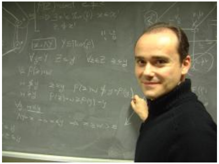
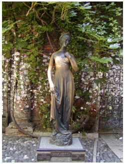
Debugging Concurrent (Logic) Programs with Abstract Interpretation

Samir Genaim and Andy King

Funded by the Royal Society short visit grant 16385



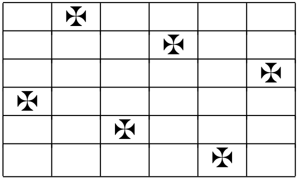



Generate-and-test search paradigm

- generate – place all the queens on the chessboard in some configuration;
- test – check whether the configuration is safe, that is, whether any one of the queens can take one another;
- repeat generate and test, searching until either a solution is found or all configurations are exhausted.

Applying generate-and-test to n-queens

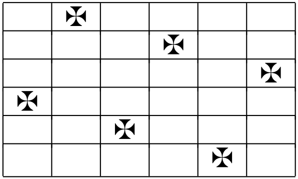
- if two queens occur in a row, then the configuration is unsafe;



$$\pi = \begin{Bmatrix} 6 \mapsto 2 \\ 5 \mapsto 4 \\ 4 \mapsto 6 \\ 3 \mapsto 1 \\ 2 \mapsto 3 \\ 1 \mapsto 5 \end{Bmatrix} \qquad L = [5, 3, 1, 6, 4, 2]$$

Applying generate-and-test to n-queens

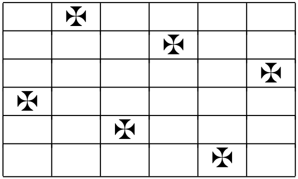
- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;



$$\pi = \left\{ \begin{array}{l} 6 \mapsto 2 \\ 5 \mapsto 4 \\ 4 \mapsto 6 \\ 3 \mapsto 1 \\ 2 \mapsto 3 \\ 1 \mapsto 5 \end{array} \right\} \qquad L = [5, 3, 1, 6, 4, 2]$$

Applying generate-and-test to n-queens

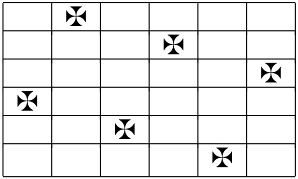
- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;
- exactly one queen occurs in each row;



$$\pi = \left\{ \begin{array}{l} 6 \mapsto 2 \\ 5 \mapsto 4 \\ 4 \mapsto 6 \\ 3 \mapsto 1 \\ 2 \mapsto 3 \\ 1 \mapsto 5 \end{array} \right\} \qquad L = [5, 3, 1, 6, 4, 2]$$

Applying generate-and-test to n-queens

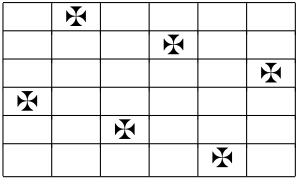
- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;
- exactly one queen occurs in each row;
- each (safe) configuration is a mapping $[1, n] \rightarrow [1, n]$ from a row number to a column number;



$$\pi = \left\{ \begin{array}{l} 6 \mapsto 2 \\ 5 \mapsto 4 \\ 4 \mapsto 6 \\ 3 \mapsto 1 \\ 2 \mapsto 3 \\ 1 \mapsto 5 \end{array} \right\} \qquad L = [5, 3, 1, 6, 4, 2]$$

Applying generate-and-test to n-queens

- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;
- exactly one queen occurs in each row;
- each (safe) configuration is a mapping $[1, n] \rightarrow [1, n]$ from a row number to a column number;
- each map is injective and surjective, hence a permutation.



$$\pi = \left\{ \begin{matrix} 6 \mapsto 2 \\ 5 \mapsto 4 \\ 4 \mapsto 6 \\ 3 \mapsto 1 \\ 2 \mapsto 3 \\ 1 \mapsto 5 \end{matrix} \right\} \qquad L = [5, 3, 1, 6, 4, 2]$$

```
main(Soln) :- perm([1,2,3,4,5,6], Soln), safe(Soln).

perm([], []).
perm(Ls, [X|Xs]) :- select(X, Ls, Rs), perm(Rs, Xs).

select(X, [X|Xs], Xs).
select(X, [CN|CNs], [CN|Rs]) :- select(X, CNs, Rs).

safe([]).
safe([CN | CNs]):- no_attack(CNs, CN, 1), safe(CNs).

no_attack([], _, _).
no_attack([CN|CNs], First_CN, Diff) :-
    diagonal(Diff, First_CN, CN), Next_Diff is Diff + 1,
    no_attack(CNs, First_CN, Next_Diff).

diagonal(Diff, First_CN, CN) :- Diff =\= abs(First_CN  CN).
```

Test-and-generate search paradign

- generate – place one new queen on the chessboard to construct a configuration incrementally;
- test – check whether the new queen is safe as soon as it is placed on the board; discard partial configurations that are definitely unsafe.
- repeat incremental generation and testing, searching until either a solution is found or all configurations are exhausted.

```
main(Soln) :-
    length(Soln, 6),
    safe(Soln),
    perm([1,2,3,4,5,6], Soln).

:- block diagonal(?, -, ?), diagonal(?, ?, -).
diagonal(Diff, First_CN, CN) :- Diff =\= abs(First_CN  CN).
```

n	*G-and-T*	*T-and-G*
10	30.313	0.107
11	>60	0.063
12	>60	0.380
13	>60	0.176
14	>60	3.300
15	>60	2.659
16	>60	21.808

Related work

- Abstract interpretation schemes have been proposed by Bigot, Codish, Codognet, Winsborough, etc for checking that a program and goal cannot reduce to such a possibly problematic suspension state.
- They simulate the operational semantics by tracing the execution of the program over a finite (though possibly large) collection of abstract states.
- These schemes either return:
  - "yes" – the program and goal *definitely* cannot reduce to a suspension state;

Related work

- Abstract interpretation schemes have been proposed by Bigot, Codish, Codognet, Winsborough, etc for checking that a program and goal cannot reduce to such a possibly problematic suspension state.
- They simulate the operational semantics by tracing the execution of the program over a finite (though possibly large) collection of abstract states.
- These schemes either return:
  - "yes" – the program and goal *definitely* cannot reduce to a suspension state;
  - "don't know" – program and goal *may* reduce to a suspension.

User-interface issues

The programmer:

- should be able to activate analysis with minimal interaction;

Visit `http://www.sci.univr.it/~genaim/www/susweb/bin/susweb.cgi` to see how bottom-up analysis can address these user-interface problems.

User-interface issues

The programmer:

- should be able to activate analysis with minimal interaction;
- sometimes will need to *carefully* scrutinise the results;

Visit `http://www.sci.univr.it/~genaim/www/susweb/bin/susweb.cgi` to see how bottom-up analysis can address these user-interface problems.

User-interface issues

The programmer:

- should be able to activate analysis with minimal interaction;
- sometimes will need to *carefully* scrutinise the results;
- should not hesitate about applying the analysis even to the largest programs.

Visit `http://www.sci.univr.it/~genaim/www/susweb/bin/susweb.cgi` to see how bottom-up analysis can address these user-interface problems.

Bugs from Arizona and Kent

For bessel, the analysis inferred a call pattern of *false* for the predicate `bessel`, the problem stemming from the clause:

```
bessel(0, X, Y1, Y2) :- Y2 = 0.0, j0(10, X, Y).
```

For queens_control, the analysis only inferred that a certain predicate, `perm`, will not suspend if its first argument is ground:

```
:- block perm_aux(-, ?, ?).  perm_aux(?, -, ?).
perm_aux(D1, D2, D) :- D1 = D2, D = D1.
```

A Bug from Argonne National Labs

For ssd, a call pattern of *false* was inferred was traced to the following predicate:

```
next_play( Remaining, Board, History, D) :-
    Remaining =[] |
    length( Board, Len),
    First is (2 * Len) // 3,
    try_pent( [], Remaining, ..., History, D).
%next_play( [], _, History, D) :-
%   print_history( " SOLN ", History, D).
next_play( [], _, History, D).
```

Bugs from Manchester Metropolitan University

The predicate lhs_strip_DmTm includes a debugging/error handling case that merely calls pp (! flushes the buffer):

```
lhs_strip_DmTm([],_,_,_,_):-
    pp('ERROR {Dm,Tm} not found in PiSet')!.
```

This clause does ground its third, fourth and fifth arguments.

```
lhs_strip_DmTm([],_,C,D,E):-
    pp('ERROR {Dm,Tm} not found in PiSet')!,
    C := error, D := error, E := error.
```

It is arguably better practise to abort the computation by binding the output arguments to rogue values.

A false positive from Oregon/ICOT

For the program semigroup, non-suspension could only not be shown for the top-level predicate `main`:

```
main(N) :-
    kernel(K),
    append([begin|K],[end|R],S),
    spawn(S,R,Out,[]), count(Out,N).
```

- The analysis infers that `spawn(S,R,Out,[])` will not suspend if both `S` and `R` are ground (correct but crude);
- Neither `S` nor `R` are ground at the time of the call (though `kernel(K)` binds `K` to a ground structure);
- `spawn` actually implements a form of pipelined filter where the input stream `S` is fed by the output stream `R`.

Timing and precision results table

		precision			*time (msecs)*				
source	*program*	*preds*	*blocks*	*%*	*abs*	*SCC*	*lfp*	*gfp*	*total*
Debray	combo	10	10	100	9	2	2	3	16
	transp	11	11	100	12	2	1	4	19
	deriv	7	7	100	9	1	1	2	13
Foster	insert	8	8	100	9	1	1	2	13
	btree	10	10	100	11	2	2	4	19
Howe	entails	8	8	100	7	1	1	3	12
Huntbach	colouring	42	37	88	42	9	30	40	121
	spanning	76	71	93	81	28	84	153	346
	eight_puzzle	97	88	91	100	40	75	211	426
Johnson	PTMddd	319	316	99	476	592	785	25138	26991
Naish	queens	10	10	100	7	1	1	3	12
King	msort_control	14	14	100	14	3	3	5	25
	queens_control	15	15	100	13	3	2	4	22
Tick	bestpath	20	11	55	39	4	11	19	73
	pascal	23	23	100	22	5	5	8	40
	semigroup	20	19	95	21	4	6	11	42
	mastermind	20	20	100	28	5	6	148	187
	nand	25	25	100	32	6	8	15	61

Monotonic and definite Boolean functions

- Let $Bool_X$ denote the set of propositional formulae over X.
- $Mon_X \subseteq Bool_X$ are those formulae which can be constructed only from $\vee$, $\wedge$ and X, ie, $\mathtt{X} \wedge (\mathtt{Y} \vee \mathtt{Z})$ where $X = \{\mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$.
- $Def_X \subseteq Bool_X$ are those formulae which are conjunctions of propositional Horn formulae, ie, $(\mathtt{W} \leftarrow (\mathtt{X} \wedge \mathtt{Y})) \wedge (\mathtt{Z} \leftarrow \mathit{true})$.

Now suppose $X = \{\mathtt{X}, \mathtt{Y}\}$. Let $model_X(\mathtt{X} \wedge \mathtt{Y}) = \{\{\mathtt{X}, \mathtt{Y}\}\}$, $model_X(\mathtt{X} \vee \mathtt{Y}) = \{\{\mathtt{X}\}, \{\mathtt{Y}\}, \{\mathtt{X}, \mathtt{Y}\}\}$ and $model_X(\mathtt{X} \leftarrow \mathtt{Y}) = \{\emptyset, \{\mathtt{X}\}, \{\mathtt{X}, \mathtt{Y}\}\}$.

- $f \in Def_X$ iff $\forall M, M' \in model_X(f). M \cap M' \in model_X(f)$;
- $f \in Mon_X$ iff $\forall M \in model_X(f) \forall M \subseteq M' \subseteq X. M' \in model_X(f)$;
- Finally let $f_1, f_2 \in Bool_X$. $f_1 \models f_2$ iff $model_X(f_1) \subseteq model_X(f_2)$.

Reordering compound goals without actually reordering

```
:- block p(-, ?).
p(X, Z) :- Z = 1.

:- block q(-, ?), q(?, -).
q(X, Y) :- true.

:- block r(?, -).
r(Y, Z) :- Y = 2.
```

$d_i \in Mon_X$	$g_i \in Bool_X$
$d_1 = \mathtt{X}$	$g_1 = \mathtt{Z}$
$d_2 = \mathtt{X} \wedge \mathtt{Y}$	$g_2 = true$
$d_3 = \mathtt{Z}$	$g_3 = \mathtt{Y}$

The *compound* goal `p(X,Z), q(X,Y), r(Y,Z)` can be executed without incurring a suspension if it is called with `X` ground.

The problem is to infer such a non-suspension property for the compound goal given d_i and g_i which describe non-suspension requirements and the success patterns for the atomic sub-goals.

Inferring a non-suspension requirement, f say, for the compound goal from the d_i and g_i

Proposition

- Let $g_i \in Bool_X$ and $d_i \in Mon_X$ for all $i \in [1, m]$.
- Let $f \in Def_X$ where $f \models d = (\wedge_{i=1}^{m}(d_i \rightarrow g_i)) \rightarrow (\wedge_{j=1}^{m} d_j)$.
- Then there exists $i \in [1, m]$ such that $f \models d_i$.

We are interested in $m = 3$ and $X = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$. Moreover:

$$\wedge_{i=1}^{3}(d_i \rightarrow g_i) = (\mathtt{X} \rightarrow \mathtt{Z}) \wedge (\mathtt{Z} \rightarrow \mathtt{Y}) \quad \wedge_{i=1}^{3} d_i = \mathtt{X} \wedge \mathtt{Y} \wedge \mathtt{Z} \qquad d = \ldots$$

Any $f \in Def_X$ such that $f \models d$ describes a state under which the compound goal can be executed without suspension.
To illustrate, consider $f = \mathtt{X} = \mathtt{X} \leftarrow \mathit{true} \in Def_X$.
Observe that $f \wedge \wedge_{i=1}^{3}(d_i \rightarrow g_i) \models \mathtt{X} \wedge \mathtt{Y} \wedge \mathtt{Z} \models (\wedge_{i=1}^{3} d_i)$.
Hence $f \models \wedge_{i=1}^{3}(d_i \rightarrow g_i) \rightarrow (\wedge_{i=1}^{3} d_i)$ and indeed $f = \mathtt{X} = d_1$.

Non-suspension of the remaining sub-goals

The state after `p(X, Z)` is described by $f \wedge g_1 = \mathtt{X} \wedge \mathtt{Z} \in Def_X$.

- Recall $f \models \wedge_{i=1}^{3}(d_i \rightarrow g_i) \rightarrow (\wedge_{i=1}^{3} d_i)$.
- Hence $f \wedge \wedge_{i=1}^{3}(d_i \rightarrow g_i) \models (\wedge_{i=1}^{3} d_i)$.
- Since $f \wedge g_1 \models f$, it follows $(f \wedge g_1) \wedge \wedge_{i=1}^{3}(d_i \rightarrow g_i) \models f \wedge \wedge_{i=1}^{3}(d_i \rightarrow g_i) \models (\wedge_{i=1}^{3} d_i)$.
- Moreover $g_1 \models (d_1 \rightarrow g_1)$, thus $(f \wedge g_1) \wedge \wedge_{i=2}^{3}(d_i \rightarrow g_i) \models (\wedge_{i=1}^{3} d_i)$.
- But $(\wedge_{i=1}^{3} d_i) \models (\wedge_{i=2}^{3} d_i)$, hence $(f \wedge g_1) \wedge \wedge_{i=2}^{3}(d_i \rightarrow g_i) \models (\wedge_{i=2}^{3} d_i)$.
- Therefore $(f \wedge g_1) \models \wedge_{i=2}^{3}(d_i \rightarrow g_i) \rightarrow (\wedge_{i=2}^{3} d_i)$.

Reapplying the proposition, there must exist $i \in [2, 3]$ such that $f \wedge g_1 \models d_i$. Indeed $f \wedge g_1 \models \mathtt{X} \wedge \mathtt{Z} \models \mathtt{Z} = d_3$, hence the third sub-goal can be executed without suspension.

Conclusions

- Backward analysis leads to a lightweight point-and-click approach to (partial) verification;
- Monotonic reordering results ensures scalability;
- The domain of boolean functions in rich enough to locate suspension bugs in real programs;
- Speed *very* significant in finding the needle in the haystack.