Analysis of Logic Programs

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

Analysis of Logic Programs: A Brief History

Early 1980s : specific (simple) analyses:

Mode analysis [Mellish 81, Debray & Warren 87]; data dependence analysis [Chang & Despain 85]; occur check analysis [Søndergaard 86].

Late 1980s : general frameworks:

Bruynooghe 87, Jones & Søndergaard 87, Marriott & Søndergaard 88, Barbuti *et al.* 90, Marriott, Søndergaard & Jones 90.

Late 1980s–Early 1990s : implementations :

Hermenegildo, Warren & Debray 88, Van Roy 90, Taylor 90, van Hentenryck *et al.* ...

Hermenegildo *et al.* ..., Jacobs & Langen 89, Santos Costa 93

Current Topics : parallelism, constraint logic programs.

Analyzing Logic Programs: Main Issues

- Which fixpoint to approximate?
  - least fixpoint (most analyses);
  - greatest fixpoint (e.g., finiteness analysis).
- Choice of concrete semantics:
  - denotational
  - operational : SLD-resolution ("top-down analyses")
  - fixpoint : T_P ("bottom-up analyses")
  - what about negation?
- Approximating the effects of unification.
- Approximating the meaning of a program.
- Ensuring termination:
  - usually, ban infinite chains in the abstract domain;
  - sometimes, resort to some form of widening.

Which Fixpoint to Approximate?

- *least fixpoint*, "from below" :
  - intermediate results may not give a sound approximation: the analysis *must* iterate to convergence;
  - (potentially) more precise, more expensive;
  - used by most analyses.
- *greatest fixpoint*, "from above" :
  - intermediate results give sound approximations: the analysis need not iterate to convergence [Codish *et al.* 88, Taylor 89];
  - may simplify some analyses (e.g., superfiniteness analysis [Kifer *et al.* 88]);
  - may be imprecise in dealing with recursion;
  - (potentially) more efficient.

Choice of Concrete Semantics

Denotational : Jones & Søndergaard 87, Winsborough 88, Jacobs & Langen 92;

Operational :

- SLD-derivations : Bruynooghe 91, Janssens & Bruynooghe 92, Muthukumar & Hermenegildo 92;
- transition system semantics [concurrent logic programs] : Codish, Falaschi & Marriott 91;

Fixpoint (based on T_P operator) : Mishra 84, Marriott & Søndergaard 88, Barbuti *et al.* 93;

Top-down Analyses :

- based on top-down, i.e., query-dependent, semantics (e.g., denotational, SLD-based.);
- essentially, *forward dataflow analyses*.

Bottom-up Analyses :

- based on bottom-up, i.e., query-independent, semantics (e.g., model-theoretic, T_P-based.);
- essentially, *backward dataflow analyses*.

Approximating the Effects of Unification

- Must cover all runtime possibilities (soundness).
- Structural issues:
  - groundness analysis: "degree of instantiation";
  - type analysis: function symbols, recursive types;
  - cost, termination analyses: term size.
- Aliasing and inter-variable dependencies:
  - all may-aliases must be accounted for;
  - can use must-alias information to improve precision [Janssens 90];
  - can be thought of in terms of *entailment* of constraints between terms [Hermenegildo 93].

Approximating the Effects of Unification (cont'd)

Consider a concrete domain $\langle \mathcal{D}_{conc}, \sqsubseteq \rangle$, and an abstract domain $\mathcal{D}_{abs}$:

- *downward closure* : $\mathcal{D}_{abs}$ is downwards closed if:

  $\forall y \in \mathcal{D}_{abs} : x \in conc(y)$ and $x' \sqsubseteq x \Rightarrow x' \in conc(y)$.

  Such domains are typically used for groundness analysis, type analysis, definite aliasing analysis, and definite sharing analysis.

- *upward closure* : $\mathcal{D}_{abs}$ is upwards closed if:

  $\forall y \in \mathcal{D}_{abs} : x \in conc(y)$ and $x' \sqsupseteq x \Rightarrow x' \in conc(y)$.

  Such domains are typically used for possible sharing (independence) analysis, possible aliasing analysis, and freeness analysis.

- Many complex analyses, e.g., mode inference or detection of AND-parallelism, can be expressed in terms of combinations of downward- and upward-closed domains [Marriott *et al.* 93].

Approximating Program Behaviors

- Most approaches use iterative techniques to compute the limit of a Kleene sequence.
- Iterative techniques may not always be suitable (or desirable):
  - solving difference equations during cost analysis [Debray & Lin 91];
  - solving type inequations [Mishra 84, Pyo & Reddy 89];
  - solving set constraints [Heinze & Jaffar 90, 92; Heinze 92].
- Termination considerations force iterative techniques to require that only finitely many values be considered during analysis. This can limit their precision.

  Where applicable, non-iterative techniques can give more precise results.

Dataflow Analysis of Logic Programs: Soundness

- Need to define appropriate relationships between abstract and concrete domains (Galois connection, widening/narrowing operators, etc.).
- Need a complete evaluation strategy for logic programs, i.e., one that:
  - (i) examines all possible execution paths; and
  - (ii) terminates in finite time if the "abstract least model" of the program is finite.

  Examples of complete evaluation strategies (for Datalog) :
  - bottom-up evaluation (poss. with Magic Sets rewriting)
  - top-down evaluation with extension tables.

Characterizing Program Analyses

1. *Direction* (w.r.t. control flow) :
   (a) *Forward Analysis* : Properties at a program point are defined in terms of program properties at earlier program points.
   Example : reaching definitions.

   (b) *Backward Analysis* : Properties at a program point are defined in terms program properties at later (w.r.t. control flow) program points.
   Example : liveness analysis.

2. *Quantification* (w.r.t. computation paths) :
   (a) $\forall$ *Analysis* : Properties that must hold along *every* computation branch.
   Example : available expressions.

   (b) $\exists$ *Analysis* : Properties must hold along *some* computation branch.
   Example : liveness analysis.

Characterizing Analyses of Logic Programs

1. *Direction* (w.r.t. control flow) :
   (a) *Top-down Analysis* : $\sim$ forward analysis.
   Goal dependent
   Based on operational semantics (SLD, . . .).
   Example: mode analysis, calling type analysis, determinacy analysis.

   (b) *Bottom-up Analysis* : $\sim$ backward analysis.
   Goal independent
   Based on a model-theoretic semantics (T_P)
   Example : success pattern analysis, success type analysis, floundering analysis (NAF).

2. *Quantification* (w.r.t computation paths) :
   (a) $\forall$ *Analysis* : groundness, freeness, types, definite aliasing, independence.

   (b) $\exists$ *Analysis* : possible aliasing (dependence).

Note : In contrast to traditional compiler analyses, most analyses for logic programs are a combination of $\forall$- and $\exists$-analyses, e.g.: mode inference, parallelism detection, etc.

Framework 1 : Bruynooghe

Concrete Semantics : based on (ordered) AND trees.

Abstract Semantics : Control flow is abstracted using *Abstract AND-OR trees*; sets of substitutions are abstracted using *abstract substitutions*.

To ensure termination, the finite-chain property is obtained by allowing "back edges" (*rational* abstract AND-OR trees).

Abstract Operations : 5 primitive operations on abstract trees:

- initialization;
- procedure entry;
- procedure exit;
- abstract interpretation of built-ins;
- abstract interpretation of a call with a given abstract substitution.

Bruynooghe's Framework : cont'd

Concrete AND-trees

- Basically, an AND-tree is a proof tree that represents a (partial) SLD derivation.
- Given a query Q (assumed to be an atom), the root of the tree is initially labelled by $\langle Q, \epsilon \rangle$.
- Each node is labelled with a pair

  $\langle A, \sigma \rangle$

  where A is a literal and σ is a substitution.
- If there is a node n labelled by $\langle A, \sigma \rangle$, and a variant of a program clause

  $H \ :- \ B_1, \ldots, B_n$

  with $\theta = mgu(H, \sigma(A)) \neq fail$, then
  - n has as children the nodes labelled

    $\langle B_i, \theta \rangle, \quad 1 \leq i \leq n$
  - the substitution θ is composed into every node in the tree.
- A computation is represented by a series of AND trees.

Generalized AND-trees

- *Observation* : To characterize the procedural behavior of programs, it suffices to know the instance of procedure calls immediately before their execution, and immediately after their completion.
- *Generalized AND-trees* :
  - represents a sequence of AND trees;
  - procedure call effects characterized by their instances immediately before/after the call;
  - nodes can be adorned with substitutions on the *left* ("call substitutions") and the *right* ("success substitutions");
  - constructed by repeated applications of three operations: *procedure entry*, *procedure exit*, and *interpretation of built-ins*.

Constructing Generalized AND-trees

Notation for Nodes : $\langle \theta, A : H, \psi \rangle$, where A is a call; θ is the call substitution (optional); H is a clause head (optional); and ψ success substitution (optional).

Initialization : assume an initial generalized AND-tree $\langle \theta, A, - \rangle$.

Procedure Entry : Given a tree with a leaf node $n \equiv \langle \theta, A, - \rangle$, and a (properly renamed) program clause $C \equiv$ '$H \ :- \ B_1, \ldots, B_n$' defining A, where A and H are unifiable:

1. the node n becomes $\langle \theta, A : H, - \rangle$;
2. $B_1, \ldots, B_n$ are added as children of n, where B_1 has the call substitution

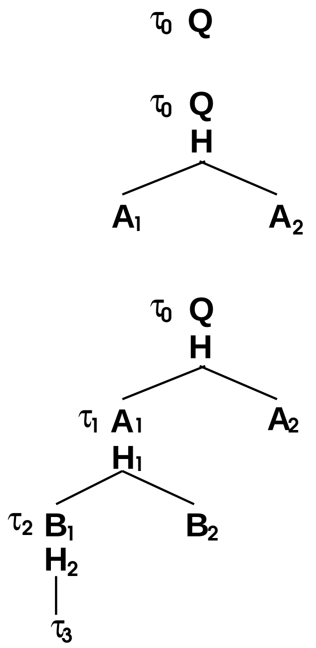
   $\psi = (\sigma \circ \theta) \mid_{vars(C)}$, where $\sigma = mgu(A, H)$.

   If $n = 0$ then ψ is the *success substitution*.

Interpretation of Built-ins : Given a tree with a leaf node $n \equiv \langle \theta, A, - \rangle$, where A is a built-in: n is transformed to $\langle \theta, A, \sigma \rangle$, where σ is the *success substitution* for A.

If A is the last call of its clause, then σ is the success substitution for the clause; else σ is the call substitution for the next call in the body.

Example :



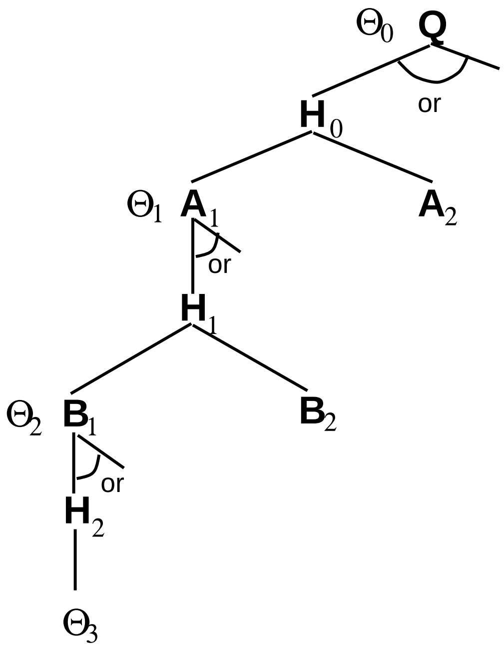
Abstract AND-OR Trees

- Represents a set of generalized AND-trees.
- Single substitutions are replaced by (possibly infinite) sets of substitutions.

  Given a set of substitutions Θ, a node $\langle \Theta, A, - \rangle$ represents the set of generalized AND-tree nodes

  $$\{\langle \theta, A, - \rangle \mid \theta \in \Theta\}.$$
- Nondeterminism, via alternative clauses for a predicate, is handled by introducing OR-nodes.
- *Operations on Abstract AND-OR Trees* :
  - abstract procedure entry;
  - abstract procedure exit;
  - abstract interpretation of built-ins.
- Construction of abstract AND-OR trees becomes nondeterministic, since there is no search rule, and operations are possible in different OR-branches.

Example :



Abstract Interpretation using AND-OR Trees

Basic Issues : Given a set of queries $\Theta(Q)$, construct an abstract AND-OR tree that is

1. *correct*, i.e., describes all concrete AND-trees that can occur during the execution of any query in $\Theta(Q)$;
2. as precise as possible;
3. constructable in a finite amount of time.

Approach : Define a procedure for construction of Abstract AND-OR trees using three basic operations:

1. procedure entry;
2. procedure exit; and
3. abstract interpretation of built-ins.

Soundness : The global correctness of the resulting AND-OR tree is ensured by giving sufficient conditions for each of these operations.

Intuitively, these represent the correctness requirement for each abstract operation in the Galois conection development of Cousot & Cousot.

Termination

Issues : Need finite representations for:

- *Infinite sets of substitutions* : this is done using *abstract substitutions* that give finite (upper) approximations to infinite sets of substitutions.
  - The set of concrete substitutions Θ represented by an abstract substitutions S is given by a function

    $$\gamma : \mathit{AbstractSubst} \longrightarrow \mathcal{P}(\mathit{Subst})$$

    such that $\Theta = \gamma(S)$.
  - Union of sets of concrete substitutions is approximated by an operation $\sqcup$ on abstract substitutions such that

    $$\gamma(S_1) \cup \gamma(S_2) \subseteq \gamma(S_1 \sqcup S_2).$$
- *Infinite AND-OR trees* : this is done using finite (cyclic) graphs to approximate infinite trees. These graphs are called *rational trees*.

  This approximation can be done in many different ways, e.g.: in a type inference system for Prolog, [Bruynooghe *et al.*, 1987] allows only one occurrence of any functor along any acyclic path starting from the root of a type graph.

Framework 2: Kanamori & Kawamura

Based on OLDT-resolution, i.e., Prolog augmented with memoization.

Basic Idea :

- To analyze a Prolog program, we need to consider all possible executions.
- Particular executions can be considered using Prolog's search strategy, i.e., using a top-down L-to-R interpreter.

  However, because Prolog's search strategy is incomplete, this may not give all possible executions.
- We need to augment Prolog's search strategy with a component that will allow all execution paths to be explored.

  I.e., we need a top-down search strategy that is complete.
- *Possibilities* : depth-bounded iterative deepening, resolution + memoization.

  Kanamori and Kawamura use resolution + memoization (OLDT).

OLDT Resolution (Extension Tables)

- For each predicate p, maintain a table that maps each call to p with the set of answers computed for that call.
- At any call $p(\bar{t})$ to p, look up the table:
  - If (a variant of) $p(\bar{t})$ is in the table and the set of answers $S \neq \emptyset$, return S.

    The original computation continues with each element of S.
  - If (a variant of) $p(\bar{t})$ is in the table but the set of answers $S = \emptyset$ then suspend (or fail+iterate).
  - If $p(\bar{t})$ is not in the table, enter it in the table with the empty set of solutions, then solve $p(\bar{t})$.
- *Solving a goal $p(\bar{t})$* : for each clause 'H :− B' do:
  - resolve $p(\bar{t})$ against the head H;
  - solve the body B using OLDT-resolution;
  - if the computation succeeds with substitution θ, enter $\theta(H)$ into the table as an answer for the call $p(\bar{t})$;
  - "awaken" any suspended computation for $p(\bar{t})$ to return $\theta(H)$.

Abstract Interpretation Using OLDT Resolution

- Essentially similar to concrete computation, but use abstract operations for unification, etc.
- Termination can be guaranteed if the abstract domain is finite.
- The analysis may not terminate if the abstract domain is infinite, even if it satisfies the finite-chain property.

  Example : Consider the program

  $p(X)$:− $p([X])$.
  $?-p(0)$.

  This generates an infinite sequence of mutually incomparable calling patterns

  $p(\mathbf{int}), p(\mathbf{list}(\mathbf{int})), p(\mathbf{list}(\mathbf{list}(\mathbf{int}))), \ldots$
- This problem can be handled using "depth abstraction" on the abstract call and success patterns.

Termination Issues

- Usual approach: disallow infinite chains in the abstract domain (*Noetherian domains*).
- Alternative: allow infinite chains, use widening to ensure termination.

Infinite chains can sneak in unexpectedly :

A suspension analysis by Codish, Falaschi & Marriott [1991] uses a finite lattice to describe all possible sets of substitutions.

However, their analysis also manipulates sets of atoms ("abstract goals")—these can become arbitrarily large.

(Codish *et al.* [1991] ensure termination by a second level of abstraction that bounds the no. of occurrences of a predicate symbol in an abstract goal.)

Goal-Dependence vs. Goal-Independence

- Traditionally, most analyses for Prolog rely on the operational model of SLD-resolution.
  ⇒ "Top-Down Analyses."
- Typical approach:
  - Start with a description of a top-level goal.
  - Trace the execution and record (descriptions of) reachable states.
- Results of analysis are applicable only for executions that correspond to the initial goal.
  ⇒ "Goal Dependent Analyses."

  Advantages : *Efficiency* : only reachable states considered.

  Disadvantage : *Repetition* : may need to redo the analysis for different goals.

Goal-Dependence vs. Goal-Independence : cont'd

- Alternatively, we can model the declarative behavior of the program.
  - Develop an abstraction of the *immediate consequence operator* T_P for a program P.
  - Use this to obtain a description of the least Herbrand model of the program.
    ⇒ "Bottom-Up Analyses."
- The results of this analysis hold for the whole program, i.e., for all goals.
  ⇒ "Goal Independent Analyses."

  Advantages : *Non-repetition* : don't need to redo the analysis for different goals.

  Disadvantage : *Cost* : May analyze irrelevant parts of the program.

Goal-Independent ≡ Bottom-Up?

- We can obtain goal-independent analyses using top-down analyses:
  - Use a "most general goal" $p(X_1, \ldots, X_n)$ in a top-down analysis.
  - Recover the results for specific analysis for different goals from this using unification.
- First proposed by Jacobs & Langen [1989].
- In general, this can cause a loss in precision because of the way in which "partial results" are combined during analysis.

  No loss in precision if abstract domain satisfies some algebraic properties (abstract unification is *commutative* and *additive*).
- *Efficiency implications* : Reusing goal-independent analysis results can improve the efficiency of goal-dependent analyses.

Goal-Dependent ≡ Top-Down?

- Use the "magic sets" transformation from deductive databases to propagate information about a particular top-level goal.
- First proposed by Codish *et al.* [1990], Debray & Ramakrishnan [1990], Kanamori & Kawamura [1990], and Mellish [1990].
- In general, this can cause a loss in precision because the transformation does not keep track of which call produced which answer.

  No loss in precision if the transformation is modified to link calls and answers.
- *Efficiency implications* : Allows the application of efficient fixpoint computation techniques from deductive databases.

The "Magic Sets" Transformation

Input Clause :

$p(\bar{t})$:− $q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$.

Transformed Clauses :

$p(\bar{t})$:− $called\_p(\bar{t}), q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$.

$called\_q_1(\bar{t}_1)$:− $called\_p(\bar{t})$.
. . .
$called\_q_{i+1}(\bar{t}_{i+1})$:− $called\_p(\bar{t}), q_1(\bar{t}_1), \ldots, q_i(\bar{t}_i)$.

- First proposed by Bancilhon *et al.* [1986] for efficient evaluation of deductive databases.
- In effect, the added rules specify explicitly the L-to-R execution strategy of Prolog.
- This approach can be used to specify also other possible execution strategies.
- The apparent recomputations of various goals can be avoided by extensions to the basic transformation ["supplementary magic sets."]

Adapting Magic Sets for Program Analysis

1. Make input and output instances of arguments explicit.

   Input Clause :

   $p(\bar{t})$:− $q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$.

   Transformed Input Clause :

   $p(\bar{t}_\downarrow, \bar{t}_\uparrow)$:− $q_1(\bar{t}_{1\downarrow}, \bar{t}_{1\uparrow}), \ldots, q_n(\bar{t}_{n\downarrow}, \bar{t}_{n\uparrow})$.

2. Make unifications explicit.

   Input Clause :

   $p(\bar{t})$:− $q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$.

   Transformed Input Clause :

   $p(\bar{x})$:− $unify(\bar{x}, \bar{t})$,
   $unify(\bar{y}_1, \bar{t}_1), q_1(\bar{y}_1)$,
   . . . ,
   $unify(\bar{y}_n, \bar{t}_n), q_n(\bar{y}_n)$.

3. Replace *unify* operations by *abstract_unify*.

4. For any predicate p : *is_called_p* gives calling patterns, while the abstract relation computed for p gives the corresponding success patterns.

Magic Sets Based Analysis in Practice

- Prototype system implemented by Codish & Demoen [1993].
- The whole system is written in Prolog.

  Provides good flexibility and capacity for experimentation.
- Very good speed of analysis, comparable to hand-crafted top-down analyzers.
- Recent experiments by Codish *et al.* [1994] indicate that for some abstract domains, a 2-phase analysis:

  Phase 1 : goal-independent analysis;

  Phase 2 : Recover goal-dependent information from results of Phase 1;

  may be faster and scale better than a purely goal-dependent analysis.