Mode Analysis for Logic Programs

Saumya K. Debray
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

Motivation

Prolog programs are, in general, "non-directional."

Example : The following procedure to concatenate lists can be executed in many different ways:

```
app([], L, L).
app([H|L1], L2, [H|L3]) :- app(L1, L2, L3).
```

1. Forward, to concatenate two given lists:

```
?- app([1,2,3], [4,5], X).
X = [1,2,3,4,5].
yes
```

2. Backward, to split up a given list:

```
?- app(X, Y, [1,2,3]).
X = [], Y = [1,2,3] ;

X = [1], Y = [2,3] ;
```

   etc.

3. As a test on three given lists:

```
?- app([1,2,3], [4,5], [1,2,3,4,5]).
yes.

?- app([1,2,3], [4,5], [6,7,8,9,0]).
no.
```


Flexibility vs. Efficiency

- The non-directionality of Prolog procedures allows for simple programming and flexible code reuse.
- However, there is a runtime cost in terms of additional tests that must be executed.
- Experiments show that in almost all cases, this generality is not utilized.

  Programmers use some arguments of a procedure consistently as input arguments, others as outputs.
- If we can identify such consistent usage in a program, the resulting code can be optimized to improve runtime performance:
  - Reduce the no. of unnecessary tests performed at runtime.
    $\Rightarrow$ improve speed.
  - Eliminate code for such tests.
    $\Rightarrow$ improve code size.

Modes

- A *mode* for a n-ary predicate p in a program is an n-tuple that describes the "instantiation" of particular arguments in every call to p.
- *Example 1* : In the Dec-10 Prolog compiler by D. H. D. Warren, which introduced modes to enable certain optimizations, a mode was an element of $\{\texttt{+, - ?}\}^n$

  where + ≡ non-variable;
  - ≡ variable;
  ? ≡ "don't know", i.e., any term.
- *Example 2* : Many researchers consider the set of modes $\{\texttt{g}, \texttt{f}, \texttt{nv}, ?\}$, where

  g ≡ ground;
  nv ≡ non-variable;
  f ≡ variable;
  ? ≡ "don't know", i.e., any term.

Early Attempts 1: Mellish 1981

- Constructed a set of (recursive) equations relating the instantiations of different variables.
- Initial formulation *ad hoc*; subsequently [Mellish 86] reconstructed in terms of abstract interpretation.
- Did not account for *aliasing*.

  As a result, was not sound.
- *Problem* : Consider the program

  ```
  p :- q(X, Y), r(X), s(Y).
  q(U, U).
  r(a).
  s(_).
  ```

  - s/1 is called with its argument bound to the ground term a.
  - Unless aliasing (within q/2) is accounted for, we don't detect this.

    Mellish's algorithm would conclude that s/1 is called with a variable as an argument.

Early Attempts 2: Debray 1987

- Recognized the problem with aliasing.
- Considered two kinds of aliasing :
  1. *Input Aliasing* :

     ```
     p :- q(X, X).
     q(a, Y) :- r(Y).  %% Y ground in r(Y)
     r(_).
     ```
  2. *Output Aliasing* :

     ```
     p :- q(X, Y), r(X), s(Y).  %% Y ground in s(Y).
     q(U, U).
     r(a).
     s(_).
     ```
- Proposed (ad hoc) "safety conditions" for dealing with possible aliases.
- Formulated in terms of abstract interpretation.

  However, because of the ad hoc nature of the safety rules, this was not a clean semantic formalization.
- Proposed "abstract compilation" as a technique to speed up analyses.

Early Attempts 3: Debray 1988

- The problem with aliasing is really that the set

  $$\{x \mid x \text{ is an unbound variable}\}$$

  is not closed under the operation of substitution.
- If we use an abstract domain $\mathcal{D}$ that is *substitution-closed*, i.e.,

  $$(\forall d \in \mathcal{D})(\forall t \in \mathsf{Term})[\ t \in d \Rightarrow (\forall \theta \in \mathsf{Subst})\ \theta(t) \in d\ ]$$

  then the soundness problem disappears.
- *Example* : $\mathcal{D} = \{$**g**, **nv**, **?**$\}$, where
  - **g** = set of all ground terms
  - **nv** = set of nonvariable terms
  - **?** = set of all terms.
- Substitution-closed domains allow aliasing effects to be ignored without affecting soundness.

  Because of this, under reasonable assumptions they admit linear-time analysis algorithms.
- *Disadvantage* : They cannot express the property *"X is an unbound variable."*

Subsequent Evolution of Mode Analysis

- Better understanding of the aliasing problem:
  - The basic problem is "action at a distance" due to sharing of variables between terms.

    Variable aliasing is a special case of this.
  - The property

    "x and y share some variable"

    is different from the property

    "x is a free variable."
- The property *"x is free"* is fundamentally different (semantically) from *"x is ground"*.
- (Possible) Sharing information can be propagated without necessarily maintaining freeness information.
- This understanding gave rise to a variety of different groundness analyses:
  - dependency sets [Debray 89, Janssens 90]
  - *Sharing* [Jacobs & Langen 89, 92]
  - *Prop* [Marriott & Søndergaard 89]

The Domain *Sharing* [Jacobs & Langen 89]

- Describes possible sharing of variables between terms.
- $Sharing = \mathcal{P}(\mathcal{P}(\mathsf{Var}))$.
  Each element is a set of sets of variables.
- A variable u *occurs through* a variable v in a substitution σ if $u \in vars(\sigma)$.
  $occ(\sigma, u) = \{v \mid u \in vars(\sigma)\}$.
- *Example* : Let
  $$\theta = \{W \mapsto A, X \mapsto A, Y \mapsto B, Z \mapsto f(A, B)\}.$$
  Then:
  $$occ(\theta, A) = \{W, X, Z\}$$
  $$occ(\theta, B) = \{Y, Z\}$$

The Domain *Sharing* : cont'd

Definition : The sharing $\mathcal{A}(\sigma)$ that describes a substitution σ is
$$\mathcal{A}(\sigma) = \{occ(\sigma, u) \mid u \in rgv(\sigma)\}$$
where $rgv(\sigma)$ is the set of variables appearing in the range of σ.

For a set of substitutions Θ:
$$\mathcal{A}(\Theta) = \bigcup_{\theta \in \Theta} \mathcal{A}(\theta).$$

Example :

- *Concrete Substitution* :
  $$\theta = \{W \mapsto A, X \mapsto A, Y \mapsto B, Z \mapsto f(A, B)\}.$$
- *Abstract Substitution* :
  $$\mathcal{A}(\{\theta\}) = \{\ \{W, X, Z\},\ \{Y, Z\}\ \}.$$

Properties of the Domain *Sharing*

Sharings capture groundness and aliasing information:

- A variable is grounded by σ if it does not appear in any set in $\mathcal{A}(\sigma)$.
- Variables x, y are aliased by σ if they appear together in some set in $\mathcal{A}(\sigma)$.
- Given a sharing S, its *closure under union* S^*, approximates all further instantiations of substitutions described by S.
  $\mathcal{A}(\{\theta\})^* = \{\ \{W, X, Z\},\ \{Y, Z\},\ \{W, X, Y, Z\}\ \}$.

The Domain *Sharing* : Abstract Unification

Abstract unification of terms t_1, t_2 under sharing S :

1. If t_1 and t_2 not unifiable then $\emptyset$.
2. If at least one of t_1 and t_2 is a variable then:
   Let $rel(t, S) = \{X \in S \mid vars(t) \cap X \neq \emptyset\}$.
   (a) Let S_1 be the sharing due to the unification of t_1 and t_2 :
   $$S_1 = (rel(t_1, S) \oplus rel(t_2, S))^*$$
   where $A \oplus B = \{X \cup Y \mid X \in A, Y \in B\}$.
   (b) Let S_2 be the sharing unaffected by the unification of t_1 and t_2 :
   $$S_2 = S - (rel(t_1, S) \cup rel(t_2, S)).$$
   (c) return $S_1 \cup S_2$.
3. If $t_1 = f(a_1, \ldots, a_n)$ and $t_2 = f(b_1, \ldots, b_n)$ then:
   $$S_0 = S;$$
   $$S_{i+1} = abstract\_unify(a_{i+1}, b_{i+1}, S_i).$$
   Return S_n.

Jacobs & Langen: Analysis Framework

- The abstract meaning of a predicate is a function from an "input" sharing to an "output" sharing.
- Since a predicate is defined once but called from many places, it may be more efficient to:
  - Compute the abstract meaning of a predicate p once, for the "most general" case (i.e., all input arguments are distinct variables).
  - The meanings of different invocations of p can then be obtained from this using *abstract unification*, without computing a fixpoint.
- *Disadvantage* : The technique seems to be applicable to only a limited class of analyses.
- The condensing step does not cause a loss in precision if abstract operations satisfy certain algebraic properties:
  - abstract unification is commutative (reordering unifications does not lose information); and
  - abstract unification is additive (LUB operations do not lose information).

Sharing Analysis [Muthukumar-Hermenegildo 89, 90]

Intended Application : automatic parallelization of Prolog programs.

Aim of Analysis : information about sharing between variables in a clause.

Abstract Domain : *Sharing*, similar to Jacobs & Langen 89.

Main difference with Jacobs & Langen 89:

- The analysis of Muthukumar & Hermenegildo is top-down, compared to the bottom-up analysis of Jacobs & Langen.
- Jacobs & Langen use *condensation* while Muthukumar & Hermenegildo use Extension Tables in an explicit fixpoint construction.

The Domain *Prop* [Marriott & Søndergaard 89]

- Provides information about groundness of variables and sharing between variables.
- *Abstract Domain* : equivalence classes (w.r.t. logical equivalence) of propositional formulae described by

  $$F ::= \mathbf{false} \mid \mathbf{true} \mid F \wedge F \mid F \vee F \mid F \leftrightarrow F$$

- F describes a substitution θ iff the truth assignment given by

  $V \mapsto \mathbf{true}$ iff $\theta(V)$ is a ground term

  satisfies F.

The Domain *Prop* : Examples

$F \equiv x \leftrightarrow y$ describes:

$[x \mapsto \mathtt{a}, y \mapsto \mathtt{f}(\mathtt{b})]$

and

$[x \mapsto z, y \mapsto \mathtt{f}(z, z)]$

but not

$[x \mapsto \mathtt{a}]$

$F \equiv x \wedge (y \vee z)$ describes

$[x \mapsto \mathtt{f}(\mathtt{b}), y \mapsto u, z \mapsto \mathtt{a}]$

$[x \mapsto \mathtt{f}(\mathtt{b}), y \mapsto \mathtt{a}, z \mapsto u]$

$[x \mapsto \mathtt{b}, y \mapsto \mathtt{g}(\mathtt{a}, \mathtt{b}), z \mapsto \mathtt{a}]$

but not

$[x \mapsto \mathtt{f}(\mathtt{b}), y \mapsto u, z \mapsto u]$

The Domain *Prop* : Abstract Unification

$\widehat{mgu}(A, H)$:

> **let** $\sigma = mgu(A, H)$ **in**
> **if** $\sigma =$ *fail* **then false**
> **else**
> $\quad \bigwedge\{v \leftrightarrow (\wedge\{v' \mid v' \in vars(\sigma(v))\}) \mid v \in dom(\sigma)\}$
> **fi**

$\widehat{unify}(H, A, F)$: A an atom, $F \in Prop$:

1. $\rho :=$ substitution for A, F renamed apart from H;
2. $F' = \rho(F) \wedge \widehat{mgu}(H, \rho(A))$;
3. **return** $project(F', vars(H))$.

Example :
Let $A =$ `app(X, Y, Z)`, $H =$ `app([], L, L)`, then:

$$\widehat{unify}(H, A, [\mathbf{true}]) = [\mathbf{true}]$$

$$\widehat{unify}(A, H, [\mathbf{true}]) = [\mathtt{X} \wedge (\mathtt{Y} \leftrightarrow \mathtt{Z})]$$

Groundness Analysis Using *Prop*

Analyzing a Predicate : Given a "call description" F,

- analyze the clauses $C_1, \ldots, C_n$ w.r.t. F; let the results be $F_1, \ldots, F_n$;
- return $\bigvee\{F_1, \ldots, F_n\}$.

Analyzing a Clause : Given a "call description" F,

- Carry our head unification w.r.t. F;
- Analyze body literals from left to right;
- Project the resulting "exit formula" on the head variables.

Example :

```
app(X,Y,Z) :- X = [], Y = Z.
app(X,L2,Z) :-
  X = [H|L1], Z = [H|L3], app(L1,L2,L3).
?- app(X, Y, Z).
```

Iteration	Analysis	Result
0	**false**	**false**
1	$(\mathtt{X} \wedge \mathtt{Y} \leftrightarrow \mathtt{Z}) \vee \mathbf{false}$	$\mathtt{X} \wedge \mathtt{Y} \leftrightarrow \mathtt{Z}$
2	$(\mathtt{X} \wedge \mathtt{Y} \leftrightarrow \mathtt{Z}) \vee (\mathtt{X} \wedge \mathtt{Y} \leftrightarrow \mathtt{Z})$	$\mathtt{X} \wedge \mathtt{Y} \leftrightarrow \mathtt{Z}$

Goal-Independent Analysis using *Prop*

Codish & Demoen [1993] describe a goal-independent analysis based on *Prop*:

- very simple and efficient;
- uses abstract compilation + magic sets transformation (for goal-dependent analysis).

General Approach : Translate the input program into a Prolog program that, when executed, will produce the results of analysis.

Goal-Independent Analysis : Compute the least Herbrand model of the generated program.

(Requires some additional machinery to halt when the fixpoint has been computed.)

Goal-Dependent Analysis : Use magic sets transformation.

Goal-Independent Analysis using *Prop* : Example

Input Program :

```
app(X,Y,Z) :- X = [], Y = Z.
app(X,Y,Z) :- X = [H|L1], Z = [H|L3], app(L1,Y,L3).
```

Generated Program :

```
app(X,Y,Z) :- X ↔ true, Y ↔ Z.
app(X,Z,Z) :-
   X ↔ (H ∧ L1) ∧ Z ↔ (H ∧ L3) ∧ app(L1, Y, L3).
```

Naive Prolog Representation :

```
app(X,Y,Z) :- iff(X, []), iff(Y,[Z]).
app(X,Y,Z) :-
  iff(X, [H,L1]), iff(Z, [H,L3]), app(L1,Y,L3).
```

Optimized Prolog Representation :

```
app(X,Y,Z) :- iff(X), iff(Y, Z).
app(X,Y,Z) :-
  iff(X,H,L1), iff(Z,H,L3), app(L1,Y,L3).
```

Applying the Magic Sets Transformation

- Goal-dependent analyses can be handled by applying the "Magic Sets" transformation introduced for deductive databases.

- *Basic Idea* : Introduce additional predicates that capture the top-down execution of Prolog.

  Example :

```
query_app(X,Y,Z) :- top_lev_query_app(X,Y,Z).
query_app(L1,Y,L3) :-
  query_app(X,Y,Z), iff(X,H,L1), iff(Z,H,L3).
```

- The effect is similar to that of condensing [Jacobs & Langen 89].

- *Performance* :
  - About 3× slower than the optimized C implementation of Le Charlier & Van Hentenryck.
  - Speed comparable to the Toupie system [Corsini *et al.* 93].
  - Very simple and flexible.

Freeness Analysis

- Domains such as *Sharing* and *Prop* can express sharing between terms.

  However, they cannot express properties such as

  "X is an unbound variable."

- *Dependency Sets* [Debray 89, Janssens 90] allow freeness properties to be expressed.

- Why is knowledge of freeness useful?
  - Unification can be made more efficient.
  - Parallel execution can be made more efficient:

```
matmult([V0|Vs],V1,[R|Rs]) :-
  dot_prod(V0,V1,R), matmult(Vs,V1,Rs).
```

    Suppose we know: `matmult(ground,ground,free)`.
    Then `R` and `Rs` are unaliased.
    So the two body goals can be executed in parallel unconditionally.

Dependency Sets

Basic Idea : Each abstract domain element consists of two parts:

1. An *instantiation* component: describes the set of terms being referred to.
2. A *dependency set*: gives the set of variables it shares with.

Example : Consider

$$\theta = \{X \mapsto a,\ Y \mapsto Z,\ Z \mapsto Z,\ U \mapsto f(X,Y,Z)\}.$$

This is described by:

$$\begin{aligned} &X \mapsto (\mathtt{ground}, \emptyset)\\ &Y \mapsto (\mathtt{free}, \{Z\})\\ &Z \mapsto (\mathtt{free}, \{Y\})\\ &U \mapsto (\mathtt{nonvar}, \{Y, Z\}).\end{aligned}$$

Debray 89 : instantiation domain of *modes* :

$\{\mathtt{ground},\ \mathtt{free},\ \mathtt{nonvar},\ \mathtt{any}\}$.

Janssens 90 : instantiation domain of *type graphs* : trees describing the structure of sets of terms.

Propagating Freeness using Dependency Sets

Consider a unification $X = Y$:

X free, Y free :

Before	*After*
$X \mapsto (\mathtt{free}, S_x)$	$X \mapsto (\mathtt{free}, S_x \cup S_y \cup \{Y\})$
$Y \mapsto (\mathtt{free}, S_y)$	$Y \mapsto (\mathtt{free}, S_x \cup S_y \cup \{X\})$

X free, Y ground :

Before	*After*
$X \mapsto (\mathtt{free}, S_x)$	$X \mapsto (\mathtt{ground}, \emptyset)$
$Y \mapsto (\mathtt{ground}, \emptyset)$	$Y \mapsto (\mathtt{ground}, \emptyset)$

Propagate : For every Z s.t. $Z \mapsto (\mathtt{free}, S_z)$ and $X \in S_z$, update to $Z \mapsto (\mathtt{any}, S_z - \{X\})$.

X free, $Y \notin \{\mathtt{free}, \mathtt{ground}\}$:

Before	*After*
$X \mapsto (\mathtt{free}, S_x)$	$X \mapsto (\mathtt{any}, S_x \cup S_y \cup \{Y\})$
$Y \mapsto (\mathtt{t}, S_y)$	$Y \mapsto (\mathtt{t}, S_x \cup S_y \cup \{X\})$

Propagate : For every Z s.t. $Z \mapsto (\mathtt{free}, S_z)$ and $X \in S_z$, update to $Z \mapsto (\mathtt{any}, S_z - \{X\})$.

Freeness in the Domain *Sharing*

- *Sharing* extended to accommodate freeness information by Muthukumar & Hermenegildo [ICLP 91].

- *Basic Idea* : Similar to dependency sets.

  Abstract substitutions described via a pair $(Inst, Shr)$ where

  – $Inst \in \{\texttt{ground}, \texttt{free}, \texttt{any}\}$ describes the instantiation of the term;

  – Shr is a set of sets of variables describing possible sharing.

- Because of the more complex structure of the domain *Sharing*, abstract unification becomes considerably more complex.

- Sharing and freeness information interact positively: each component can be used to improve the precision of the other.

Applications

- Speeding up unification: Van Roy *et al.* [TAPSOFT 87] estimate a speed improvement of ~10%.

- Inferring determinacy. E.g.:

```
rev([], L, L).
rev([H|L], S, R) :- rev(L, [H|S], R).
```

  Knowing the mode (`ground`, ...) for `rev/3` allows us to infer it is deterministic.

  ⇒ Simplify, speed up clause indexing code.

- Parallel execution: &-Prolog system [Hermenegildo 90].

Performance figures [Mariën *et al.*, ICLP 89] :

Pgm	Normal	+modes	+types	+length	+destr
nrev	100	51	41	41	41
insert	100	72	54	41	42
sumlist	100	85	44	31	–