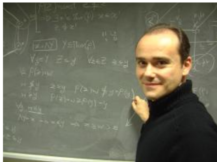
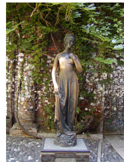
Debugging Concurrent (Logic) Programs with Abstract Interpretation

Samir Genaim and Andy King

Funded by the Royal Society short visit grant 16385





Outline of this talk

The role of concurrency in search
- Generate-and-test search paradigm
- Test-and-generate search paradigm

User-interface issues in suspension analysis

Applying suspension analysis
- Bugs
- False positives
- Timing and Precision Results Summary

How the analysis works
- Program abstraction
- lfp – macro-scale
- gfp calculation – macro-scale
- gfp calculation – micro-scale

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;

	✠				
			✠		
					✠
✠					
		✠			
				✠	

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

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;

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

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;

	✠				
			✠		
					✠
✠					
		✠			
				✠	

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

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;

	✠				
			✠		
					✠
✠					
		✠			
				✠	

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

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;
- 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.

	✠				
			✠		
					✠
✠					
		✠			
				✠	

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

```
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 paradigm

- 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) :-
    Remaing =\= [] |
    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).
```

An anomaly 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);

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);

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

source	program	precision			time (msecs)				
		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

Analysis target

```
:- block app(-, ?, -).
app([], X, X).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).

inorder(nil,[]).
inorder(tree(L,V,R),I) :-
    app(LI,[V|RI],I),
    inorder(L,LI),
    inorder(R,RI).
```

```
?- inorder(T, [a,b]).
T = tree(nil,a,tree(nil,b,nil)) ? ;
T = tree(tree(nil,a,nil),b,nil) ? ;
no
```

```
?- inorder(tree(nil, a, tree(nil, b, nil)), L).
L = [a,b] ? ;
no
```

Program abstraction

```
:- block app(-, ?, -).
app([], X, X).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).

inorder(nil,[]).
inorder(tree(L,V,R),I) :-
    app(LI,[V|RI],I),
    inorder(L,LI),
    inorder(R,RI).
```

app(L, Ys, A) :-
  assert($L \vee A$);
  $L \wedge (Ys \leftrightarrow A)$.
app(L, Ys, A) :-
  assert($L \vee A$);
  $L \leftrightarrow (X \wedge Xs)$,
  $A \leftrightarrow (X \wedge Zs)$,
  app(Xs,Ys,Zs).

inorder(T, I) :-
  $T \wedge I$.
inorder(T, I) :-
  $T \leftrightarrow (L \wedge V \wedge R)$,
  $A \leftrightarrow (V \wedge RI)$,
  app(LI, A, I),
  inorder(L, LI),
  inorder(R, RI).

lfp calculation – macro-scale

- The success patterns of the program are computed as a lfp (simulating the T_P operator);

lfp calculation – macro-scale

- The success patterns of the program are computed as a lfp (simulating the T_P operator);
- A success pattern is an atom with distinct variables for arguments paired with a Boolean formula over those variables;

lfp calculation – macro-scale

- The success patterns of the program are computed as a lfp (simulating the T_P operator);
- A success pattern is an atom with distinct variables for arguments paired with a Boolean formula over those variables;
- The lfp of the program is the following:

$$F = \left\{ \begin{array}{lcl} inorder(x_1, x_2) & \text{:-} & x_1 \leftrightarrow x_2, \\ app(x_1, x_2, x_3) & \text{:-} & (x_1 \wedge x_2) \leftrightarrow x_3 \end{array} \right\}$$

lfp calculation – macro-scale

- The success patterns of the program are computed as a lfp (simulating the T_P operator);
- A success pattern is an atom with distinct variables for arguments paired with a Boolean formula over those variables;
- The lfp of the program is the following:

$$F = \left\{ \begin{array}{lcl} inorder(x_1, x_2) & \text{:-} & x_1 \leftrightarrow x_2, \\ app(x_1, x_2, x_3) & \text{:-} & (x_1 \wedge x_2) \leftrightarrow x_3 \end{array} \right\}$$

- Observe that F faithfully describes the grounding behaviour of inorder and app (assuming termination)

gfp calculation – macro-scale

- A gfp is computed to characterise the call patterns of the program; a call pattern has the same syntactic form as a success pattern
- Iteration commences with $D_0 = \top$ and incrementally strengthens the call pattern formulae until they describe queries that do not violate the assertions:

$$D_1 = \left\{ \begin{array}{lcl} \mathit{inorder}(x_1, x_2) & \text{:-} & \mathit{true}, \\ \mathit{app}(x_1, x_2, x_3) & \text{:-} & x_1 \vee x_3 \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{lcl} \mathit{inorder}(x_1, x_2) & \text{:-} & x_1 \vee x_2, \\ \mathit{app}(x_1, x_2, x_3) & \text{:-} & x_1 \vee x_3 \end{array} \right\} = D_3$$

gfp calculation – under the microscope (part I)

- Compute $e = \wedge_{i=1}^{n}(d_i \rightarrow f_i)$ to capture the grounding behaviour of the compound goal $p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ where f_i and d_i are the success and call patterns for $p_i(\vec{x}_i)$
- Consider $p_1(\vec{x}_1), \ldots, p_3(\vec{x}_3) =$ $app(LI, A, I), inorder(L, LI), inorder(R, RI)$ with

$$\begin{array}{lll} d_1 = LI \vee I & d_2 = L \vee LI & d_3 = R \vee RI \\ f_1 = (LI \wedge A) \leftrightarrow I & f_2 = L \leftrightarrow LI & f_3 = R \leftrightarrow RI \end{array}$$

- Then $e = (d_1 \rightarrow f_1) \wedge (d_2 \rightarrow f_2) \wedge (d_3 \rightarrow f_3)$ where

$$\begin{array}{l} d_1 \rightarrow f_1 = (LI \rightarrow (A \leftrightarrow I)) \wedge (I \rightarrow (LI \wedge A)) \\ d_2 \rightarrow f_2 = (L \rightarrow LI) \wedge (LI \rightarrow L) \\ d_3 \rightarrow f_3 = (R \rightarrow RI) \wedge (RI \rightarrow R) \end{array}$$

gfp calculation – under the microscope (part II)

- Compute $e' = \wedge_{i=1}^{n} d_i$ which is a groundness property sufficient for scheduling *all* the goals of $p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ without suspension;
- Too draconian since one goal can activate another;
- Compute $e \rightarrow e'$ which is a weaker groundness property;
- If $e \rightarrow e'$ holds when $p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ is called, then a permutation π over $[1, n]$ exists such that the re-ordered goals $p_{\pi(1)}(\vec{x}_{\pi(1)}), \ldots, p_{\pi(n)}(\vec{x}_{\pi(n)})$ does not suspend;
- Then $e \rightarrow e' = f_1 \vee f_2$ where $f_1 = (I \wedge (R \vee RI))$, $f_2 = (L \wedge (R \vee RI))$ which correspond to the orderings:
  - $app(LI, A, I), inorder(L, LI), inorder(R, RI)$
  - $inorder(L, LI), inorder(R, RI), app(LI, A, I)$