Abstract Interpretation of Logic Programs

Implementation and Efficiency Issues

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

Improving Efficiency of Analysis

- Exploit properties of specific abstract domains to reduce the cost of various operations:
  - use abstract domains where aliasing can be ignored [Debray 88]
  - use precomputation to reduce the cost of fixpoint computation [Jacobs & Langen 89].
- Use partial evaluation to derive specialized interpreters that are efficient [Debray 87, Hermenegildo *et al.* 92].
- Consider a generic analysis algorithm that is efficient for specific classes of programs [Le Charlier *et al.* 91].
- Use a "low level" representation to speed up the analysis [Tan & Lin 92, Krall & Berger 95].

Ignoring Aliasing Effects

[Debray 88]

- In general, handling aliasing precisely can give bad worst case complexity. However, many analyses (groundness analysis, type analysis, etc.), do not necessarily need aliasing information.
- Aliasing can be ignored if each element of the abstract domain is a set of terms that is closed under substitution.
- If aliasing can be safely ignored, then programs can be analyzed efficiently: for most programs,
  - the max. arity does not increase with program size;
  - the number of calling/success patterns for any predicate is small, typically $O(1)$.

  An analysis algorithm can be given whose worst case complexity is $O(N)$ for such programs, where N is the program size.
- *Disadvantage* : Such domains cannot express *"definite freeness"*.

Reducing Fixpoint Computation Costs I

"Condensing" [Jacobs & Langen 89]

- Focuses on program analysis to detect groundness and aliasing (intended application: parallelization of Prolog).
- Aimed at reducing the cost of iterating to a fixpoint during analysis.
- *Basic Idea* :
  - "Preanalyze" each clause in isolation to generate a fixed approximation to its meaning (*"condensing"*).

    This step may require iterating to a fixpoint.
  - Abstract interpretation of a specific call can then be done using a single "abstract unification" step.
- The condensing step does not cause a loss in precision if abstract operations satisfy certain algebraic properties:
  - abstract unification is commutative (order of unifications does not lose information); and
  - abstract unification is additive (LUB operations do not lose information).

Condensing vs. Extension Tables

- Extension Tables and Condensing are alternative algorithms for fixpoint computation w.r.t. a generic "abstract unification" procedure.
- Condensing involves a single fixpoint computation per predicate, and a number of abstract unifications.

  Extension Tables may require multiple fixpoint computations per predicate.
- <u>*Comparative Efficiency*</u> :
  1. Extension Tables more efficient if procedures are used in a small number of specific modes.
  2. Condensing more efficient if procedures are used in a large number of different modes.
  3. Both are equally efficient if procedures are always used in their most general modes.
- Experience suggests that most programs are written to use a small number of specific modes: thus, (1) seems to hold in general.

Reducing Fixpoint Computation Costs II

"Abstract Compilation" [Debray 87, Hermenegildo *et al.* 92]

To analyze a program P over abstract domain $\mathcal{D}_{abs}$:

- Make one pass over P to generate an "approximate program" P'.
- When P' is executed, the result it computes is the same as that of analyzing P over $\mathcal{D}_{abs}$.
- P' can be thought of as an abstract interpreter for $\mathcal{D}_{abs}$ that has been specialized for input P.

$$\begin{tikzcd}[row sep=huge, column sep=huge]
\underset{P}{\text{Source Program}} \arrow[r, "\text{semantics } \mu"] \arrow[d, "\text{transformation } \tau"'] & \underset{M}{\text{Source Meaning}} \arrow[d, "\text{abstraction } \alpha"] \\
\overset{P'}{\text{"Approximate" Program}} \arrow[r, "\text{semantics } \mu"] & \overset{M'}{\text{Simplified Meaning}}
\end{tikzcd}$$

Abstract Compilation: Implementation Issues

Main Issue : management of the extension table.

Alternatives:

- Pass around explicitly as a data structure.
- Use the Prolog database via Prolog's `assert` primitive.

Related Issues :

- Control management: all execution paths have to be explored.
- Memory management: dead memory should be reclaimed where possible.

How Expensive is Assert?

Cost of assert :

	Unification	Assert
Quintus 1.6	1.0	544–1477
SB-Prolog 2.3	1.0	3038–6075
Sicstus 0.5	1.0	359–678

Cost of accessing asserted code :

	Unification	Assert
Quintus 1.6	1.0	300–930
SB-Prolog 2.3	1.0	103–144
Sicstus 0.5	1.0	308–639

Abstract Compilation: Explicit Tables

Generated Code : A predicate p with n clauses generates the code:

```
p$pred(InMode, ExtTbl, OutMode) :-
   exec_clauses(n, p$cl(_,InMode,ExtTbl,_), Modes),
   lub(Modes, OMode).

   p$cl(1, InMode, ExtTbl, OutMode) :- ...
     ...
   p$cl(n, InMode, ExtTbl, OutMode) :- ...
```

Advantages : No asserts at analysis time.

Asserts are expensive, so this speeds things up.

Disadvantages :

- Control management : since extension table is a data structure, alternative execution branches have to be explored explicitly.
  ⇒ more code has to be generated.
- Memory management : memory can't be reclaimed via failure.

Abstract Compilation: Implicit Tables

Generated Code : Each clause in the input program maps to a clause in the generated program.

A source clause of the form

$$p(\bar{t}) :- q_0(\bar{t}_0), \ldots, q_n(\bar{t}_n)$$

corresponds to the generated clause

comp\$MODE($p(\bar{x})$, Asub$_{in}$, Asub$_{out}$) :−
  \$unify($p(\bar{x})$, $p(\bar{t})$, Asub$_{in}$, Asub$_0$),
  \$find_mode($q_0(\bar{t}_0)$, Asub$_0$, Asub$_1$),
    ...,
  \$find_mode($q_n(\bar{t}_n)$, ASub$_n$, Asub$_{out}$).

Advantages :

- Control management: extension table is persistent, so alternative execution paths can be explored by backtracking.
  ⇒ Simplifies the generated code.
- Memory management: backtracking also reclaims dead memory.

Disadvantages : Expensive asserts at analysis time.

"Low-Level" Algorithmic Optimizations

[Le Charlier *et al.*, 91, 92]

Basic Idea : Implement a generic dataflow analysis algorithm. Such an algorithm must specify how a number of decisions should be made, e.g.:

- when and how to test for termination;
- how to choose the "next" execution branch to examine;
- what program fragments to (re)analyze when information about part of the program changes.

Careful choices here, and also in careful design of the algorithm to avoid unnecessary work, can reduce the computation cost.

Algorithmic Optimizations 1: *Termination Testing*

- Suppose we implement a dataflow analysis for Prolog in Prolog.

  Typically the fixpoint computation will have the form:

  ```
  fixpt(Pgm, OldInfo) :-
     analyze(Pgm, OldInfo, NewInfo),
     (OldInfo \== NewInfo ->
        fixpt(Pgm, NewInfo) ;
        true
     ).
  ```

- Checking `OldInfo \== NewInfo` is expensive!

  Translation into C retains this overhead.

- The value of `OldInfo` is needed to test for termination.

  ⇒ `NewInfo` must be a different data structure [need to copy much of `OldInfo`].

- Significant space as well as time overheads.

Algorithmic Optimizations 1: *Termination Testing*

- Instead of explicit comparison, use a Boolean to indicate whether or not another iteration of analysis is necessary.
- This reduces the actual cost of checking whether a fixpoint has been reached.

```
fixpt(Pgm, OldInfo) :-
    analyze(Pgm, OldInfo, NewInfo),
    (change ->
        fixpt(Pgm, NewInfo) ;
        true
    ).
```

- Means that `OldInfo` is not used subsequently.
- If the implementation is in C, the structure `OldInfo` can be destructively updated during analysis.
- This improves both the space and time requirements of the analysis.

Algorithmic Optimizations 2: *Definitive Values*

- Dataflow analyses typically maintain a worklist of goals to be processed.

  When analyzing a clause

  $H\ :-\ B_0, \ldots, B_i, \ldots, B_n$

  if the analysis of B_i produces any new information, then $B_{i+1}, \ldots, B_n$ are re-analyzed.
- Re-analysis is not always necessary.

  B_j $(j > i)$ need not be reanalyzed if the current information for B_j is such (for the given abstract domain) that it will not change as a result of re-analysis.
- Benefits of saving redundant work balanced by the cost of maintaining and checking dependency information.

  Careful attention to representation of abstract domain allows the cost of manipulating such dependencies to be kept small.
- Le Charlier & Van Hentenryck [1991] report that by maintaining this information, almost all unnecessary work can be eliminated.

Algorithmic Optimizations 3: *Clausal Prefixes*

- When analyzing a clause

  $H\ :-\ B_0, \ldots, B_i, \ldots, B_j, \ldots, B_n$

  B_j need not be reanalyzed if the abstract substitution β_i from B_i does not affect B_j.
- Suppose B_j is defined by

  $B_j\ :-\ A_0, \ldots, A_i, \ldots, A_m.$

  Suppose the abstract substitution β_i does not affect $A_0, \ldots, A_{i-1}$.

  When analyzing B_j with β_i, we can start at A_i.
- Each body literal B and abstract substitution β in a program defines a set of literals that *depend on* them, i.e., need to be reanalyzed.

  These dependencies can be constructed incrementally during analysis.
- Engelbert *et al* [SP&E 1993] : Maitaining clausal dependencies yields a ~60% speed improvement w.r.t. a straightforward implementation, ~30% improvement on a more optimized implementation.

Algorithmic Optimizations 4: *Operation Caching*

- Sophisticated analyses often involve abstract domains with nontrivial structure, e.g., type graphs, sharing sets, propositional formulae.
- For such domains, abstract operations, e.g., abstract unification, LUB, can be quite expensive.
- Efficiency can be improved considerably by caching the results of abstract operations.

  Cache lookup involves hashing; insertion involves hashing and copying. For sophisticated abstract domains these operations can be expensive.
- Performance improvements [Engelbert *et al*, SP&E 1993; Van Hentenryck *et al.*, JLP 1995] :
  - Without clausal dependencies, caching produces ~60% speed improvement w.r.t. a straightforward implementation, ~30% on an optimized implementation.
  - Caching adds very little additional speed improvement once clausal dependencies are maintained.

Analysis of Low Level Representation

- Usual approach to analysis [Le Charlier, Van Hentenryck, *et al.*; Van Roy; Taylor; ...] :

  Interpret the program (in abstract syntax tree form) over the abstract domain.

- Abstract compilation [Debray 87; Hermenegildo *et al.* 92] :

  Compile the input program to obtain a Prolog program which, when executed, will produce the analysis results.

- This idea can be extended logically to use a low level representation [Tan & Lin 92, Krall & Berger 95]:

  Compile the input program to obtain a sequence of WAM instructions (in byte-code form) that can be executed (on a special "abstract WAM engine.") to produce the analysis results.

- Each step in this progression reduces the cost of various abstract operations.

Analysis of Low Level Representation: *Example*

- *Unification* : Re-interpret the unification instructions of the WAM. E.g.:

  `get_list` $A_i \equiv$
  begin
  deref(A_i);
  if $A_i \simeq$ **'.'**/2 **then**
  ... abstract unification ...
  else *fail*
  end

  Control Flow :

  - The `call` p instruction of the WAM is re-interpreted to consult the extension table:

    `call` $p \equiv$
    begin
    get arguments
    if $already\_explored(p(args))$ **then**
    ... retrieve from extension table...
    else
    ... transfer control to code for p
    end

  - The `proceed` and `execute` instructions are re-interpreted to explore the other clauses for the predicate via failure.

Analysis of Low Level Representation: *Advantages*

- "Impedance mismatch" between high level representation (syntax tree) and low level information (registers etc.) is avoided for low level code optimizations.

- Code optimizations that change the program can be accommodated easily.

- Different abstract domains can be used for different analyses by changing the "abstract WAM engine."

  The Prolog-to-WAM compiler is unaffected.

- Speed: Tan & Lin report speedups of 57–575 (average speedup: 152) compared to Van Roy's Aquarius system.

Improving Precision of Analysis: Reexecution

- Use a more sophisticated ($\sim$ larger) abstract domain:
  - discriminate better among different sets of terms;
  - track aliasing more precisely.

- *Reexecution* [Bruynooghe *et al.* 87, Le Charlier & Van Hentenryck 91] :
  - use a simple abstract domain;
  - re-execute parts of the program to propagate aliasing effects.

- *Empirical Evaluation of Reexecution* :
  - On a simple domain, reexecution improves the information inferred for over 50% of the argument positions, with an average slowdown of 8%.
  - On an elaborate domain, reexecution produces smaller improvements (8% of arg. positions) and is more expensive (2.3 times slower).
  - *Reexecution on a simple domain attains almost identical precision as no reexecution on an elaborate domain, but is more than twice as fast.*

Improving Precision: Combining Domains

- Suppose we have two analyses with abstract domains $\langle \mathcal{D}_1, \sqsubseteq_1, abs_1, conc_1 \rangle$ and $\langle \mathcal{D}_2, \sqsubseteq_2, abs_2, conc_2 \rangle$.

  The obvious way to combine these analyses is to do them "in parallel", i.e.:

  $$\langle (\mathcal{D}_1, \mathcal{D}_2), (\sqsubseteq_1, \sqsubseteq_2), (abs_1, abs_2), (conc_1, conc_2) \rangle.$$

- If we allow the analyses to share information with each other, we can get better precision.

- *Example* : Suppose we have two abstract domains: (1) groundness; and (2) pairwise sharing.

  Consider the unification

  `h(U, U, g(U, V)) = h(X, Y, Z)`

  where `X` is ground, and there is no sharing.

  – If groundness information is not taken into account, the sharing analysis infers that `X`, `Y` and `Z` can all share a common variable.

  – Taking the groundness information into account, we can infer that `X`, `Y` and `Z` are all ground and so do not share.

Improving Precision: Combining Domains

Combining analyses by combining their domains and allowing them to interact has several benefits [Codish *et al.*, TOPLAS 1995] :

– Only one analysis pass over the program is necessary, instead of multiple passes.

– The abstract operations for the individual analyses do not have to be redefined.

  Instead, they are combined and allowed to use information from the other analyses.

– The precision of the analyses can be improved by their interaction.

– In some cases, the improved precision can cause the analysis to reach a fixpoint in a smaller no. of iterations.

  This can result in improved speed.

Experimental Results 1

Warren, Hermenegildo, & Debray, 1988 :

- Tested two different flow analysis systems on a variety of medium-sized programs.

- System characteristics:

  – Both implemented in Prolog;

  – Both used program transformation to partially evaluate the abstract interpreter ("*abstract compilation*");

  – Both used extension tables.

  However, the extension tables were implemented in different ways.

- *Broad Conclusions* :

  – *cost* : analysis time/total compile time $\simeq$ 0.3–0.8.

  – *precision* : %hits $\simeq$ 35%–100% (one exception, 11.5%).

Experimental Results 2

Van Roy, 1990 :

- Describes the dataflow analysis component of the Aquarius compiler.

- System characteristics:

  – implemented in Prolog;

  – no partial evaluation of the abstract interpreter;

  – the abstract domain used provides information for low-level optimizations;

  – uses heuristics to balance the tradeoff between precision and efficiency.

- *Broad Conclusions* :

  – type information derived for 56% of argument positions;

  – analysis time: roughly linear in the no. of argument positions.

Experimental Results 3

Le Charlier & Van Hentenryck, 1991 :

- Tested a generic flow analysis systems on a variety of medium-sized programs.
- System characteristics:
  - implemented in Pascal;
  - no partial evaluation of the abstract interpreter;
  - abstract domain for experiments: an 8-element domain for groundness analysis.
- *Broad Conclusions* :
  - *cost (no. of iterations)* : typically varies linearly with program size.
    (average: $3N$; worst: $7.5N$)
  - *cost (execution time)* : typically varies linearly with program size.
    (average: $0.6N$)
  - *precision* : %hits $\simeq$ 76%–100%;

Experimental Results 4

Le Charlier & Van Hentenryck, 1992 :

- Uses a generic abstract interpretation algorithm to compare three abstract domains for groundness analysis.
  - *Mode* : a simple domain of modes;
  - *Prop* : a domain of propositional formulae [Marriott & Søndergaard 89, 92];
  - *Pattern* : a domain that keeps track of function symbols.
- *Precision* :
  - *Prop* and *Pattern* are both quite precise: on many programs, they infer over 80% of ground arguments.
    Mode is somewhat less precise than *Prop*.
    Mode+Reexecution attains the same precision as *Prop*.
  - *Prop* only slightly less precise than *Pattern*.
- *Efficiency* :
  - *Prop* about twice as slow as *Mode*, about 40% slower than *Mode*+Reexecution.
  - *Prop* about 40% faster than *Pattern*.

Experimental Results: Absolute Times

[Van Hentenryck *et al.*, JLP 1995] (Sun Sparcstation 10/30, secs) :

Program	*Prop*	*Types*	*Mode*
CS	1.34	2.00	1.29
Disj	1.01	1.12	0.74
Gabriel	0.47	0.69	0.31
Kalah	0.93	1.86	0.72
Peep	1.16	2.14	1.11
PG	0.16	0.27	0.16
Plan	0.12	0.20	0.11
Press1	5.96	8.80	1.51
Press2	6.03	2.77	1.55
Qsort	0.05	0.06	0.08
Queens	0.04	0.05	0.06
Read	1.66	5.29	1.39

Summary

- Careful implementation of dataflow analyzers can produce systems with good speed.
- A variety of techniques available for speeding up dataflow analysis of logic programs:
  - Choice of appropriate domains.
  - Partial evaluation.
  - Low level algorithmic optimizations.
  - Low level abstract interpretation via re-interpreted WAM instructions.
- Precision can be improved in a number of ways:
  - Using more elaborate abstract domains.
  - Using simple domains with re-execution.
  - By combining domains.
- Techniques such a re-execution and domain combination retain the simplicity advantages of simple domains while yielding potentially more precise analyses.
  The improved precision can lead to an improvement in speed.