Abstract Interpretation of Logic Programs

Complexity Issues

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

Introduction

- Compile-time analyses attempt to get finitely computable approximations to undecidable properties.

  ⇒ the more work we put in, the better the approximation.

- This suggests that the complexity of analyses will increase with the precision.

- Existing analyses have complexities ranging from linear time to exponential time.

  Question : Is high complexity due to sloppy design or fundamental difficulty?

- *Question* : Is it *intrinsically* expensive to carry out some analyses sufficiently precisely?

- *Question* : What does it mean to carry out an analysis "sufficiently precisely"?

Characterizing "Precision" of Analysis

- *relative precision* : precision of one analysis relative to another.

  Unsuitable for our needs.

- *exactness set* : the class of programs for which an analysis gives exact results.

  The precision of an analysis algorithm is given by its exactness set.

Proposition : If A is relatively more precise than B, then $exact(A) \supseteq exact(B)$.

Examining the Complexity of an Analysis

- Carefully identify all the different situations in which an analysis sacrifices precision.

  Example : A groundness analysis of logic programs may not check whether a clause fails at runtime.

- For each situation i in which precision is sacrificed, identify the class of programs P_i that do not cause any loss in precision for that reason.

  Example : For groundness analyses that ignore failure: consider programs that never fail at runtime.

- Consider $P = \bigcap P_i$.

  The analysis *must* compute exact results for every program in P.

- Determine the complexity of deciding programs in the class P.

Some Specific Analyses Considered

1. Analysis of normal programs (i.e., logic programs with negation) [Marriott & Søndergaard NACLP 89, JLP 92].

2. Type inference using type graphs [Van Hentenryck *et al.* PLDI 94].

3. Type Analysis using depth-k abstraction [Codish *et al.* TOPLAS 94, King ICLP 94, Santos Costa *et al.* ICLP 91, Sato & Tamaki TCS 84, ...].

4. Groundness and Sharing analysis using the domain *Prop* [Marriott & Søndergaard NACLP 89, Marriott Søndergaard & Jones TOPLAS 94, Van Hentenryck *et al.* JICSLP 92].

Some Simple Classes of Programs

- *recursion-free* : programs with no recursion.

- *function-free* : programs containing no function or constant symbols.

- *bounded arity* : programs where every predicate and function symbol has arity $O(1)$.

- *alias-free* : programs where no two variables are aliased during execution.

- *failure-free* : programs that do not give rise to any failed execution branches.

Analysis of Programs Containing Negation

Definition : A Schönfinkel-Bernays formula is of the form

$$\exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n F$$

where F is quantifier free and does not contain any function symbols.

Fact [Lewis 1980] : Determining whether an arbitrary Schönfinkel-Bernays formula is satisfiable is NEXPTIME-complete.

If a formula $\exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n F$ is satisfiable, it has a model of size at most m.

Theorem : Precise analysis of recursion-free logic programs containing negation is NEXPTIME-hard.

Addresses the complexity of an analysis of Marriott & Søndergaard [NACLP 89, JLP 92].

Analysis of Programs Containing Negation 2

Intuition : Given a formula $\varphi \equiv \exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n F$, consider models of size m.

General Approach : Generate a Prolog program P_φ such that by analyzing P_φ we can determine whether φ is satisfiable.

P_φ behaves as follows:

- It uses unification to bind the x_i (to one of m constants);
- it uses negation by failure to consider all possible bindings for the y_i (from among m constants).

```
main :- bind(X1),...,bind(Xm), \+ q(X1,...,Xm,f).
```

`main/0` has non-empty type iff φ satisfiable.

Technical Issue : Suppose there are atoms $p(\bar{u})$ and $p(\bar{v})$ in F.

If we generate the same ground instance $p(\bar{t})$ for both these atoms, we must ensure that it gets consistent truth values in the model.

⇒ Need some kind of symbol table.

Analysis of Programs Containing Negation 3

Symbol Table Issues : Can't use recursive data structures, e.g., lists (exact analysis becomes undecidable).

The code for constructing/searching the symbol table must be generated in polynomial time.

Idea : Each n-ary predicate p has its own symbol table, traversed by a group of $n+1$ predicates `lookup_j` that handles argument j of the predicate:

```
lookup_j(Atom, Table, Tval) :-
   Atom = p(A_1, ..., A_j, ..., A_n),
   A_j = a_1,
   Table = table(Tab_1, _2, ..., _m),
   lookup_j+1(Atom, Tab_1, Tval).
     ...
lookup_j(Atom, Table, Tval) :-
   Atom = p(A_1, ..., A_j, ..., A_n),
   A_j = a_k,
   Table = table(_1, _2, ..., Tab_m),
   lookup_j+1(Atom, Tab_m, Tval).
```

The resulting program is recursion-free.

Consequence : Precise analysis of recursion-free programs containing negation is NEXPTIME-hard.

Analysis of Programs Containing Negation 4

Suppose $\varphi \equiv \exists x_1 x_2 \forall y_1 [(p(x_1, x_2) \wedge q(x_2, y_1)) \vee \neg p(y_1, x_2)]$.

Then P_φ is (in part) :

```
main :- bind(X1), bind(X2), \+ q(X1, X2, f).

q(X1, X2, T) :- bind(Y1),
    p(X1, X2, A1), q(X2, Y1, A2), and(A1, A2, A3),
    p(Y1, X2, B1), neg(B1, B2),
    or(B1, B2, T),
    lookup_p(p(X1, X2), SymTab_p, A1),
    lookup_q(q(X2, Y1), SymTab_q, A2),
    lookup_p(p(Y1, X2), SymTab_p, B1).

bind(a1).
bind(a2).

p(_, _, t).
p(_, _, f).

q(_, _, t).
p(_, _, f).
```


Analysis of Recursion-free Horn Programs

Definition : A Quantified Boolean Formula is of the form

$$Q_1 x_1 \cdots Q_n x_n B \qquad Q_i \in \{\exists, \forall\}$$

where B is a Boolean formula over $\{x_1, \ldots, x_n\}$.

Fact [Stockmeyer & Meyer 1973] : The problem of determining whether an arbitrary QBF is true is PSPACE-complete.

Theorem : Type analysis of logic programs (i.e., keep track of function symbols) that give up precision only when recursion is encountered is PSPACE-hard.

It is PSPACE-hard even if every function and predicate symbol has arity at most 2, and each clause has at most two literals in its body.

Addresses the complexity of ananalysis of Van Hentenryck *et al.* [PLDI 94].

Analysis of Recursion-free Horn Programs 2

General Approach : Given a QBF φ, generate a Prolog program whose analysis tells us whether or not φ is true.

Main Idea : Pass truth values for variables as a list of $\{$`t`, `f`$\}$, and at the end evaluate a ground Boolean formula.

Technical Issue : Need to evaluate an arbitrary Boolean formula without using recursion.

Solution : Use "partial evaluation."

Analysis of Recursion-free Horn Programs 3

Evaluating Boolean Formulae *(Naive approach)* :

```
eval(and(F1, F2), t) :-
    eval(F1, t), eval(F2, t).
eval(and(F1, F2), f) :- eval(F1, f).
eval(and(F1, F2), f) :- eval(F2, f).
```

etc.

This doesn't work: recursion + binary function symbol is undecidable.

Partial Evaluation : Generate a family of predicates $\mathtt{eval}_i$ that handles formulae with at most i connectives:

```
eval_i(and(F1, F2), t) :-
    eval_{i-1}(F1, t), eval_{i-1}(F2, t).
eval_i(and(F1, F2), f) :- eval_{i-1}(F1, f).
eval_i(and(F1, F2), f) :- eval_{i-1}(F2, f).
```

etc.

This yields a non-recursive program.

This works because any given QBF will have a fixed depth n.

Analysis of Recursion-free Horn Programs 4

Suppose $\varphi = \forall x \exists y \forall z [x \wedge \neg(y \vee z)]$.

Then P_φ is (in part) :

```
main :- p1([], t).

p1(V1, t) :- p2([t|V1], t), p2([f|V1], t).
p1(V1, f) :- p2([t|V1], f).
p1(V1, f) :- p2([f|V1], f).

p2(V2, t) :- p3([t|V2], t).
p2(V2, t) :- p3([f|V2], t).
p2(V2, f) :- p3([t|V2], f), p3([f|V2], f).

p3(V3, t) :- p4([t|V3], t), p4([f|V3], t).
p3(V3, f) :- p4([t|V3], f).
p3(V3, f) :- p4([f|V3], f).

p4([Z, Y, X], Tval) :-
    eval_3(and(X, neg(or(Y, Z))), Tval).
```


Depth Abstraction

- In the presence of function symbols with arity > 0, the Herbrand base becomes infinite.

  Example : with the function symbols **0**/0, **s**/1, we can generate the infinite set of terms

  $$\mathbf{N} = \{\mathbf{0}, \mathrm{s}(\mathbf{0}), \mathrm{s}(\mathrm{s}(\mathbf{0})), \mathrm{s}(\mathrm{s}(\mathrm{s}(\mathbf{0}))), \ldots\}$$

  Because of this, dataflow analyses may fail to terminate.

- *Depth Abstraction* : for a given $k \in \mathbf{Z}^+$,
  - "cut off" every term at depth k;
  - replace each such "cut off" subterm with a new variable;
  - any such depth-abstracted term denotes the set of all its ground instances.

  Example : Under depth-2 abstraction, the infinite set **N** above is abstracted to

  $$\{\mathbf{0}, \mathrm{s}(\mathbf{0}), \mathrm{s}(\mathbf{X})\}.$$

- For any $k > 0$, the set of k-depth abstracted terms is finite for any finite program.

  This allows us to obtain terminating analyses.

Analysis using Depth Abstraction

Definition : A recursive monotone Boolean function (RMBF) is an equation $f(x_1, \ldots, x_n) = E$, where E has the form

$$E ::= \mathbf{t} \mid \mathbf{f} \mid x_i \mid E_1 \wedge E_2 \mid E_1 \vee E_2 \mid f(E_1, \ldots, E_n).$$

Fact [Hudak & Young 1986] : Given a RMBF F and a tuple of n arguments $A = \langle a_1, \ldots, a_n \rangle$, the problem of determining whether $f(a_1, \ldots, a_n) = \mathbf{t}$ in the least fixpoint of F (with $\mathbf{f} \sqsubseteq \mathbf{t}$) is EXPTIME-complete.

Theorem : Type analysis using depth abstraction is EXPTIME-complete.

It remains EXPTIME-complete even if there is no aliasing and we use only depth-1 abstraction.

This addresses the complexity of analyses by Codish *et al.* [TOPLAS 94], King [ICLP 94], Santos Costa *et al.* [ICLP 91], Sato & Tamaki [TCS 84], etc.

Analysis using Depth Abstraction 2

General Approach : Given a RMBF F and arguments A, generate a Prolog program $P_{F,A}$ whose type analysis using depth-1 abstraction tells us whether or not $F(A) = \mathbf{t}$.

Example : Let F be

$g(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge g(g(x_3 \vee x_2, x_3, x_1), x_3, x_1 \wedge x_2))$

and $A = \langle \mathbf{t}, \mathbf{f}, \mathbf{f} \rangle$.

Then $P_{F,A}$ is given (in part) by:

```
g(X1, X2, X3, Y) :-
    or(X3, X2, U1),
    g(U1, X3, X1, U2), and(X1, X2, U3),
    g(U2, X3, U3, U4),
    and(X2, U4, U5), or(X1, U5, Y).

main :-
    X1 = t, X2 = f, X3 = f,
    g(X1, X2, X3, Y),
    Y = t.
```

`main/0` has nonempty type iff $F(A) = \mathbf{t}$.

Analysis using Depth Abstraction 3

- What happens if we give up precision on recursive calls (similar to the widening of type graphs by Van Hentenryck *et al.* [PLDI 94])?
- In this case:
  - Loses some precision.
  - Complexity drops to PSPACE-hard.
- The proof is by reduction from QBF.

  In the program generated for this, the arities of predicates grows with the no. of quantifiers in the input QBF.

  If we assume that the maximum arity of any predicate is bounded, then the dataflow analysis using depth-abstraction is NP-complete.

Analysis using Depth Abstraction 4

Example : Let $\varphi = \forall x \exists y \forall z [x \wedge \neg(y \vee z)]$.

Then the Prolog program generated is:

```
main :- p1(t).

p1(t) :- p2(t,t), p2(f,t).
p1(f) :- p2(t,f).
p1(f) :- p2(f,f).

p2(V1,t) :- p3(V1,t,t).
p2(V1,t) :- p3(V1,f,t).
p2(V1,f) :-
    p3(V1,t,f), p3(V1,f,f).

p3(V1,V2,t) :-
    p4(V1,V2,t,t), p4(V1,V2,f,t).
p3(V1,V2,f) :- p4(V1,V2,t,f).
p3(V1,V2,f) :- p4(V1,V2,f,f).

p4(X,Y,Z,A) :-
    or(Y,Z,U1), neg(U1,U2), and(X,U2,A).
```


Groundness and Aliasing Analysis

The Abstract Domain *Prop* [Marriott & Søndergaard 89] : Defined by formulas of the form

$$E ::= x \mid E \wedge E \mid E \vee E \mid E \leftrightarrow E.$$

Intuition : **true** denotes "ground."

$E_1 \leftrightarrow E_2$ denotes sharing between E_1 and E_2.

Example : Consider the program clause

```
app(X1,X2,X3) :-
    X1 = [H|L1], X3 = [H|L3], app(L1,X2,L3).
```

Abstraction using *Prop* gives

$$app(X_1, X_2, X_3) :- \\ X_1 \leftrightarrow (H \wedge L_1) \wedge X_3 \leftrightarrow (H \wedge L_3) \wedge \\ app(L_1, X_2, L_3).$$

Theorem : Groundness and sharing analysis using *Prop* is EXPTIME-complete.

This addresses the complexity of an analysis of Marriott, Søndergaard and Jones [TOPLAS 94].

Groundness and Aliasing Analysis 2

General Approach : Given a RMBF F and arguments A, generate a Prolog program $P_{F,A}$ whose aliasing analysis using *Prop* tells us whether $F(A) = \mathbf{t}$.

Technical Issues : Representing truth values.

- Aliasing/Groundness analyses typically assume that everything in a program is executed.
  Execution failure is not taken into account.
- The generated program $P_{F,A}$ cannot have any failed execution branches.
- This means that we cannot use distinct constants to represent **true**, **false**.

Groundness and Aliasing Analysis 3

Representing Truth Values : Use a pair of Prolog variables X_L, X_R to represent a *Prop* variable x.

X_L, X_R aliased $\sim$ **t**
X_L, X_R independent $\sim$ **f**

Conjunction :

```
and(Xl, Xr, Yl, Yr, Xl, Yr) :- Xr = Yl.
```

Disjunction :

```
or(Xl, Xr, _, _, Xl, Xr).
or(_, _, Yl, Yr, Yl, Yr).
```

Note : For the complexity result, we don't need to deal with the $\leftrightarrow$ connective of *Prop*.

Groundness and Aliasing Analysis 4

Let F be

$f(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge f(f(x_3 \vee x_2, x_3, x_1), x_3, x_1 \wedge x_2)$

and $A = \langle \mathbf{t}, \mathbf{f}, \mathbf{f} \rangle$.

Then $P_{F,A}$ is given (in part) by :

```
f(X1l, X1r, X2l, X2r, X3l, X3r, Vl, Vr) :-
    or(X3l, X3r, X2l, X2r, U1l, U1r),
    f(U1l, U1r, X3l, X3r, X1l, X1r, U2l, U2r),
    and(X1l, X1r, X2l, X2r, U3l, U3r),
    f(U2l, U2r, X3l, X3r, U3l, U3r, U4l, U4r),
    and(X2l, X2r, U4l, U4r, U5l, U5r),
    or(X1l, X1r, U5l, U5r, Vl, Vr).

p(Ul, Ur) :-
    X = Y, f(X, Y, _, _, _, _, Ul, Ur).
```

Aliasing analysis of $P_{F,A}$ infers `Ul` $\leftrightarrow$ `Ur`, i.e., `Ul` and `Ur` are aliased, if and only if $F(A) = \mathbf{t}$.

Separating out Sources of Complexity

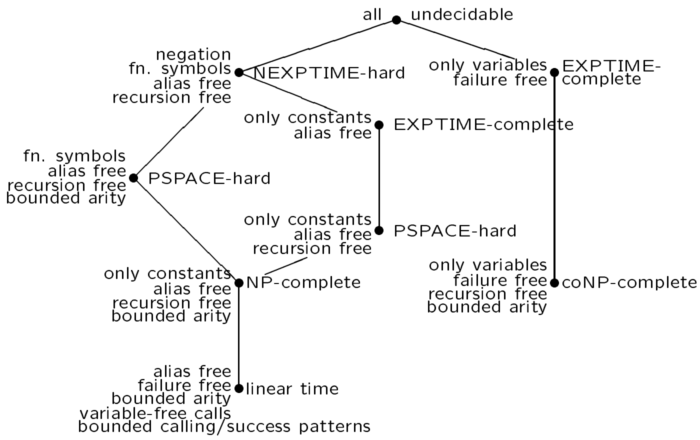
- Using the approach of *exactness sets*, we can separate out differenty sources of complexity.
- High-level structure of most analysis algorithms:
  (1) Analyze the program.
  (2) If (fixpoint not reached) goto (1).
- Thus, there are two independent sources of complexity in an analysis:
  1. the processing of unifications etc. on each iteration over a program; and
  2. the check to see whether a fixpoint has been reached.
- Multiple iterations over a program are necessary only in the presence of recursion.
  $\Rightarrow$ For non-recursive programs, the cost to check for fixpoints can be omitted.
- For groundness/aliasing analysis using *Prop* :
  - Analysis of non-recursive programs is co-NP-complete.
  - Checking whether a fixpoint has been reached is co-NP-hard.

Polynomial Time Analyses

Polynomial-time analyses can be obtained provided the exactness set is limited to programs where each predicate has at most one calling and one success pattern; and where

- aliasing/sharing is transitive; or
- calls are variable-free.

Complexity vs. Precision: Correlation



What Does All This Mean?

- The complexity results are discouraging:

  In order to get reasonable precision, for many analyses we have a worst case complexity that is exponential in the size of the program.

- Experimental results seem to indicate that in practice, precise analyses are not very expensive.

- *Reconciliation* :
  - The cases that give rise to the theoretical worst case do not arise in practice.
  - Need to define *"in practice"* more precisely.

    It isn't enough to simply present a collection of benchmarks.
  - Provide precise syntactic characterization of classes of programs.
  - *Example* : Marriott & Søndergaard [LOPLAS 94] show that groundness and aliasing analysis using *Prop* is polynomial-time if we assume that the no. of variables in any clause is $O(1)$.

Summary

- Analysis of logic programs appears harder than analysis of imperative programs (cf. Myers 1981, Jones & Muchnick 1981).

- Precise analysis of even simple programs appears to be exponential (in the program size) in the worst case.

- The average-case behavior of analysis algorithms may be better.

- Analyze the complexity of other languages.

  E.g.: for an analysis of $CLP(\mathcal{R})$ programs due to Jørgensen, Marriott, and Michaylov [1991]:
  - If the analysis is precise enough to handle quadratic equations, then it is NP-hard.