An Engine for Logic Program Analysis

Nevin HEINTZE
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
nch@cs.cmu.edu

Joxan JAFFAR
IBM T.J. Watson Research Center
PO Box 740
Yorktown Heights, NY 10598
joxan@watson.ibm.com

Abstract

Most logic program analyzers employ a standard approach based on abstract interpretation. At the core of these is a generic algorithm or "engine" which is parameterized by an abstract domain. This standard engine is essentially a fixpoint computation over the abstract semantic equations of a program. This paper presents a new engine which is based on unfolding of semantic equations. A main advantage of the unfolding engine is a uniform treatment of structural information in a program. In particular, reasoning about partially instantiated structures, an area where traditional algorithms have been weak, is greatly enhanced. The main result of this paper shows that our engine is uniformly more accurate than the standard engine in the following sense: given an abstract domain, the output of our engine, for any program, is more accurate than that of the standard engine.

1 Introduction

Most algorithms for logic program analysis are based on abstract interpretation, and essentially consist of two components: an abstract domain (and implementations of its associated functions) and an implementation of an iterative fixpoint computation. The focus of these algorithms is in the design of the abstract domain. In general, more sophisticated domains give better analysis.

Consider the simple abstract domain whose formulas map program variables into either *ground* or *any*. For example, the formula $[X \mapsto ground, Y \mapsto any]$ represents substitutions which ground X (and may or may not ground Y). For the first program below, this domain can be used to deduce that Y is ground, but not that X is ground.

```
?- eq(X, Y), r(Y).
eq(U, U).
r(a).
```

```
?- p(f(X, b)).
p(U) ← q(U).
q(f(a, V)).
```

```
?- p(Y), r(Y, X).
p(U) ← q(U).
q(f(a, V)).
r(f(W1, W2), W1).
```

To overcome this problem, domains have been developed in which information about the aliasing and sharing behavior of variables is maintained; see [3, 12, 7, 9, 11] for example. These *dependency-based* approaches improve the accuracy of the analysis and, for the first program, they infer that both X and Y are ground. They are, however, rather weak in reasoning about partially instantiated structures. For example, in the second program, the dependency-based approaches cannot infer that X is ground since this requires reasoning about the structure of the terms to which U can be bound.

A different approach, that of set constraints [5], has been shown to be useful for reasoning about structural information. The basic idea here is to reason about the program variables as sets of possible values, as opposed to one set of substitutions. While the target application has been type analysis, this approach can be used for groundness analysis. For the second program, it can infer that X is ground. The weakness of using set constraints, however, is that intervariable dependencies are not represented. For example in the first and third programs, it cannot be inferred that X is ground. Thus the dependency-based and set constraint approaches have different advantages and disadvantages, and hence are not directly comparable.

This paper focuses on the fixpoint computation aspect of program analysis, rather than the domain. Whereas domains vary widely from one algorithm to another, the fixpoint computation aspects are similar. (Often algorithms differ because ad hoc approximations are introduced for efficiency reasons.) In fact, it possible to speak of a generic algorithm or "engine" which encompasses the essence of these fixpoint computations.

This paper presents a new engine which can reason about structural information in a uniform manner (as in the set constraint approach), and can also reason about dependencies between variables (as in the standard algorithms). The third program above, for example, can be analyzed by the new unfolding engine. The underlying iteration technique is based upon the use of unfold transformations. While the standard engine iterates over fixed semantic equations, the unfolding engine iterates over dynamically changing equations. The main result shows that the unfolding engine is uniformly more accurate than the standard engine in the sense that, given an abstract domain, the output of the unfolding engine, for any program, is more accurate than that of the standard engine.

2 Semantic Equations

The starting point for our engine is an equational representation of a program's operational semantics. While such representations are well understood, the purpose of this section is to provide the specific formulation required for the rest of this paper. We introduce our semantic equations by means of an example. Consider the `append` program:

```
1.  append(nil, W, W).
2.  append(X.Xs,Y,X.Zs)←append(Xs,Y,Zs).
```

The formulation of semantic equations presented here serves to represent, for each rule, the computed answer substitutions obtained from executing the body of that rule. Specifically, introduce variables Ψ_1 and Ψ_2 to represent the sets corresponding to rules 1 and 2 respectively. Now construct the equations:

$$\begin{aligned}\Psi_1 &= \{mgu()\}|_{\mathcal{V}_1} \\ \Psi_2 &= \begin{array}{l}\left\{mgu\left(body = \overline{\gamma(head_1)}\right) : \gamma \in \Psi_1\right\}\Big|_{\mathcal{V}_2} \\ \bigcup \left\{mgu\left(body = \overline{\gamma(head_2)}\right) : \gamma \in \Psi_2\right\}\Big|_{\mathcal{V}_2}\end{array}\end{aligned}$$

where $head_1$ and $head_2$ denote the head atoms of rules 1 and 2 respectively, $body$ denotes the body atom of rule 2, and $\mathcal{V}_1$ and $\mathcal{V}_2$ denote the variables in rules 1 and 2 respectively. The function mgu[1] maps a set of equations to its most general unifier[2]. The function $|_{\mathcal{V}}$ restricts the domain of a set of substitutions to the variables $\mathcal{V}$. Finally, $\overline{t}$ denotes the renaming of the variables in t into new variables. Intuitively, a semantic equation $\Psi_i = \cdots$ provides a definition for the variables in the i^{th} rule.

A *model* of such equations is a mapping from each Ψ_i to a set of substitutions such that each equation is satisfied. For example

$$\Psi_1 = \{(W \mapsto W)\},$$

$$\Psi_2 = \left\{\begin{array}{l}\begin{pmatrix} Xs \mapsto nil \\ Y \mapsto W \\ Zs \mapsto W\end{pmatrix}, \begin{pmatrix} Xs \mapsto X.nil \\ Y \mapsto W \\ Zs \mapsto X.W\end{pmatrix} \\ \begin{pmatrix} Xs \mapsto X.X'.nil \\ Y \mapsto W \\ Zs \mapsto X.X'.W\end{pmatrix}, \cdots\end{array}\right\}$$

where $X', \cdots$ are new variables, is a model of the above equations for the `append` program. It is in fact the least such model[3] (up to renaming of new variables). It coincides exactly with the program's meaning in the sense that the set for Ψ_i consists of the computed answer substitutions obtained when the body of the i^{th} rule is executed as a goal.

More formally, let P be a logic program whose rules are $R_1, \cdots, R_N$. Let H_i denote the head atom of R_i, and let $\mathcal{V}_i$ denote $var(R_i)$, the variables in R_i. Now, for each rule R_i of the form $H \leftarrow B_1, \ldots, B_m$, construct an equation $\Psi_i =$

$$\bigcup_{j_1,\cdots,j_m} \left\{ mgu \begin{pmatrix} B_1 = \overline{\gamma_1(H_{j_1})} \\ \cdots \\ B_m = \overline{\gamma_m(H_{j_m})}\end{pmatrix} : \begin{array}{c}\gamma_k \in \Psi_{j_k} \\ 1 \le k \le m\end{array} \right\}\Bigg|_{\mathcal{V}_i}$$

where the sequence $j_1, \cdots, j_m$ ranges over all possible m-tuples of indices such that (a suitably renamed version of) H_{j_k} unifies with B_k, $1 \le k \le m$. Note that the variable renaming above is such that the expressions $\overline{\gamma_k(H_{j_k})}$ are variable disjoint, and further, their variables do not appear in $B_1, \cdots, B_m$.

Before considering more examples, consider abbreviating expressions on the rhs (right hand side) of semantic equations of the form

$$\left\{ mgu\left(s_1 = \overline{\gamma_1(t_1)}, \cdots, s_m = \overline{\gamma_m(t_m)}\right) : \begin{array}{c}\gamma_k \in \Psi_{j_k} \\ 1 \le k \le m\end{array}\right\}\Bigg|_{\mathcal{V}}$$

by $\left([s_1 = t_1]_{j_1}, \cdots, [s_m = t_m]_{j_m}\right)_{\mathcal{V}}$. Such an expression is called a *cluster*. For example, the equations for the `append` program can be written as follows. Note that the rhs of the first equation is an empty cluster, and

[1] There can be different definitions; we assume any fixed one.

[2] $mgu(\phi)$ is the identity substitution.

[3] Models are ordered componentwise: $model_1 \le model_2$ iff $model_1(\Psi_i) \subseteq model_2(\Psi_i)$ for each variable Ψ_i.

the rhs of the second equation contains two clusters.

$$\Psi_1 = ()_{\mathcal{V}_1}$$
$$\Psi_2 = \big([append(Xs, Y, Zs) = append(nil, W, W)]_1\big)_{\mathcal{V}_2} \cup \big([append(Xs, Y, Zs) = append(X.Xs, Y, X.Zs)]_2\big)_{\mathcal{V}_2}$$

In general, expressions such as $[s = t]_i$ contained in clusters are called *groups*, and the individual equations contained in groups are called *group equations*. (Note that group equations are not pure equations; thus $[s = t]_i$ is different from $[t = s]_i$.) The subscript i indicates the rule i, and is called the *dependency* of the group. Consider below another example program and its semantic equations, illustrating clusters of more than one group. Note that we have omitted the subscripts on groups and clusters since these values are obvious from the context (for example, the dependency of the first group is 1 because the variables on the rhs appear only in rule 1, and the subscript of the first cluster is $\{X, Y\}$ because the cluster appears in the equation for rule 1).

```
1.  p(f(X)) ← p(X), q(X), r(Y).
2.  p(a).
3.  q(W).
4.  r(b).
```

$$\Psi_1 = \left(\begin{array}{c} [p(X) = p(f(X))] \\ [q(X) = q(W)] \\ [r(Y) = r(b)] \end{array} \right) \bigcup \left(\begin{array}{c} [p(X) = p(a)] \\ [q(X) = q(W)] \\ [r(Y) = r(b)] \end{array} \right)$$
$$\Psi_2 = \Psi_3 = \Psi_4 = ()$$

In the least model, Ψ_1 is $\{(X \mapsto a, Y \mapsto b),\ (X \mapsto f(a), Y \mapsto b),\ (X \mapsto f(f(a)), Y \mapsto b),\ \cdots\}$.

We finally remark that the semantics equations we have defined correspond to a bottom-up execution of the program P at hand. We can also define semantic equations corresponding to a top-down execution of P; we defer an illustration of this to the appendix. Neither formulation presents any new concepts; rather, they are necessary preliminary definitions.

In what follows, the term semantic equations shall mean either the equations corresponding to bottom-up or top-down program execution. Our engine can be applied in either case.

3 Unfolding Semantic Equations

A key idea behind our engine is the substitution of one cluster into another. Such a substitution step may generate a new cluster. The engine is essentially an exhaustive application of this step. We now elaborate.

Consider a group $\mathcal{G}$ whose dependency i is a fact, that is, rule i has no body. Since the equation for rule i is of the form $\Psi_i = ()$, any model of the semantic equations must map Ψ_i to the singleton set consisting of the identity substitution. Hence $\mathcal{G}$ has a fixed interpretation in any model, and can be thought of as representing a fixed set of substitutions. Call such groups $\mathcal{G}$ *terminal*. A group is in *substitution form* if the lhs (left hand side) of each equation therein is a variable; a cluster is in substitution form if each of its groups is in substitution form. We illustrate informally how such clusters are used as substitutions by returning to the `append` example. Recall the semantic equations for `append`, which we can simplify (by rewriting equations between predicates $p(\tilde{s}) = p(\tilde{t})$[4] into $\tilde{s} = \tilde{t}$) to become:

$$\Psi_1 = ()$$
$$\Psi_2 = \left(\begin{bmatrix} Xs = nil \\ Y = W \\ Zs = W \end{bmatrix} \right) \cup \left(\begin{bmatrix} Xs = X.Xs \\ Y = Y \\ Zs = X.Zs \end{bmatrix} \right)$$

Denote the two clusters in the equation for Ψ_2 by $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. $\mathcal{C}_1$ contains a terminal group $\mathcal{G}_1$ which represents the substitution $(Xs \mapsto nil,\ Y \mapsto W,\ Zs \mapsto W)$. $\mathcal{C}_2$ contains a group $\mathcal{G}_2$ which is non-terminal. By substituting $\mathcal{C}_1$ into $\mathcal{C}_2$, we obtain the new cluster:

$$\left(\begin{bmatrix} Xs & = & X.nil \\ Y & = & W \\ Zs & = & X.W \end{bmatrix} \right) \tag{3.1}$$

which can be added to the equation for Ψ_2 so that the rhs of Ψ_2 becomes:

$$\left(\begin{bmatrix} Xs{=}nil \\ Y{=}W \\ Zs{=}W \end{bmatrix} \right) \cup \left(\begin{bmatrix} Xs{=}X.Xs \\ Y{=}Y \\ Zs{=}X.Zs \end{bmatrix} \right) \cup \left(\begin{bmatrix} Xs{=}X.nil \\ Y{=}W \\ Zs{=}X.W \end{bmatrix} \right)$$

The correctness of a substitution step follows from the fact that it is essentially an "unfolding" operation using the definition of Xs, Y, and Zs. That is, the new expression (3.1) is subsumed by $\mathcal{C}_2$ in the following sense: in all models $\mathcal{I}$ of the original equations, $\mathcal{I}((3.1)) \subseteq \mathcal{I}(\mathcal{C}_2)$. Hence the original and new equations are equivalent.

However, the new equation is more explicit in the sense that it displays more elements of the set Ψ_2 (as terminal groups). Clearly the process of unfolding of equations can be repeated indefinitely to obtain new

[4] The symbol $\sim$ groups terms into a sequence; for example, $s_1, \cdots, s_n$ may be abbreviated $\tilde{s}$ and equations $s_1 = t_1, \cdots, s_n = t_n$ abbreviated $\tilde{s} = \tilde{t}$.

clusters ($[Xs = X.X'.nil, Y = W, Zs = X.X'.W]$), $\cdots$, etc. Note that each such cluster represents an answer substitution for the body of the second rule of `append`. To obtain a terminating analysis algorithm, this process of adding new clusters must be curtailed by using a notion of approximation.

Consider approximating the new groups constructed during the substitution step. Each such group is terminal and hence defines a fixed set of substitutions. Hence any traditional means for approximating substitution sets can be employed (see, for example, [8]). We present one formalization as follows.

Definition 1 An abstract domain $\mathcal{D}$ is a set of *abstract formulas* together with (a) an *abstraction function* ABS which takes a set of variables and a set of substitutions and returns an abstract formula, and (b) a *concretization function* CONC which takes an abstract formula and returns a set of substitutions satisfying $\text{CONC}(\text{ABS}_{\mathcal{V}}(\Theta)) \supseteq \Theta$ for all Θ and variables $\mathcal{V}$. (Typically, some further algebraic conditions are specified, but these will not concern us here.) The definition of ABS and CONC induces a notion of *abstract unification*[5] which is the mapping $\mathcal{D}\text{-}unify_{\mathcal{V}}(s, t, \mathcal{A}) = \text{ABS}_{\mathcal{V}}(\{mgu(s = t\theta) : \theta \in \overline{\text{CONC}(\mathcal{A})}\})$ where s, t are terms, $\mathcal{A} \in \mathcal{D}$, and the overline operator in $\overline{\text{CONC}(\mathcal{A})}$ renames the range of substitutions in $\text{CONC}(\mathcal{A})$ to avoid name clashes. $\square$

Example 1 [11] Let $\mathcal{D}_{prop}$ consist of all propositional formulas over certain variables, whose purpose is to describe propositional relationships between groundness of variables. Let ABS and CONC be described informally as follows. Define that a basic proposition (a variable) X is true in a substitution θ if θ grounds X; the proposition $X \vee Y$ is true in θ if θ grounds *either* X or Y; similarly for $X \wedge Y$, $X \longleftrightarrow Y$, $X \wedge Y \longleftrightarrow Z$, etc. Now define $\text{CONC}(\mathcal{A})$ to be $\{\theta : \mathcal{A} \text{ is true in } \theta\}$, and define $\text{ABS}_{\mathcal{V}}(\Theta)$ to be the weakest proposition $\mathcal{A}$ such that $var(\mathcal{A}) \subseteq \mathcal{V}$ and $\mathcal{A}$ is true for all $\theta \in \Theta$. For instance, the equation $X = f(a, Y)$ can be approximated wrt variables $\{X, Y\}$ by the formula $X \longleftrightarrow Y$; similarly $Xs = X.nil, Y = W, Zs = X.W$ wrt $\{Xs, Y, Zs\}$ by $(Xs \wedge Y \longleftrightarrow Zs)$. $\square$

We return now to the the `append` example. Using the above abstract domain $\mathcal{D}_{prop}$, we can obtain a terminating unfolding process in a straightforward way: approximate any new groups $\mathcal{G}$ constructed during the unfolding step by $\text{ABS}_{\mathcal{V}}(\mathcal{G})$. Specifically, the first unfolding step yields the equation $\Psi_2 =$

$$\left(\begin{bmatrix} Xs = nil \\ Y = W \\ Zs = W \end{bmatrix}\right) \cup \left(\begin{bmatrix} Xs = X.Xs \\ Y = Y \\ Zs = X.Zs \end{bmatrix}\right) \cup ([Xs \wedge Y \longleftrightarrow Zs])$$

The algorithm outlined so far is not very accurate because the substitution step, when approximated, loses a significant amount of structural information. In fact, when a group $\mathcal{G}_1$ is substituted into a group $\mathcal{G}_2$, the resulting cluster loses all information about these two groups except for that contained in the $\mathcal{D}$-approximation. A key observation of this paper is that the group $\mathcal{G}_2$ can in fact be retained. Specifically, we define that the substitution of $\mathcal{G}_1$ into $\mathcal{G}_2$, which originally yielded a single group $[Xs \wedge Y \longleftrightarrow Zs]$, now yields two groups:

$$\left(\begin{bmatrix} Xs \wedge Y \longleftrightarrow Zs \end{bmatrix} \quad \begin{bmatrix} Xs & = & X.Xs \\ Y & = & Y \\ Zs & = & X.Zs \end{bmatrix}^{\text{Ⓡ}} \right)$$

In other words, the group $\mathcal{G}_2$ remains after the substitution, and is then pronounced a *residual group* (marked with Ⓡ). It is these residual groups which provide our engine with the ability to reason about structures.

4 The Engine

We begin with some definitions, some of which were informally discussed above. Let VAR denote a denumerable set of variables, and let P be a logic program constructed from functors Σ and variables VAR. Let the rules in P be $R_1, \cdots, R_N$ ($N \geq 1$) with variables $\mathcal{V}_1, \cdots, \mathcal{V}_N$ respectively. We assume that the sets $\mathcal{V}_i$ are mutually disjoint.

An *exact group* is of the form $[s_1 = t_1, \cdots, s_n = t_n]_k$ where the s_i and t_i are terms constructed from Σ and VAR. Such a group is *terminal* if R_k is a fact. Some exact groups may be marked *residual* during execution of the engine. An *approximate group* is defined wrt a given abstract domain $\mathcal{D}$; it is simply of the form $[A]$ where $A \in \mathcal{D}$. A *group* is an exact or approximate group. A *cluster* is of the form $(\mathcal{G}_1, \cdots, \mathcal{G}_n)$ where each $\mathcal{G}_i$ is a group. In the previous section, the semantic equations for a program P were defined. Our engine starts with these equations and modifies them during its execution. At all times, corresponding to each rule R_k, there is one semantic equation of the form $\Psi_k = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n$ where the $\mathcal{C}_i$ are clusters and Ψ_k is a distinguished variable representing substitutions on $\mathcal{V}_k$.

[5] For pragmatic reasons, some analysis algorithms do not implement $\mathcal{D}$-*unify* exactly, but instead use a conservative approximation.

An exact group is in *substitution form* if the lhs of each equation therein is a variable. Let $\mathcal{G}$ be a group in substitution form, and let $X_1, \cdots, X_n$ be the variables $var(lhs(\mathcal{G}))$. Now define $subs(\mathcal{G})$ to be the set of all substitutions of the form $(X_1 \mapsto t_1, \cdots, X_n \mapsto t_n)$ where $X_i = t_i \in \mathcal{G}$, $1 \leq i \leq n$. A exact group can be made into substitution form by deleting certain equations: $\widehat{[\mathcal{E}]}_j \stackrel{\text{def}}{=} [X = t : X = t \in \mathcal{E} \text{ and } X \text{ is a variable }]_j$. (This overhat operator will be used only in the construction of residual groups.) A cluster is in *substitution form* if each group therein is either an approximate group, or a terminal group in substitution form, or a residual group (such groups are identified by the engine).

The basic operation of our algorithm is the notion of substituting a cluster in substitution form *into* a group. What results is a new cluster. First define the composition of an exact group with an approximate group as follows:

$$[\mathcal{E}]_i \circ [\mathcal{A}] \stackrel{\text{def}}{=} \mathcal{D}\text{-}unify_{var(\tilde{s})}(\tilde{s}, \tilde{t}, \mathcal{A})$$

where $\mathcal{E}$ is $\tilde{s} = \tilde{t}$. Similarly define the composition of an exact group $[\mathcal{E}_1]$ with an exact group $[\mathcal{E}_2]$ in substitution form as follows:

$$[\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j \stackrel{\text{def}}{=} \left[s = t\theta : (s = t) \in \mathcal{E}_1, \theta \in subs(\overline{\mathcal{E}_2})\right]_j$$

where $\overline{\mathcal{E}_2}$ renames all of the head-only[6] variables in $rhs(\mathcal{E}_2)$ into new ones. (For example, $[X = f(W,Y)] \circ [Y = W]$, where W is a head-only variable, should not yield $[X = f(W,W)]$, but rather $[X = f(W,W')]$.)

To embed a notion of approximation into the composition $[\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j$ of groups, define the "approximate composition" $[\mathcal{E}_1]_i \diamond [\mathcal{E}_2]_j$ to consist of two groups. The first group is simply the $\mathcal{D}$-approximation of $[\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j$:

$$\left[\text{ABS}_{var(lhs(\mathcal{E}_1))}\left(mgu\left([\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j\right)\right)\right]$$

The second group is a subset of $[\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j$. It is defined by $select([\mathcal{E}_1]_i, [\mathcal{E}_2]_j) =$

$$\left[s = t\theta : \begin{array}{l} (s = t) \in \mathcal{E}_1,\ \theta \in subs(\overline{\mathcal{E}_2}) \text{ and} \\ \text{either } t \text{ is a variable or } |t\theta| = |t| \end{array}\right]_j$$

where $|s|$ is the number of symbols in s. Intuitively, *select* chooses equations from $[\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j$ in such a way the size of any term therein does not exceed the size of the terms in the original equations $[\mathcal{E}_1]_i$ and $[\mathcal{E}_2]_j$.

[6] A head-only variable appears in the head of a rule but not in the body.

For example, if $[\mathcal{E}_1]_i$ and $[\mathcal{E}_2]_j$ were $[X = W_1, Y = f(W_1), Z = f(W_2)]_i$ and $[W_1 = g(a), W_2 = V]_j$ respectively, then $[\mathcal{E}_1]_i \circ [\mathcal{E}_2]_j$ is $[X = g(a), Y = f(g(a)), Z = f(V)]_j$. However $select([\mathcal{E}_1]_i, [\mathcal{E}_2]_j)$ will contain the first and third equations, but not the second because $f(g(a))$ is too big. As we see later, it is possible to consider other definitions of *select*.

We are now in a position to define the main step of our engine.

Definition 2 (Unfolding Step) *Let $\mathcal{C}_1$ be a cluster containing a non-terminal group $\mathcal{G}$. Let $\mathcal{C}_2$ be a cluster in substitution form. Construct a new cluster by replacing $\mathcal{G}$ in $\mathcal{C}_1$ by all of the following groups:*

- *$\widehat{\mathcal{G}}$, and this group is marked as residual;*
- *$\mathcal{G} \circ [\mathcal{A}]$, for all approximate groups $[\mathcal{A}] \in \mathcal{C}_2$, and*
- *$\mathcal{G} \diamond [\mathcal{E}]_i$, for all exact groups $[\mathcal{E}]_i \in \mathcal{C}_2$.*

We say that this new cluster is the result of <u>substituting</u> $\mathcal{C}_2$ <u>into</u> $\mathcal{C}_1$ <u>at</u> $\mathcal{G}$. $\Box$

Example 2 Consider the abstract domain $\mathcal{D}_{prop}$ above. The following presents clusters $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ respectively. $\mathcal{C}_3$ is the result of substituting $\mathcal{C}_2$ into $\mathcal{C}_1$.

$$([X = f(Y)]) \qquad \left(\begin{array}{l} \left[\begin{array}{c} Y = U \\ Y = g(a) \\ Z = b \end{array}\right] \\ [Y = V] \end{array}\right) \qquad \left(\begin{array}{l} [X = f(Y)]^{\circledR} \\ [X = f(U)] \\ [X] \\ [X = f(V)] \\ [true] \end{array}\right)$$

Consider $\mathcal{C}_3$. The group $[X = f(Y)]$ is residual. The second group $[X = f(U)]$ is obtained from $select([X = f(Y)] \circ [Y = U, Y = g(a), Z = b])$. (Note that the equation $X = f(g(a))$ is omitted by *select*.) The third group $[X]$ is approximate, and is obtained from $\text{ABS}_{\{X\}}(mgu([X = f(Y)] \circ [Y = U, Y = g(a), Z = b]))$. It contains the proposition that X is ground. The fourth group $[X = f(V)]$ is obtained from $select([X = f(Y)] \circ [Y = V])$. Finally, the fifth group $[true]$ is approximate and is obtained from $\text{ABS}_{\{X\}}(mgu([X = f(Y)] \circ [Y = V]))$. $\Box$

An unfolding step is applied to a collection of semantic equations as follows. Let $\mathcal{C}_1$ appear in the semantic equation for Ψ_i, and let $[\mathcal{E}]_j \in \mathcal{C}_1$. Let $\mathcal{C}_2$ appear in the semantic equation for Ψ_j. If substituting $\mathcal{C}_2$ into $\mathcal{C}_1$ at $[\mathcal{E}]_j$ results in a cluster $\mathcal{C}$ which does not already appear in the semantic equation for Ψ_i, then $\mathcal{C}$ is added to this equation and we say that the unfolding step was *effective*.

The complete engine is presented in figure 1. It is essentially an exhaustive application of the unfolding step just defined. Note that before an unfolding step is performed, it is necessary to perform some straightforward simplification steps which can be described as follows:

- replace a group equation of the form $f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n)$ by $s_1 = t_1, \cdots, s_n = t_n$;
- delete any cluster which contains (a) a group equation of the form $f(\tilde{s}) = g(\tilde{t})$ or (b) two group equations of the form $X = f(\tilde{s})$ and $X = g(\tilde{t})$, where $f \neq g$;
- replace any group of the form $[s_1 = t_1, \cdots, s_n = t_n]$, where one of the equations is of the form $f(u_1, \cdots, u_k) = W$ and W is a head-only variable, by $[s_1 = t_1\theta, \cdots, s_n = t_n\theta]$ where θ is the substitution $[W \mapsto f(W_1, \cdots, W_k)]$ and the variables $W_1, \cdots, W_k$ are new variables. (Intuitively, a group such as $[f(X) = W, Y = W]$ does define a substitution though not in substitution form. This simplification step remedies the problem by transforming it to $[f(X) = f(W_1), Y = f(W_1)]$, which in turn can be simplified into $[X = W_1, Y = f(W_1)]$.)

When no more effective unfolding steps can be performed, all clusters in substitution form are output after first deleting the residual groups therein. Such clusters represent fixed sets of substitutions because the groups therein are not recursively defined in terms of other groups and/or Ψ_i variables. The important point here is that the output is an *explicit* representation of a set of substitutions for each Ψ_i. A post-processing phase may then be applied to extract the specific information sought.

Example 3 Consider the following program and its (simplified) semantic equations:

```
main(Y) ← p(f(X)), q(X, Y).
p(U) ← r(U).
q(g(W1, W2), W1).
r(g(a, W3)).
r(f(V)) ← r(V).
```

[7] These groups in fact do contain some structural information. We choose to delete them because substantial post-processing is required to extract this information, and space reasons forbid further description of this.

> **input** an arbitrary program P;
> **obtain** the semantic equations for P;
> **repeat**
>   **exhaustively apply** the basic simplification steps;
>   **apply** an effective unfolding step if possible;
> **until** no effective unfolding step is possible;
> **delete** all residual groups[7];
> **output** all clusters in substitution form;

Figure 1: **The Unfolding Engine**

$$\Psi_1 = \begin{pmatrix} [f(X) = U] \\ [X = g(W1, W2),\ Y = W1] \end{pmatrix}^{①}$$

$$\Psi_2 = ([U = g(a, W3)])^{②} \ \cup\ ([U = f(V)])^{③}$$

$$\Psi_3 = \Psi_4 = ()$$

$$\Psi_5 = ([V = g(a, W3)])^{④} \ \cup\ ([V = f(V)])^{⑤}$$

These initial semantic equations contain two substitution clusters ② and ④. The following three unfolding steps can be performed: cluster ② can be substituted into ① and cluster ④ can be substituted into ③ and ⑤ to obtain

$$\begin{pmatrix} [f(X) = g(a, W3)] \\ [X = g(W1, W2),\ Y = W1] \end{pmatrix}^{⑥}$$

$$\begin{pmatrix} [U = f(V)]^{®} \\ [true] \end{pmatrix}^{⑦}$$

$$\begin{pmatrix} [V = f(V)]^{®} \\ [true] \end{pmatrix}^{⑧}$$

respectively. Consider the first unfolding step. There is no residual group in ⑥ because $\widehat{[\mathcal{E}]} = \emptyset$ when $[\mathcal{E}] = [f(X) = U]$. Cluster ⑥ is (eventually) deleted because it contains the conflicting group equation $f(X) = g(a, W3)$. The clusters ⑦ and ⑧ are added to the equations for Ψ_2 and Ψ_5 respectively.

Next, the clusters ⑦ and ⑧ can be used in unfolding steps. We omit the details for ⑧ since these steps do not lead to any new clusters. Substituting ⑦ into ① yields ⑨ which can be simplified into ⑩ and then added to the equation for Ψ_1:

$$\begin{pmatrix} [f(X)=f(V)] \\ [true] \\ \begin{bmatrix} X=g(W1,W2) \\ Y=W1 \end{bmatrix} \end{pmatrix}^{⑨} \quad \begin{pmatrix} [X=V] \\ [true] \\ \begin{bmatrix} X=g(W1,W2) \\ Y=W1 \end{bmatrix} \end{pmatrix}^{⑩}$$

Now, there are two possible substitutions into this new cluster: ④ into ⑩ and ⑧ into ⑩, respectively yielding

$$\begin{pmatrix} [X=V]^{®} \\ [X=g(a,W3)] \\ [true] \\ [X=g(W1,W2),\ Y=W1] \end{pmatrix}^{⑪}$$

$$\begin{pmatrix} [X=V]^{®} \\ [X=f(V)] \\ [true] \\ [X=g(W1,W2),\ Y=W1] \end{pmatrix}^{⑫}$$

The second cluster ⑫ is (eventually) deleted since it contains the conflicting equations $X = f(V)$ and $X = g(W1, W2)$. The cluster ⑪ is added to the equation for Ψ_1. At this point the engine terminates, and outputs all clusters in substitution form: ②, ④, ⑦, ⑧, and ⑪. Of interest is ⑪, corresponding to Ψ_1, which represents the fixed collection of substitutions given by $mgu(X = g(a, W3), X = g(W1, W2), Y = W1)$ conjoined with *true*. This in turn can be simplified and projected onto $\{X, Y\}$ to obtain $\{(X \mapsto g(a, t), Y \mapsto a) : t \text{ is any term}\}$ from which we can deduce that Y is ground. □

To see need for residual groups in the above example, consider removing the residual group in ⑦. Then the group $[f(X) = f(V)]$ would not be in in ⑨, and consequently the group $[X = V]$ would not be in in ⑩, and consequently the group $[X = g(a, W3)]$ would not be in ⑪, our output cluster. Hence it cannot be inferred that Y is ground.

One could raise the question as to why the group in ③, which gave rise to the residual group in ⑦, could not *itself* be used for substitution into ①. (That is, consider substituting ③ into ①.) Allowing this means that groups which are neither terminal nor residual are allowed to be substituted into other clusters. This means that many more unfolding steps are permitted in general. The following example shows that such substitutions cause loss of information.

Example 4 Consider the domain $\mathcal{D}_{prop}$ once again, and let $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ respectively be

$$\left(\begin{bmatrix} U_1=f(X_1) \\ f(U_2)=X_2 \end{bmatrix}\right) \quad \left(\begin{bmatrix} X_1=f(Y_1) \\ X_2=f(Y_2) \end{bmatrix}\right) \quad \left(\begin{bmatrix} Y_1=f(W) \\ Y_2=f(W) \end{bmatrix}\right)$$

Let the group in $\mathcal{C}_3$ be the only terminal group. Our engine dictates that $\mathcal{C}_3$ is first substituted into $\mathcal{C}_2$ to obtain $([X_1 = f(Y_1), X_2 = f(Y_2)]^{®}, [X_1 \longleftrightarrow X_2])$ which is then substituted into $\mathcal{C}_1$ to obtain a new cluster $\mathcal{C}_4$ which is $([U_1 = f(X_1)]^{®}, [U_2 = Y_2], [U_1 \longleftrightarrow U_2])$ after simplification. Finally, $\mathcal{C}_3$ is substituted into $\mathcal{C}_4$ to obtain a cluster in substitution form containing $[U_1 \longleftrightarrow U_2]$. Assuming that this is the only cluster output for the rule containing U_1 and U_2, it deduced that U_1 is ground iff U_2 is ground.

Now suppose that we allow the substitution of $\mathcal{C}_2$ into $\mathcal{C}_1$, obtaining $([U_1 = f(X_1)]^{®}, [U_2 = Y_2], [true])$, and then using $\mathcal{C}_3$ to substitute into this, we obtain $([U_1 = f(X_1)]^{®}, [U_2 = Y_2]^{®}, [U_2 = f(W)], [true])$. Since this cluster is substitution form, it is output. Clearly it forbids the inference that U_1 is ground iff U_2 is ground. Intuitively, the problem here was that the substitution of $\mathcal{C}_2$ into $\mathcal{C}_1$ was performed before the information from $\mathcal{C}_3$ was used; this information could not be recovered later. □

In order to prove correctness of the engine, it is necessary to first define the meaning of semantic equations used by the engine. Now, the meaning of semantic equations containing only exact groups was given above. However approximate groups are generated during the algorithm. We now informally describe the meaning of these more general equations.

Let $mgi(\theta_1, \theta_2)$ denote the most general instance of θ_1 and θ_2 if it exists, that is, the most general substitution which is an instance of both θ_1 and θ_2. For example, $mgi((X \mapsto f(W), Y \mapsto a), (X \mapsto f(g(W))))$ is $(X \mapsto f(g(W')), Y \mapsto a)$. Next define the function $join_{\mathcal{V}}(\theta_1, \theta_2)$ to be $mgi(\overline{\theta_1}, \overline{\theta_2})|_{\mathcal{V}}$ where the overline renaming operator makes the range of θ_1 and θ_2 disjoint. For example, $join_{\mathcal{V}}((X \mapsto f(a), Y \mapsto W), (X \mapsto W))$ is $mgi((X \mapsto f(a), Y \mapsto W_1), (X \mapsto W_2))|_{\mathcal{V}}$ where W_1 and W_2 are new variables. Finally, extend the *join* function to apply over sets: $join_{\mathcal{V}}(\Theta_1, \Theta_2) \stackrel{\text{def}}{=} \{join_{\mathcal{V}}(\theta_1, \theta_2) : \theta_1 \in \Theta_1, \theta_2 \in \Theta_2\}$.

An *interpretation* $\mathcal{I}$ for semantic equations is a mapping from each Ψ_i variable, $1 \le i \le N$ to a set of substitutions $\mathcal{I}(\Psi_i)$. For groups, $\mathcal{I}([\tilde{s} = \tilde{t}]_j) = \{mgu(\tilde{s} = \overline{\gamma(\tilde{t})}) : \gamma \in \mathcal{I}(\Psi_j)\}$, and $\mathcal{I}([\mathcal{A}])$ is simply $\textsc{conc}(\mathcal{A})$. For clusters in the equation Ψ_i, $\mathcal{I}((\mathcal{G}_1, \cdots, G_n)) = join_{\mathcal{V}_i}(\mathcal{I}(G_1), \cdots, \mathcal{I}(G_n))$. As usual, an interpretation $\mathcal{I}$ is a *model* of semantic equations $\Psi_i = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n$, $1 \le i \le N$, if $\mathcal{I}(\Psi_i) =$

$\mathcal{I}(C_1) \cup \cdots \cup \mathcal{I}(C_n)$.

Theorem 1 (Correctness and Termination) *Let P be a program and let $\mathcal{D}$ be a finite[8] abstract domain. Then the unfolding engine terminates on P using $\mathcal{D}$, and the output semantic equations are a conservative approximation of the initial semantic equations for P.*

Proof outline: Let $\mathcal{E}^P$ denote the semantic equations obtained from the input program P. The first part of the proof establishes that the least model of these equations (denoted $lm(\mathcal{E}^P)$) corresponds to the collecting semantics at hand. This is tedious but fairly straightforward (see [5] and [4] for proofs of related results).

We next prove that the algorithm constructs a conservative approximation to $lm(\mathcal{E}^P)$. Let $\mathcal{E}^{loop}$ denote the equations obtained from $\mathcal{E}^P$ after execution of the main **repeat-until** loop of the unfolding engine. Let $\mathcal{E}^{no-res}$ denote the equations obtained by deleting the residual expressions from $\mathcal{E}^{loop}$, and let $\mathcal{E}^{final}$ denote the final equations, that is those obtained by selecting only the substitution form clusters from $\mathcal{E}^{no-res}$. Now, the main loop of the algorithm merely serves to add expression to the rhs of equations in $\mathcal{E}^P$. Moreover, the original equations in $\mathcal{E}^P$ do not contain any residual expressions. Therefore, for each equation $X = rhs$ in $\mathcal{E}^P$, there will be an equation of the form $X = rhs \cup \cdots$ in both $\mathcal{E}^{loop}$ and $\mathcal{E}^{no-res}$. Hence both $\mathcal{E}^{loop}$ and $\mathcal{E}^{no-res}$ approximate $\mathcal{E}^P$ in the sense that $lm(\mathcal{E}^P) \subseteq lm(\mathcal{E}^{loop})$ and $lm(\mathcal{E}^P) \subseteq lm(\mathcal{E}^{no-res})$.

The main part of the correctness proof establishes that the exhaustive application of the unfolding step generates sufficient substitution form clusters that these clusters alone characterize the least model of $\mathcal{E}^{no-res}$. In essence we prove a form of completeness of the unfolding step with respect to the least model. The proof of this requires reasoning about the specific formulation of the unfolding step, and is omitted.

Termination is proved by showing that there are only a finite number of different clusters that can be produced. This is essentially because during an unfolding step, the function *select* ensures that the size of terms appearing the new cluster are not larger than those appearing in the previous clusters. The main complications are (a) the part of the simplification step which deals with head only variables, since this step can increase the size of terms and introduce new variables, and (b) the renamings of equations during the unfolding step, since this may introduce new variables. A key part of the termination proof is that the number of new variables can be bounded. $\square$

[8] Some algorithms use domains satisfying a weaker condition such as the "finite chain" property. Our engine can in general be adapted to terminate on such domains. We omit the details for simplicity.

5 Variations of the Engine

The main parameter of the engine is the domain used for approximation. We now discuss two other parts of the engine where there is scope for variation.

Consider the *select* function which is used to curtail the growth of group equations. Another reasonable definition of *select* can be obtained by using a uniform bound on the size of terms, the so-called "depth-k" approximation. The important point is that the use of *any select* function does not affect correctness. It will, however, affect accuracy and termination. In general, using a smaller or more restrictive function will make the proof of termination easier (and can also enhance efficiency). Using a bigger function, on the other hand, is more accurate, but it complicates the termination proof in general.

Next consider the output of the engine. As defined, this consists of a collection of clusters in substitution form. Such a collection is an explicit definition, for each program rule, of a set of substitutions. However, to maximize the structural information obtained from the engine, residual groups should not be deleted upon termination. This however raises the problem that if clusters contain residual groups, then they are no longer an explicit representation of a set of substitutions. In fact, they represent an undecidable set of equations in general.

Perhaps the best general technique for extracting information from these residual groups is to approximate them by interpreting them as set constraints, and then employing an intersection algorithm for set expressions (see [6] for example). It is important to note that the engine as stated already has much of the analytical power of the set constraint approach to analysis. Indeed the third program in section 1 shows that it is sometimes strictly more accurate. By augmenting the engine output with the residual groups, and applying an intersection algorithm, the engine in fact becomes uniformly more accurate than the set constraint approach to analysis.

6 Comparison with Abstract Interpretation Algorithms

It is difficult to characterize every algorithm for logic program analysis using a single algorithm. This is because each algorithm is designed with different criteria — ad hoc approximations are often introduced for efficiency reasons. However, these differences are rarely fundamental from a conceptual point of view. Consequently, it is possible to define an idealized engine which encompasses the essence of the abstract interpretation technique which underlies these algorithms, and is at least as accurate as any of them. We can therefore call this idealized engine a *standard* engine, which we now outline.

In general each analysis algorithm starts by associating a value from the chosen abstract domain with designated parts of the program. The operation of the algorithm then consists of repeatedly recomputing each value from the values previously computed. For a bottom-up analysis, an informal and simplified outline is:

initialize the array $value[1..N]$;
loop {
  for $(1 \leq i \leq N)$ {
    let $\tilde{B}$ denote the body atoms of rule i;

$$newvalue[i] := \bigcup_{\tilde{H}} \mathcal{D}\text{-}unify(\tilde{B}, \tilde{H}, value[\,])$$

where $\tilde{H}$ ranges over all sequences of rule heads which (after appropriate renaming) unify with $\tilde{B}$;
  }
  if $(newvalue[\,] = value[\,])$ **break**;
  $value[\,] := newvalue[\,]$;
}
output $value[\,]$;

Example 5 The following illustrates each iteration of the standard engine on an example program. The abstract domain used here is $\mathcal{D}_{prop}$. Note that the i^{th} column describes $value[\,]$ after the i iterations of the main loop. □

```
1:    ?- eq(X, Y), p(f(Y, Z)).
2:    p(U) ← q(U).
3:    q(f(S, V)) ← r(S).
4:    r(a).
5:    eq(W, W).
```

	initial	1	2	3	4
$value[1]$	false	false	false	false	X $\leftrightarrow$ Y
$value[2]$	false	false	false	true	true
$value[3]$	false	false	S	S	S
$value[4]$	false	true	true	true	true
$value[5]$	false	true	true	true	true

We can similarly specify a standard engine for top-down analysis, but we defer the details to the appendix. Amongst the many papers for which the standard engine applies are [10, 7] (bottom-up) and [1, 3, 9] (top-down).

In order to compare the standard engine with the unfolding engine, we will now formulate the operation of the standard engine using our framework of group equations, clusters, composition etc. Recall that in our framework, we can perform bottom-up or top-down analysis simply by using the appropriate semantic equations. Now, the standard engine proceeds exactly as the unfolding engine, except that:

- all terminal groups $[\mathcal{E}]$ are initially replaced by $\text{ABS}_{var(lhs(\mathcal{E}))}([\mathcal{E}])$;
- the unfolding step is simplified so that no residual groups are produced (and this can achieved simply by redefining $\widehat{G} = \emptyset$).

The main effect of these changes is that clusters in substitution form contain only approximate groups (and no terminal or residual groups). Hence the only operation required during unfolding is the composition $[\mathcal{E}] \circ [\mathcal{A}]$ of an exact and approximate group.

Proposition 1 *This formulation correctly describes the standard engine.* □

Clearly these modified unfolding steps are less accurate than the unfolding step used in the (full) unfolding engine. Hence we can verify that

Theorem 2 *The unfolding engine is uniformly more accurate than the standard engine in the sense: given a finite abstract domain, the output of the unfolding engine, for any program, is more accurate than that of the standard engine.* □

An alternative view of the differences between the standard and unfolding engines is as follows. The standard engine implements abstract interpretation by a successive iteration starting from some initial values $\mathcal{A}_0$ and computing the sequence $\mathcal{A}_0 \xrightarrow{\mathcal{F}} \mathcal{A}_1 \xrightarrow{\mathcal{F}} \mathcal{A}_2 \xrightarrow{\mathcal{F}} \mathcal{A}_3 \cdots$ where $\mathcal{F}$ represents the abstract semantic function at hand and the $\mathcal{A}_i$ are the successive abstract values being computed. The unfolding engine, on the other hand, uses a *changing* abstract semantic function and computes $\mathcal{A}_0 \xrightarrow{\mathcal{F}_0} \mathcal{A}_1 \xrightarrow{\mathcal{F}_1} \mathcal{A}_2 \xrightarrow{\mathcal{F}_2} \mathcal{A}_3 \cdots$. Such a change takes place whenever an unfolding step is applied. The important point is that each function $\mathcal{F}_i$ is at least as accurate as the original function $\mathcal{F}$.

We conclude this section with the realization of some analysis algorithms using our engine. Let $\text{UNFOLD}(\mathcal{D})$ denote the algorithm obtained by using the unfolding engine and the abstract domain $\mathcal{D}$; similarly for STANDARD($\mathcal{D}$). Let $\mathcal{D}_{sharing}$ denote the domain described in [7].

Corollary 1 *For groundness analysis,* UNFOLD($\mathcal{D}_{prop}$) *is uniformly more accurate than* [11] [9]*.* □

Corollary 2 *For sharing analysis,* UNFOLD($\mathcal{D}_{sharing}$) *is uniformly more accurate than* [7]*.* □ *(The second example program in section* 1 *shows that* UNFOLD($\mathcal{D}_{sharing}$) *is* strictly *more accurate.)*

7 Conclusion

A new engine for analysis of logic program is presented. Its starting point is a specific formulation of semantic equations of a program. We have illustrated how these equations can model bottom-up and top-down execution. In fact, they can model the usual variations of top-down execution such as those with fixed atom selection rule (for example left-to-right). Using a given abstract domain $\mathcal{D}$, the engine performs unfolding on these equations. Each unfolding is augmented with a notion of approximation, defined in terms of $\mathcal{D}$, to curtail the expansion of terms. The main advantage of our specific technique of unfolding, which uses the key concept of residual groups, is that there is a uniform and accurate treatment of structural information. By formulating the standard analysis engine as a special case of the unfolding engine, we then showed that the unfolding engine is uniformly more accurate than the standard engine.

Yet to be addressed is the issue of efficiency. There are two main parts of the engine in this regard. First is the initial number of clusters in the equations. This number is bounded by the number of sequences of head atoms which match the sequence of body atoms, for each rule. The second part concerns the unfolding step. Though the number of new groups produced by this step is linear in the size of the input clusters, the termination of the engine is based on an exhaustive application of this step. Experience with a prototype implementation of set constraints [4] provides some evidence that these two problems can be overcome. First, the issue of matching body and head atoms in set constraints is essentially the same as in the unfolding engine. Using our set constraint implementation[10] on reasonable example programs, this issue was not prohibitive. Next consider the issue of unfolding efficiency. In set constraints, there is a notion of substitution which is similar to unfolding in terms of the number of new expressions generated (but whose semantics is very different). The said implementation exploits the crucial fact that the substitution step gives rise to mostly *redundant* expressions. This experience suggests that the currently planned implementation of the unfolding engine can be engineered to be practical.

[9] By [2], *UNFOLD*($\mathcal{D}_{prop}$) is uniformly more accurate than a number of other algorithms as well.

[10] Approximately 3000 lines of Standard ML.

Acknowledgments

We thank Michael Maher, Kim Marriott and Roland Yap for their comments.

References

[1] M. Bruynooghe and G. Janssens, "An Instance of Abstract Interpretation integrating Type and Mode Inference", *Proc.* 5^{th} *Int. Conf. on Logic Programming*, MIT Press, pp 669–683, August 1988.

[2] A. Cortesi, G. Filé and W. Winsborough, "Equivalence of Abstract Domains for Groundness Analysis", Technical Report, Dip. di Matematica Pure é Applicata, Universitá di Padova, 1991.

[3] S. Debray, "Global Optimization of Logic Programs", Ph.D. Thesis, Department of Computer Science, SUNY at Stony Brook, December 1986.

[4] N. Heintze, "Set Based Program Analysis", forthcoming Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1992 (expected).

[5] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proc.* 17^{th} *ACM-POPL*, 1990, pp. 197–209. (A full version of this paper appears as IBM technical report RC 16089 (# 71415), August 1990, 66 pp.)

[6] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proc.* 5^{th} *IEEE-LICS*, June 1990, pp 42 - 51. (A full version of this paper appears as Carnegie Mellon University technical report CMU-CS-91-110, February 1991, 42 pp.)

[7] D. Jacobs and A. Langen, "Accurate and Efficient Approximation of Variable Aliasing in Logic Programs", *Proc. North American Conf. on Logic Programming*, 1989, pp 154–165.

[8] N. Jones and H. Søndergaard, "A Semantics-based Framework for the Abstract Interpretation of PROLOG", in *Abstract Interpretation of Declarative*

Languages, (S. Abramsky and C. Hankin, Eds), Ellis Horwood, 1987, pp 123–142.

[9] K. Muthukumar and M. Hermenegildo, "Determination of Variable Dependence Information Through Abstract Interpretation", *Proc. North American Conf. on Logic Programming*, 1989, pp 166–186.

[10] K. Marriott and H. Søndergaard, "Bottom-Up Abstract Interpretation of Logic Programs" *Proc. Fifth International Conf. and Symp. on Logic Programming*, 1988, pp 733–748.

[11] K. Marriott and H. Søndergaard and N.D. Jones, "Abstract Interpretation of Logic Programs: The Denotational Approach", *ACM-TOPLAS*, to appear. (Also, *Proc. Fifth Italian Conf. Logic Programming*, 1990, pp 399-425.)

[12] H. Søndergaard, "An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction", in Proc. ESOP '86, Saarbrucken, March 86.

Appendix: Top-Down Analysis

Whereas semantic equations for bottom-up analysis collect a set of substitutions for each rule, top-down semantic equations collect information about a number of other "program points". Typically there is one point corresponding to each rule and one point corresponding to each body atom. The idea is that each point corresponding to a rule is assigned the set of substitutions obtained upon successful execution of the rule body. Similarly, each point corresponding to a body atom is associated with the set of substitutions encountered just before the atom is resolved. Another difference between top-down and bottom-up equations is that execution proceeds from a given initial goal (or set of goals). For technical convenience, it is simplest to distinguish a rule, say `main ← A1, ···, An`, which specifies that the initial goal is `?- A1, ···, An`.

We omit the details of top-down semantic equations, and instead illustrate them using the program

```
1.  append(nil, W, W).
2.  append(X.Xs,Y,X.Zs)←append(Xs,Y,Zs)[4].
3.  main ← goal[5].
```

where `goal` is some atom whose variables are denoted by $\mathcal{V}_3$, and $1-5$ denote program points. Variables Ψ_i are introduced to collect the substitutions encountered at each point. Ψ_2 collects the substitutions just after execution of the body atom in the second rule. Ψ_4 collects those substitutions encountered just before this body atom is called. The top-down equations are:

$$\begin{aligned}\Psi_1 = {} & \left\{ mgu\left(head_1 = \overline{\gamma(body)}\right) : \gamma \in \Psi_4 \right\}\Big|_{\mathcal{V}_1} \\ & \cup \left\{ mgu\left(head_1 = \overline{\gamma(goal)}\right) : \gamma \in \Psi_5 \right\}\Big|_{\mathcal{V}_1}\end{aligned}$$

$$\begin{aligned}\Psi_2 = {} & \left\{ mgu\begin{pmatrix} head_2 = \overline{\gamma_1(body)} \\ body = \overline{\gamma_2(head_1)} \end{pmatrix} : \begin{matrix} \gamma_1 \in \Psi_4 \\ \gamma_2 \in \Psi_1 \end{matrix} \right\}\Bigg|_{\mathcal{V}_2} \\ & \cup \left\{ mgu\begin{pmatrix} head_2 = \overline{\gamma_1(body)} \\ body = \overline{\gamma_2(head_2)} \end{pmatrix} : \begin{matrix} \gamma_1 \in \Psi_4 \\ \gamma_2 \in \Psi_2 \end{matrix} \right\}\Bigg|_{\mathcal{V}_2} \\ & \cup \left\{ mgu\begin{pmatrix} head_2 = \overline{\gamma_1(goal)} \\ body = \overline{\gamma_2(head_1)} \end{pmatrix} : \begin{matrix} \gamma_1 \in \Psi_5 \\ \gamma_2 \in \Psi_1 \end{matrix} \right\}\Bigg|_{\mathcal{V}_2} \\ & \cup \left\{ mgu\begin{pmatrix} head_2 = \overline{\gamma_1(goal)} \\ body = \overline{\gamma_2(head_2)} \end{pmatrix} : \begin{matrix} \gamma_1 \in \Psi_5 \\ \gamma_2 \in \Psi_2 \end{matrix} \right\}\Bigg|_{\mathcal{V}_2}\end{aligned}$$

$$\begin{aligned}\Psi_3 = {} & \left\{ mgu\left(goal = \overline{\gamma(head_1)}\right) : \gamma \in \Psi_1 \right\}\Big|_{\mathcal{V}_2} \\ & \cup \left\{ mgu\left(goal = \overline{\gamma(head_2)}\right) : \gamma \in \Psi_2 \right\}\Big|_{\mathcal{V}_3}\end{aligned}$$

$$\begin{aligned}\Psi_4 = {} & \left\{ mgu\left(head_2 = \overline{\gamma(body)}\right) : \gamma \in \Psi_4 \right\}\Big|_{\mathcal{V}_2} \\ & \cup \left\{ mgu\left(head_2 = \overline{\gamma(goal)}\right) : \gamma \in \Psi_5 \right\}\Big|_{\mathcal{V}_2}\end{aligned}$$

$$\Psi_5 = \{mgu()\}|_{\mathcal{V}_3}$$

Finally, we informally outline the standard engine which idealizes the commonly used algorithms for top-down analysis. Let there be n predicates. Let $pred(A)$ map an atom A to the predicate of A. The function $\mathcal{D}\text{-}unify(s, t, \mathcal{A}_1, \mathcal{A}_2)$ returns the abstract unification of s in the context of $\mathcal{A}_1$ and t in the context of $\mathcal{A}_2$.

initialize the arrays $value[1..N]$ and $call[1..n]$;
loop {
  for $(1 \leq i \leq N)$ {
    let rule i be $A \leftarrow \tilde{B}$ where $\tilde{B}$ is $B_1, \cdots, B_m$;
    $newvalue[i] :=$
$$\bigcup_{\tilde{H}} \mathcal{D}\text{-}unify(\tilde{B}, \tilde{H}, call[pred(A)], value[\,]),$$
      where $\tilde{H}$ ranges over heads which, when suitable renamed, match $\tilde{B}$;

    $newcall[pred(B_j)] :=$
$$\bigcup_{\tilde{H}} \mathcal{D}\text{-}unify\begin{pmatrix} (B_1, \cdots, B_{j-1}), \tilde{H}, \\ call[pred(A)], value[\,] \end{pmatrix},$$
      where $1 \leq j \leq m$ and $\tilde{H}$ ranges over heads which, when suitably renamed, match $B_1, \cdots, B_{j-1}$;
  }
  if $(newvalue[\,] = value[\,]\ \&\ newcall[\,] = call[\,])$
    break;
  $value[\,] := newvalue[\,]$;
  $call[\,] := newcall[\,]$;
}
output $value[\,]$ and $call[\,]$;