dependency relation for P. As was the case for the resultants program Res_P, the downwards closure of local variables is included in the relation $\mathcal{B}$ in $\mathcal{C}(\mathsf{Bin}_P)$.

2.3 Transforming with Respect to Program Points

Next, a further simplification is made, when calls at specified program points are to be observed, rather than all calls. We may if required observe only a specific argument of a call at some program point. A meta-predicate $\mathsf{Dep}(A_1, A_2)$ is defined, whose meaning is that there is a clause $A_1 \leftarrow A_2$ in the binary clause semantics, and A_2 is a call, or some argument of a call, at one of the specified program points to be observed.

Let $H \leftarrow B_1, \ldots, B_j, \ldots, B_n$ be a clause in a program P. Suppose that we wish to observe calls to B_j in this clause body, and determine some property of initial goals which establish some property of B_j. In the semantics, only the binary clauses of the form $A \leftarrow B_j$ are to be observed: no other calls other than those to B_j need be recorded.

To achieve this, we simply modify the binary clause transformation shown above. Specifically, instead of the clauses of form $\mathcal{B}(p(\bar{x}), p(\bar{x}))$, we create base case clauses for the given program points.

For instance, for the clause $H \leftarrow D_1, \ldots, D_j, \ldots, D_n$ with one point D_j to be observed, the following clauses for Dep are generated.

$$\mathsf{Dep}(H, D_j) \leftarrow D_1 \ldots, D_{j-1} \qquad \begin{array}{l} \mathsf{Dep}(H, Q) \leftarrow \mathsf{Dep}(D_1, Q). \\ \mathsf{Dep}(H, Q) \leftarrow D_1, \mathsf{Dep}(D_2, Q). \\ \ldots \\ \mathsf{Dep}(H, Q) \leftarrow D_1, \ldots, D_{n-1}, \mathsf{Dep}(D_n, Q). \end{array}$$

For each body atom to be observed, we add one clause similar to the one for D_j above. We can see that the only atoms that can appear in the second argument of Dep are instances of D_j. Denote by Dep_P the transformed program consisting of P together with the clauses defining Dep as shown above.

Proposition 6. *Let P be a program, and $\{D_{j_1}, \ldots, D_{j_k}\}$ be a set of body atoms from clauses in P. Let Dep_P be the transformed program consisting of P together with the clauses defining Dep as shown above. Then for all resultants $A \leftarrow D_{j_i}, \ldots \in \mathcal{O}^+(P)$, where D_{j_i} is an instance of one of the specified atoms, $\mathsf{Dep}(A, D_{j_i}) \in \mathcal{C}(\mathsf{Dep}_P)$.*

The transformation can be refined (with respect to computational efficiency) by having a separate Dep predicate corresponding to each predicate in P. That is, each occurrence of $\mathsf{Dep}(p(\bar{t}), Q)$ in the transformed program is replaced by $\mathsf{Dep}_p(\bar{t}, Q)$.

The transformation can be varied by observing in the second argument of Dep not the actual call, but simply one or more variables from the call. This is illustrated in the next example.

Example 3. Let P be the "naive reverse" program. Suppose the call that we wish to observe is `app(Ys,[X],Zs)` in the recursive clause for `rev` as shown in Figure 3. For example, we suppose that we require that `integer(X)` holds whenever this call is encountered. We need observe only the variable `X` in `app(Ys,[X],Zs)`.

However, the transformation is independent of the actual property. The transformed program, shown in Figure 3, consists of P together with the clauses defining `drev/2` and `dapp/3` (representing the meta-predicates Dep_{rev} and Dep_{app}). In place of the call `app(Ys,[X],Zs)` in the final argument, we observe only the variable `X`.

```
drev([X|Xs],Zs,X) :-
  rev(Xs,Ys).
drev([X|Xs],Zs,Q) :-
  drev(Xs,Ys,Q).
drev([X|Xs],Zs,Q) :-
  rev(Xs,Ys), dapp(Ys,[X],Zs,Q).
dapp([X|Xs],Ys,[X|Zs],Q) :-
  dapp(Xs,Ys,Zs,Q).
```

```
rev([],[]).
rev([X|Xs],Zs) :-
  rev(Xs,Ys),app(Ys,[X],Zs).
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :-
  app(Xs,Ys,Zs).
```

Fig. 3. Transformed Naive Reverse Program for Backwards Analysis

Next, we apply standard static analysis techniques to the transformed program.

2.4 Analysis of the Transformed Programs

The transformed program can be input to an abstract interpretation framework. In the experiments carried out so far, analysis was based on the c-semantics abstracted using pre-interpretations [GBS95]. A pre-interpretation is a mapping from terms into a (finite) domain D, defined by a pre-interpretation function J. For each n-ary function symbol f, J contains a function $D^n \rightarrow D$, written $J(f(d_1,\ldots,d_n)) = d$ for $d_1,\ldots,d_n,d \in D$. A mapping α is defined inductively as $\alpha(c) = d$ where $J(c) = d$, for 0-ary functions c, and $\alpha(f(t_1,\ldots,t_n)) = J(f(\alpha(t_1),\ldots,\alpha(t_n)))$ for terms with functions of arity greater than 0. An abstract "domain program" is generated by abstract compilation, in the style introduced by Codish and Demoen [CD93]. A bottom-up analysis of the domain program yields its c-semantics. Let P be a program and $\mathcal{C}(P)$ its minimal model, which is identical to the c-semantics in this case. Let P^J be the abstract domain program for some pre-interpretation J. The safety result is that for all atoms $p(t_1,\ldots,t_n) \in \mathcal{C}(P)$, $p(\alpha(t_1),\ldots,\alpha(t_n)) \in \mathcal{C}(P^J)$.

Example 4. We analyse the above example where we wish to establish the property `app(Ys,[X],Zs)` $\leftrightarrow$`integer(X)`, for the occurrence of `app(Ys,[X],Zs)` in the recursive clause for `rev/2`. A simple type domain could be used, consisting of the types `int, listint, other`. We construct an abstract "domain program" as described in [GBS95], based on the pre-interpretation constructed from the program's function symbols and the given types.

[] $\longrightarrow$ listint	[int | other] $\longrightarrow$ other	[other | other] $\longrightarrow$ other
[listint | other] $\longrightarrow$ other	[int | int] $\longrightarrow$ other	[listint | int] $\longrightarrow$ other
[other | int] $\longrightarrow$ other	[int | listint] $\longrightarrow$ listint	[listint | listint] $\longrightarrow$ other
[other | listint] $\longrightarrow$ other		

```
rev(X1,X2):-
  []→X1,[]→X2.
rev(X1,X2):-
  rev(X3,X4),app(X4,X5,X2),
  [X6|X3]→X1,[]→X7,[X6|X7]→X5.
app(X1,X2,X2):-
  []→X1.
app(X1,X2,X3):-
  app(X4,X2,X5),[X6|X4]→X1,[X6|X5]→X3.
drev(X1,X2,X3):-
  rev(X4,X5),[X3|X4]→X1.
drev(X1,X2,X3):-
  rev(X4,X5),dapp(X5,X6,X2,X3),
  [X7|X4]→X1,[]→X8,[X7|X8]→X6.
drev(X1,X2,X3):-
  drev(X4,X5,X3),[X6|X4]→X1.
dapp(X1,X2,X3,X4):-
  dapp(X5,X2,X6,X4),[X7|X5]→X1,[X7|X6]→X3.
```

Fig. 4. Domain Program for Backwards Analysis of Naive Reverse

The pre-interpretation is encoded as a predicate →/2 corresponding to the pre-interpretation, that is, for each mapping $f(d_1, \ldots, d_n) \rightarrow d$ in the pre-interpretation, we write an atomic clause $\mathtt{f}(\mathtt{d}_1, \ldots, \mathtt{d}_n) \rightarrow \mathtt{d} : -\mathsf{true}$. The domain program is shown in Figure 4. Its least model over the pre-interpretation for the domain of simple types is shown in Figure 5.

2.5 Interpretation of the Analysis Result

Examining the results in Figure 5, we see a number of abstract facts for drev. (There are no results for dapp derived since no call to app affects the given program point.) The results show that whenever rev/2 is called with its first argument a list of integers, then X is an integer at the given program point. This is indicated by the fact that $\mathtt{drev}(\mathsf{listint}, \mathtt{X1}, \mathsf{int})$ is in the model of the abstract program, and there are no other tuples $\mathtt{drev}(\mathsf{listint}, \mathtt{X1}, \mathtt{Y})$ where $\mathtt{Y} \neq \mathsf{int}$. By contrast, there is a tuple $\mathtt{drev}(\mathsf{other}, \mathtt{X1}, \mathsf{int})$ but there is also a tuple $\mathtt{drev}(\mathsf{other}, \mathtt{X1}, \mathsf{listint})$, so although goals of the form rev(other,Y) *might* establish the property, they are not *guaranteed* to establish it.

In terms of the discussion in Section 1.2, the goal dependency $\Theta \rightarrow \Phi$ follows from the abstract relation, where $\Theta(\mathtt{rev}(\mathtt{X}, \mathtt{Y}))$ is true if X is a list of integers, and $\Phi(\mathtt{app}(\mathtt{Ys}, [\mathtt{X}], \mathtt{Zs}))$ is true is this call arises from the specified program point, and X is an integer.

Example 5. Let P be the *quicksort* program, shown in Figure 6. Backwards analysis was considered for this program in [KL02]. Suppose we wish to check the calls to the built-in predicates $\geq$ and $<$. The intention is that these predicates require their argument to be ground when called in order to prevent run-time instantiation errors. The transformed *quicksort* program is included in Figure 6.

```
app(listint,X1,X1)          rev(listint,listint)  drev(listint,X1,int)
app(listint,int,other)      rev(other,other)      drev(other,X1,int)
app(other,other,other)                            drev(other,X1,listint)
app(other,int,other)                              drev(other,X,other)
app(other,listint,other)
```

Fig. 5. Least model of program in Figure 4, over domain of simple types

```
qsort([],Ys,Ys).
qsort([X|Xs],Ys,Zs) :-
  partition(Xs,X,Us,Vs),
  qsort(Us,Ys,[X|Ws]),
  qsort(Vs,Ws,Zs).
partition([],Z,[],[]).
partition([X|Xs],Z,Ys,[X|Zs]) :-
  X ≥ Z, partition(Xs,Z,Ys,Zs).
partition([X|Xs],Z,[X|Ys],Zs) :-
  X < Z, partition(Xs,Z,Ys,Zs).

dqsort([X|Xs],Ys,Zs,Q) :-
  dpartition(Xs,X,Us,Vs,Q).
dqsort([X|Xs],Ys,Zs,Q) :-
  partition(Xs,X,Us,Vs),
  dqsort(Us,Ys,[X|Ws],Q).
dqsort([X|Xs],Ys,Zs,Q) :-
  partition(Xs,X,Us,Vs),
  qsort(Us,Ys,[X|Ws]),
  dqsort(Us,Ys,[X|Ws],Q).
dpartition([X|Xs],Z,Ys,[X|Zs],X ≥ Z).
dpartition([X|Xs],Z,Ys,[X|Zs],Q) :-
  X ≥ Z, dpartition(Xs,Z,Ys,Zs,Q).
dpartition([X|Xs],Z,[X|Ys],Zs,X<Z).
dpartition([X|Xs],Z,[X|Ys],Zs,Q) :-
  X < Z, dpartition(Xs,Z,Ys,Zs,Q).
```

Fig. 6. Transformed Quicksort Program for Backwards Analysis

2.6 Analysis of Quicksort

We perform groundness analysis on the program in Figure 6. A pre-interpretation over the domain elements `g` and `ng` (standing for *ground* and *non-ground*) is constructed. This is equivalent to the Pos boolean domain.

$$[] \longrightarrow \mathsf{g} \quad [\mathsf{g} \mid \mathsf{g}] \longrightarrow \mathsf{g} \quad [\mathsf{g} \mid \mathsf{ng}] \longrightarrow \mathsf{ng} \quad [\mathsf{ng} \mid \mathsf{g}] \longrightarrow \mathsf{ng} \quad [\mathsf{ng} \mid \mathsf{ng}] \longrightarrow \mathsf{ng}$$

After generating the domain program, the least model is computed and is shown in Figure 7. (When computing the minimal model we assign the success modes $\mathtt{g} \geq \mathtt{g}$ and $\mathtt{g} < \mathtt{g}$ to the built-ins).

Examining the results via the relation `dqsort`, we see that the only calls to `qsort(X,Y,Z)` that guarantee that the required groundness properties $\mathtt{g} \geq \mathtt{g}$ and $\mathtt{g} < \mathtt{g}$ are those in which `X` is ground. The arguments `Y` and `Z` are completely independent of the property. For `dpartition`, note that a variable `X1` occurs in both the final argument of `dpartition` and in the second argument of `partition`. This variable can be instantiated by `g` or `ng`. Thus the second argument of `partition` has to be ground to establish $\mathtt{g} \geq \mathtt{g}$ and $\mathtt{g} < \mathtt{g}$. In addition, the arguments of $\geq$ and $<$ are ground if either the first argument of `partition` or the third and fourth are ground. These are the same results reported by King and Lu [KL02], summarised as $X_2 \wedge (X_1 \vee (X_3 \wedge X_4))$ in the notation of Pos, where $X_1, \ldots, X_4$ are the arguments of `partition`.

partition(g,X1,g,g)	qsort(g,X1,X1)
	qsort(ng,ng,g)
	qsort(ng,ng,ng)
dpartition(ng,X1,ng,X2,g<X1)	dqsort(ng,X1,X2,ng≥ng)
dpartition(ng,X1,g,X2,g<X1)	dqsort(ng,X1,X2,ng≥g)
dpartition(ng,X1,ng,X2,ng<X1)	dqsort(ng,X1,X2,g≥ng)
dpartition(g,X1,ng,X2,g<X1)	dqsort(ng,X1,X2,ng<ng)
dpartition(g,X1,g,X2,g<X1)	dqsort(ng,X1,X2,ng<g)
dpartition(ng,X1,X2,ng,g≥X1)	dqsort(ng,X1,X2,g<ng)
dpartition(ng,X1,X2,g,g≥X1)	dqsort(g,X1,X2,g≥g)
dpartition(ng,X1,X2,ng,ng≥X1)	dqsort(g,X1,X2,g<g)
dpartition(g,X1,X2,ng,g≥X1)	dqsort(ng,X1,X2,g≥g)
dpartition(g,X1,X2,g,g≥X1)	dqsort(ng,X1,X2,g<g)

Fig. 7. Least model of program in Figure 6, over groundness domain

2.7 Computing the Goal Conditions

For examples such as the ones discussed above, the required properties of the input goals that guarantee the observed property were derived informally by examining the abstract tuples. We now explain how to do this systematically.

Let $\mathsf{Dep}(A, B)$ be the abstract dependency relation returned by the analysis, which is a finite set of tuples. Let Φ be the property required in the call; that is, we seek calls B where $\Phi(B)$ is true. Consider the set $S = \{A \mid \mathsf{Dep}(A, B) \wedge \Phi(B)\}$. S is the set of calls that *possibly* establishes $\Phi(B)$. Now consider candidate properties Θ that hold for all elements of S. For each such property, check whether there exists $\mathsf{Dep}(A, B)$ such that $\Theta(A)$ and $\neg\Phi(B)$. If there is, the candidate property is eliminated. For all other candidate properties, we have established that $\Theta \rightarrow \Phi$ follows from the abstract dependency relation.

We illustrate this process for the *quicksort* example. Consider the relation `dqsort` shown in Figure 7. The required property is that $\Phi(\mathtt{g} \geq \mathtt{g})$ and $\Phi(\mathtt{g} < \mathtt{g})$ are true and Φ is false for all other arguments of $\geq$ and $<$. The tuples in the abstract `dqsort` relation in which Φ holds are the following.

dqsort(g, X1, X2, g≥g)
dqsort(g, X1, X2, g<g)
dqsort(ng, X1, X2, g≥g)
dqsort(ng, X1, X2, g<g)

A candidate property is then that the first argument of `qsort` can be either `g` or `ng`, to establish the required property. However, we can search the relation to find a counterexample to the candidate property that the first argument is `ng`, such as dqsort(ng,X1,X2,ng<g). However we can find no counterexample to the property that the first argument is `g`. Hence we have established that `qsort(g,X1,X2)` $\rightarrow \Phi$.

2.8 The Relative Pseudo-Complement

Domains which possess a relative pseudo-complement allow a more direct method. Giacobazzi and Scozzari [GS98] identified a property of abstract domains that allows analyses to be reversible. This property is central to the approach of King and Lu [KL02,KL03]. The key property is that the domain possesses a *relative pseudo-complement* operator. We quote the definition as given by King and Lu. Let D be an abstract domain with meet and join operations $\sqcap$ and $\sqcup$. Let d_1, d_2 be elements of D. The pseudo-complement of d_1 relative to d_2, denoted $d_1 \Rightarrow d_2$ is the greatest element whose meet with d_1 is less than d_2: that is, $d_1 \Rightarrow d_2 = \sqcup\{d \in D \mid d \sqcap d_1 \sqsubseteq d_2\}$.

To take Example 5 again, treat `g` and `ng` as `true` and `false` respectively. The set of abstract tuples for say, `dpartition` in Figure 7, can be rewritten as the following boolean expression, in the domain Pos, which possesses a relative pseudo-complement operation (here $q(X, Y)$ means $X \geq Y \wedge X < Y$).

$$\texttt{dpartition}(X_1, X_2, X_3, X_4, q(X_5, X_6)) \equiv$$
$$(X_2 \leftrightarrow X_6) \wedge ((\bar{X_1} \wedge \bar{X_3} \wedge X_5) \vee (\bar{X_1} \wedge X_3 \wedge X_5) \vee (\bar{X_1} \wedge \bar{X_3} \wedge \bar{X_5}) \vee$$
$$(X_1 \wedge \bar{X_3} \wedge X_5) \vee (X_1 \wedge X_3 \wedge X_5) \vee (\bar{X_1} \wedge \bar{X_4} \wedge X_5) \vee (\bar{X_1} \wedge X_4 \wedge X_5) \vee$$
$$(\bar{X_1} \wedge \bar{X_4} \wedge \bar{X_5}) \vee (X_1 \wedge \bar{X_4} \wedge X_5) \vee (X_1 \wedge X_4 \wedge X_5))$$

The pseudo-complement of the above boolean expression relative to the desired property $X_5 \wedge X_6$ gives $X_2 \wedge (X_1 \vee (X_3 \wedge X_4))$, which is equivalent to the result derived in Example 5, and the same as that reported by King and Lu [KL02] for this predicate.

3 Related Work

The most closely related work is that of King and Lu [KL02,KL03], who describe a method for backwards analysis of logic programs, and report results for the domain of ground and non-ground modes. Their results have all been reproduced by the technique shown above, but a formal proof of equivalence has not yet been constructed. Their approach requires the construction of an abstract interpretation which under-approximates the concrete semantics. This requires the definition of a universal projection operator, and requires a condensing domain possessing a relative pseudo-complement operator. The fixpoint computation uses a greatest fixpoint rather than the standard least fixpoint. Our approach appears to be more flexible in the sense that a wide variety of domains can be used for the analysis, not only condensing domains. The relative pseudo-complement, if it exists, can be used in our approach to extract the result from the abstract program, but is not essential.

Mesnard *et al.* [Mes96,MN01] have also performed termination inference, which is a form of backwards analysis.Their approach uses a greatest fixpoint, and in this respect seems to align more with the approach of King and Lu.

The *binary clause* semantics of Codish and Taboch [CT99] was used to make loops observable, by deriving an explicit relationship between a calls and its

successor calls. The transformation presented here can be targeted to observe any program points of interest, not only loops, but the spirit of the approach is the same. In later work based on binary clause semantics, Genaim and Codish [GC01] perform termination inference which involves backwards analysis. However, they use the framework of King and Lu for the backwards analysis, rather than the binary clause semantics.

Binary clause semantics is derived from the more general and expressive resultants semantics [GLM96,GG94]. We do not know of any implemented applications of resultants semantics, apart from the present work and that of [CT99,GC01], nor any previous suggestion that resultants semantics could form the basis for backwards analysis.

The approach of transforming programs to realise non-standard semantics is also followed in the *query-answer* transformations, which include magic-set transformations and its relations [DR94,BMSU86]. There, the aim is to simulate a top-down goal-directed computation, in a bottom-up semantic framework. A related approach is advocated by Codish and Søndergaard [CS02]. Different semantics for logic programs can be represented by meta-interpreters, which are also written as logic programs. Codish and Genaim's implementation of the binary semantics [GC01] follows this style.

4 Conclusion

A method for backwards analysis of logic programs has been presented. Given a program, and one or more specified body calls, a program transformation is performed. In the transformed program, the dependencies between the selected calls and initial goals is made explicit. Analysis of the transformed program using abstract interpretation yields an over-approximation of the dependency relation, and it was proved that dependencies could safely be derived from the approximation.

In contrast to previous work on backwards analysis, our approach requires no special properties of the abstract domain, nor any non-standard operations such as universal projection, or a greatest fixpoint computation. This is put forward as an advantage of our approach, since implementations can be based on existing abstract interpretation tools.

Experimental results carried out so far indicate that this method is of similar complexity to other reported work on backwards analysis, and gives equivalent precision at least over the Boolean domain Pos. A detailed analytical comparison is difficult due to the great differences between the two approaches. It is indeed quite surprising that two such different algorithms yield the same results in experiments carried out so far.

Our use of downwards closed semantics does not seem to be essential to our general approach, but does allow a simpler analysis and implementation.

Acknowledgements

Thanks to Andy King and Mike Codish for introduction to, and discussions on backwards analysis, and to Maurice Bruynooghe and the LOPSTR'03 referees for valuable comments on an earlier draft. Roberto Giacobazzi, Samir Genaim and Maurizio Gabbrielli also provided useful feedback on the version appearing in the LOPSTR'03 Pre-proceedings. This research is supported in part by the IT-University of Copenhagen.

References

[BGLM94] Annalisa Bossi, Maurizio Gabbrielli, Giorgio Levi, and Maurizio Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5^{th} ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

[CD93] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*. MIT Press, 1993.

[Cla79] K. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.

[CLM01] Marco Comini, Giorgio Levi, and Maria Chiara Meo. A theory of observables for logic programs. *Information and Computation*, 169(1):23–80, 2001.

[CS02] Michael Codish and Harald Søndergaard. Meta-circular abstract interpretation in prolog. In Torben Mogensen, David Schmidt, and I. Hal Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 109–134. Springer-Verlag, 2002.

[CT99] Michael Codish and Cohavit Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[DR94] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[GBS95] J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365, 1995.

[GC01] S. Genaim and M. Codish. Inferring termination conditions of logic programs by backwards analysis. In *International Conference on Logic for Programming, Artificial intelligence and reasoning*, volume 2250 of *Springer Lecture Notes in Artificial Intelligence*, pages 681–690, 2001.

[GG94] Maurizio Gabbrielli and Roberto Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the 1994 ACM Symposium on Applied Computing, SAC 1994*, pages 394 – 399, 1994.

[GLM96] Maurizio Gabbrielli, Giorgio Levi, and Maria Chiara Meo. Resultants semantics for Prolog. *Journal of Logic and Computation*, 6(4):491–521, 1996.

[GS98] R. Giacobazzi and F Scozzari. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[KL02] Andy King and Lunjin Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4-5):514–547, 2002.

[KL03] Andy King and Lunjin Lu. Forward versus backward verification of logic programs. In *ICLP'2003 (to appear)*, 2003.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.

[Mes96] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. J. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 7–21. MIT Press, 1996.

[MN01] F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In *Static Analysis Symposium*, volume 2126 of *LNCS*, pages 93–110, 2001.