A Backward Analysis for Constraint Logic Programs[*]

Andy King
University of Kent, UK

Lunjin Lu
Oakland University, USA

[*]To appear in Theory and Practice of Logic Programming, 200x.

Context and motivation

- One recurring problem in program development is that of understanding how to re-use code developed by a third party.
- In the context of (constraint) logic programming, part of this problem reduces to figuring out how to query a program.
- If the logic program does not come with any documentation, then the programmer is forced to either experiment with queries in an *ad hoc* fashion
- *or* trace the control-flow of the program (backward) to infer ground (input) argument combinations which avoid instantiation errors.

Example toy program

```
qs([], s, s).
qs([m|xs], s, t) :-
    pt(xs,m,l,h),
    qs(l, s, [m|r]),
    qs(h, r, t).

pt([], _, [], []).
pt([x|xs], m, [x|l], h) :-
    m =< x,
    pt(xs,m,l,h).
pt([x|xs], m, l, [x|h]) :-
    m > x,
    pt(xs,m,l,h).
```

```
| ?- pt([2,1,3], M, L1, L2).
!Instantiation error in argument 1 of =< /2
!goal:  _76=<2

| ?- pt([2,1,3], 2, L1, L2).
L1 = [2,3], L2 = [1] ?  ;
no

| ?- qs([2,1,3], L1, L2).
L1 = [3,2,1|L2] ?

| ?- qs(L1, L2, L3).
L1 = [], L3 = L2 ?  ;
L1 = [_A], L2 = [_A|L3] ?  ;
!Instantiation error in argument 1 of =< /2
!goal:  _76=<_77
```

Worked example

This talk *informally* presents an abstract interpretation scheme.

The abstract interpretation scheme infers how to query a given predicate so as to avoid run-time instantiation errors.

The analysis deduces grounding properties of the call that, if satisfied, guarantee that resulting derivations cannot encounter an instantiation error.

Basic ingredients of the analysis

The Quicksort program is the first ingredient of the analysis: the input program.

The second ingredient is an abstract domain: Pos.

Pos is the domain of positive Boolean functions, that is, the set of functions $f : \{0, 1\}^n \to \{0, 1\}$ such that $f(1, \ldots, 1) = 1$.

Hence $x \vee y \in Pos$ since $1 \vee 1 = 1$ but $\neg y \notin Pos$ since $\neg 1 = 0$.

Basic domain ingredients: concretisation

A concretisation map $\gamma : Pos \to \wp(Sub)$ defines the *set* of substitutions represented by a positive function f.

Concretisation is defined in terms of $assign : Sub \to Pos$ where $assign(\theta) = \wedge_{i=1}^{n}(x_i \Leftrightarrow var(t_i))$ and $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

Then $\gamma(f) = \{\theta \in Sub \mid assign(\theta) \models f\}$.

For example, $\gamma(1) = Sub$, $\gamma(0) = \emptyset$ and $\theta, \pi \in \gamma(y \Leftarrow x)$ where $\theta = \{x \mapsto f(y, z)\}$ and $\pi = \{y \mapsto x\}$.

In this worked example, positive functions express *grounding dependencies*.

Basic domain ingredients: conjunction and disjunction

Pos comes equipped with the logical operations: conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$ (and thus bi-implication $\Leftrightarrow$).

Conjunction is used to conjoin information from different body atoms.

Disjunction is used to combine information from different clauses.

Basic domain ingredients: elimination

Conjunction and disjunction, in turn, enable two projection operators to be defined: $\exists_x(f) = f[x \mapsto 0] \vee f[x \mapsto 1]$ and $\forall_x(f) = f'$ if $f' \in Pos$ otherwise $\forall_x(f) = 0$ where $f' = f[x \mapsto 0] \wedge f[x \mapsto 1]$.

Note that $(x \Leftarrow y)[x \mapsto 0] \wedge (x \Leftarrow y)[x \mapsto 1] = \neg y$.

These elimination operators are dual in that $\forall_x(f) \models f \models \exists_x(f)$.

Normalisation and abstraction

To simplify analysis, the program is put into a form in which the arguments of head and body atoms are distinct variables.

This *normalised* program is then abstracted by replacing each Herbrand constraint $x = f(x_1, \ldots, x_n)$ with a formula $x \Leftrightarrow \wedge_{i=1}^{n} x_i$ that describes its grounding dependency.

The $\diamond$ symbol separates an assertion (the required dependency) from another Pos formula describing the grounding behaviour of unification/builtin (the success dependency).

Normalisation and abstraction of Quicksort

$$
\begin{array}{l}
\texttt{qs}(t_1, s, t_2) \texttt{ :-} \\
\quad t_1 \texttt{ = [], } t_2 \texttt{ = } s\texttt{.} \\
\texttt{qs}(t_1, s, t) \texttt{ :-} \\
\quad t_1 \texttt{ = } [m|xs]\texttt{,} \\
\quad t_3 \texttt{ = } [m|r]\texttt{,} \\
\quad \texttt{pt}(xs, m, l, h)\texttt{,} \\
\quad \texttt{qs}(l, s, t_3)\texttt{,} \\
\quad \texttt{qs}(h, r, t)\texttt{.} \\
\\
\texttt{pt}(t_1, \_, t_2, t_3) \texttt{ :-} \\
\quad t_1 \texttt{ = [], } t_2 \texttt{ = [], } t_3 \texttt{ = [].} \\
\texttt{pt}(t_1, m, t_2, h) \texttt{ :-} \\
\quad t_1 \texttt{ = } [x|xs]\texttt{, } t_2 \texttt{ = } [x|l]\texttt{,} \\
\quad m \texttt{ =< } x \\
\quad \texttt{pt}(xs, m, l, h)\texttt{.} \\
\texttt{pt}(t_1, m, l, t_2) \texttt{ :-} \\
\quad t_1 \texttt{ = } [x|xs]\texttt{, } t_2 \texttt{ = } [x|h]\texttt{,} \\
\quad m \texttt{ > } x\texttt{,} \\
\quad \texttt{pt}(xs, m, l, h)\texttt{.}
\end{array}
$$

$$
\begin{array}{l}
\texttt{qs}(t_1, s, t_2) \texttt{ :-} \\
\quad 1 \diamond g_1\texttt{.} \\
\texttt{qs}(t_1, s, t) \texttt{ :-} \\
\quad 1 \diamond g_2\texttt{,} \\
\quad \texttt{pt}(xs, m, l, h)\texttt{,} \\
\quad \texttt{qs}(l, s, t_3)\texttt{,} \\
\quad \texttt{qs}(h, r, t)\texttt{.} \\
\\
\texttt{pt}(t_1, \_, t_2, t_3) \texttt{ :-} \\
\quad 1 \diamond g_3\texttt{.} \\
\texttt{pt}(t_1, m, t_2, h) \texttt{ :-} \\
\quad 1 \diamond g_4\texttt{, =<'}(m\texttt{, } x)\texttt{,} \\
\quad \texttt{pt}(xs, m, l, h)\texttt{.} \\
\texttt{pt}(t_1, m, l, t_2) \texttt{ :-} \\
\quad 1 \diamond g_5\texttt{, >'}(m\texttt{, } x)\texttt{,} \\
\quad \texttt{pt}(xs, m, l, h)\texttt{.} \\
\\
\texttt{=<'}(m\texttt{, } x) \texttt{ :- } g_6 \diamond g_6\texttt{.} \\
\texttt{>'}(m\texttt{, } x) \texttt{ :- } g_6 \diamond g_6\texttt{.}
\end{array}
$$

$$
\begin{aligned}
g_1 &= t_1 \wedge (t_2 \Leftrightarrow s) \\
g_2 &= t_1 \Leftrightarrow (m \wedge xs) \wedge t_3 \Leftrightarrow (m \wedge r) \\
g_3 &= t_1 \wedge t_2 \wedge t_3 \\
g_4 &= t_1 \Leftrightarrow (x \wedge xs) \wedge t_2 \Leftrightarrow (x \wedge l) \\
g_5 &= t_1 \Leftrightarrow (x \wedge xs) \wedge t_2 \Leftrightarrow (x \wedge h) \\
g_6 &= m \wedge x
\end{aligned}
$$

Analysis in two steps

The first step is a bottom-up analysis for success patterns.

This step is a least fixpoint (lfp) calculation.

The second step is a bottom-up analysis for call patterns (the objective of the analysis).

This step is a greatest fixpoint (gfp) computation.

Success patterns in the lfp

A success pattern is a pair consisting of an atom with distinct variables for arguments paired with a Pos formula.

A success pattern $\langle p(\vec{x}), f\rangle$ states that if a substitution θ is generated whilst solving the query $p(\vec{x})$, then $\theta \in \gamma(f)$.

For example $\langle \mathtt{qs}(x_1, x_2, x_2), x_2 \Leftrightarrow (x_1 \wedge x_3)\rangle$ (correctly) states that if $\mathtt{qs}(x_1, x_2, x_2)$ succeeds thereby generating θ, then $assign(\theta) \models x_2 \Leftrightarrow (x_1 \wedge x_3)$, that is, θ grounds x_2 iff θ grounds x_1 and x_3.

Upper approximation of success patterns

If the success pattern $\langle p(\vec{x}), f \rangle$ correctly describes the behaviour of $p(\vec{x})$ and $f \models g$, then the success pattern $\langle p(\vec{x}), g \rangle$ also correctly describes $p(\vec{x})$.

Suppose θ is generated whilst solving the query $p(\vec{x})$.

Since $\theta \in \gamma(f)$, then $assign(\theta) \models f \models g$, hence $\theta \in \gamma(g)$.

Thus the success pattern $\langle p(\vec{x}), g \rangle$ also correctly describes $p(\vec{x})$.

Success patterns can thus be approximated from *above* without compromising correctness.

Iteration for success patterns

Iteration commences with $F_0 = \emptyset$. F_{j+1} is computed from F_j by considering each clause $p(\vec{x}) \leftarrow d \diamond f, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ in turn.

Initially $F_{j+1} = \emptyset$. The success pattern formulae f_i for the n body atoms are conjoined with f to obtain $g = f \wedge \wedge_{i=1}^{n} f_i$.

Variables not present in $p(\vec{x})$, $\{y_1 \ldots y_n\}$ say, are then eliminated from g by computing $g' = \exists_{y_1}(\ldots \exists_{y_n}(g))$ (weakening g).

Recall that weakening g does not compromise correctness since success patterns can be safety approximated from above.

Updating F_{j+1}

If F_{j+1} already contains a pattern of the form $\langle p(\vec{x}), g'' \rangle$, then this pattern is replaced with $\langle p(\vec{x}), g' \vee g'' \rangle$.

Otherwise F_{j+1} is revised to include $\langle p(\vec{x}), g' \rangle$.

Thus the success patterns become progressively weaker on each iteration until stability is achieved.

This iterative scheme will converge onto the lfp since iteration commences with the bottom element.

Least fixpoint calculation for Quicksort

For brevity, let $\vec{u} = \langle x_1, x_2 \rangle$, $\vec{v} = \langle x_1, x_2, x_3 \rangle$ and $\vec{w} = \langle x_1, x_2, x_3, x_4 \rangle$.

Then the lfp for the abstracted Quicksort program is obtained (and checked) in the following 3 iterations:

$$F_1 = \left\{ \begin{array}{c} \langle \mathtt{qs}(\vec{v}), x_1 \wedge (x_2 \Leftrightarrow x_3) \rangle \\ \langle \mathtt{pt}(\vec{w}), x_1 \wedge x_3 \wedge x_4 \rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2 \rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2 \rangle \end{array} \right\} \quad F_2 = \left\{ \begin{array}{c} \langle \mathtt{qs}(\vec{v}), x_2 \Leftrightarrow (x_1 \wedge x_3) \rangle \\ \langle \mathtt{pt}(\vec{w}), x_1 \wedge x_3 \wedge x_4 \rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2 \rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2 \rangle \end{array} \right\}$$

Finally, $F_3 = F_2$.

Note that the lfp faithfully describes the grounding behaviour: a qs goal will ground its second argument if it is called with its first and third arguments already ground and *vice versa*.

Call patterns in the gfp

A call pattern has the same syntactic form as a success pattern but instead describes those queries which are safe – those queries that do not violate any assertions.

The call pattern $\langle p(\vec{x}), f \rangle$ states that solving the query $\theta(p(\vec{x}))$ will not violate any assertions whenever $\theta \in \gamma(f)$.

For example $\langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_1 \wedge x_2 \rangle$ is a (correct) call pattern since the query $\theta(\mathtt{pt}(x_1, x_2, x_3, x_4))$ will never generate an instantiation error if $var(\theta(x_1)) = var(\theta(x_1)) = \emptyset$.

Lower approximation of call patterns

If the call pattern $\langle p(\vec{x}), f\rangle$ correctly describes the behaviour of $p(\vec{x})$ and $g \models f$, then the call pattern $\langle p(\vec{x}), g\rangle$ also correctly describes $p(\vec{x})$.

Let $\theta \in \gamma(g)$ and consider the query $\theta(p(\vec{x}))$.

Since $\theta \in \gamma(g)$, then $assign(\theta) \models g \models f$, hence $\theta \in \gamma(f)$.

Thus the call pattern $\langle p(\vec{x}), g\rangle$ also correctly describes $p(\vec{x})$.

Call patterns can thus be approximated from *below* without compromising correctness (but not necessarily from above).

Iteration for call patterns

Iteration commences with $D_0 = \{\langle p(\vec{x}), 1\rangle \mid p \in \Pi\}$ where Π is the set of predicate symbols occurring in the program.

D_{k+1} is computed from D_k by considering each clause $p(\vec{x}) \leftarrow d \diamond f, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ in turn and calculating a formula that defines a correct call pattern.

Initially set $D_{k+1} = D_k$ and then strengthen D_{k+1}.

Propagating requirements right-to-left

A correct call pattern is calculated by propagating dependencies requirements right-to-left by repeated application of $\Rightarrow$.

More exactly, let f_i denote the success pattern formula for $p_i(\vec{x}_i)$ in the previously computed lfp and let d_i denote the (current) call pattern formula for $p_i(\vec{x}_i)$ in D_k.

Set $e_{n+1} = 1$ and then compute $e_i = d_i \wedge (f_i \Rightarrow e_{i+1})$ for $1 \leq i \leq n$.

Intuition behind the construction

Suppose e_{i+1} is a correct call pattern for $p_{i+1}(\vec{x}_{i+1}), \ldots, p_n(\vec{x}_n)$.

Suggests $e_i \models d_i$ to ensure that $p_i(\vec{x}_i)$ does not error.

If $p_i(\vec{x}_i), \ldots, p_n(\vec{x}_n)$ is called with a substitution described by d_i, then $p_{i+1}(\vec{x}_{i+1}), \ldots, p_n(\vec{x}_n)$ is called with a substitution described by $(d_i \wedge f_i)$ since f_i describes the success patterns of $p_i(\vec{x}_i)$.

The dependency $(d_i \wedge f_i)$ may satisfy the e_{i+1} demand.

Suggests $e_i \models ((d_i \wedge f_i) \Rightarrow e_{i+1})$ – the *weakest* dependency that, in conjunction with $(d_i \wedge f_i)$, ensures that e_{i+1} holds.

Therefore put $e_i = d_i \wedge ((d_i \wedge f_i) \Rightarrow e_{i+1}) = d_i \wedge (f_i \Rightarrow e_{i+1})$.

Pseudo-complement

Step of calculating the *weakest* function that when conjoined with $d_i \wedge f_i$ implies e_{i+1}, is the very heart of the analysis.

For Pos, computing the weakest e_i reduces to applying the $\Rightarrow$ operator, but more generally, this step amounts to applying the pseudo-complement operator.

Strengthening lower approximations

Variables not present in $p(\vec{x})$, $\{y_1 \dots y_n\}$ say, are then eliminated by $g' = \forall_{y_1}(\dots\forall_{y_n}(g))$ (*strengthening* g).

A safe calling mode for this particular clause is then given by g'.

Eliminating variables from g by strengthening g is unusual and initially appears strange.

Updating D_{k+1}

D_{k+1} will contain a call pattern $\langle p(\vec{x}), g'' \rangle$ and this is updated with $\langle p(\vec{x}), g' \wedge g'' \rangle$ if $g'' \not\models g'$.

Call patterns become progressively stronger on each iteration.

This iterative scheme will converge onto the gfp since iteration commences with the top element $D_0 = \{\langle p(\vec{x}), 1 \rangle \mid p \in \Pi\}$.

Greatest fixpoint calculation for Quicksort

Quicksort generates the following D_k sequence:

$$D_0 = \left\{ \begin{array}{r} \langle \mathtt{qs}(\vec{v}), 1\rangle \\ \langle \mathtt{pt}(\vec{w}), 1\rangle \\ \langle =<'(\vec{u}), 1\rangle \\ \langle >'(\vec{u}), 1\rangle \end{array} \right\} D_1 = \left\{ \begin{array}{l} \quad\langle \mathtt{qs}(\vec{v}), 1\rangle \\ \quad\langle \mathtt{pt}(\vec{w}), 1\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \quad\langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{l} \quad\langle \mathtt{qs}(\vec{v}), 1\rangle \\ \quad\langle \mathtt{pt}(\vec{w}), x_2 \wedge (x_1 \vee (x_3 \wedge x_4))\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \quad\langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\} D_3 = \left\{ \begin{array}{l} \quad\langle \mathtt{qs}(\vec{v}), x_1\rangle \\ \quad\langle \mathtt{pt}(\vec{w}), x_2 \wedge (x_1 \vee (x_3 \wedge x_4))\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \quad\langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\}$$

The gfp is reached at D_3 since $D_4 = D_3$.

The gfp often infers surprising call patterns that can be easily missed by manual inspection.

Speed of the analysis on medium benchmarks

file	*size*	*abs*	*lfp*	*gfp*	*sum*	*file*	*size*	*abs*	*lfp*	*gfp*	*sum*
cry_mult	118	10	10	10	30	bryant	349	30	120	21	171
browse	125	10	10	0	20	ga	363	50	30	20	100
bid	128	10	10	0	20	neural	378	30	10	0	40
disj_r	148	30	0	10	40	press	381	30	20	0	50
consultant	151	20	0	10	30	peep	414	50	20	10	80
ncDP	156	10	10	0	20	nbody	421	40	20	20	80
tsp	162	30	20	10	60	eliza	432	50	20	0	70
elex_scanner	165	20	10	0	30	read	434	40	20	10	70
robot	165	10	10	0	20	simple_analyzer	512	90	701	20	811
sorts	172	0	10	10	20	ann	547	50	30	10	90
cs2	175	30	10	10	50	diffsimpsv	681	61	100	0	161
scc	175	10	141	0	151	arch1	692	50	40	10	100
bp0-6	201	20	10	0	30	asm	800	60	40	30	130
bnet	205	20	20	0	40	poker	962	81	70	10	161
jons	222	40	0	10	50	pentomino	981	50	40	80	170
mathlib	226	10	10	0	20	chat	1037	411	1422	1082	2915
intervals	230	20	10	10	40	sim_v5-2	1308	80	70	0	150
barnes_hut	240	40	30	40	110	semigroup	2328	180	90	60	350

Spin offs and related work

- "Missing link" between termination checking and termination inference is our backward analysis[†] (Israeli);
- Our analysis has recently lifted to (sharing) domains that are closed under linear implication[‡] (Italian);
- Our analysis relates to a compiler technique that derives (goal-independent) reuse conditions for Mercury[§] (Belgian).

[†] Genaim and Codish, International Conference on Logic for Programming, Artificial Intelligence and Reasoning, 2001.

[‡] Giacobazzi, Ranzato and Scozzari, Submitted to ACM TOPLAS, 2002.

[§] Mazur, Janssens and Bruynooghe, Computational Logic, 2000.