Horn Clauses for Program Analysis

Helmut Seidl

Workshop in Honor of Hubert Comon-Lundh, 2008

Overview

- Data-flow Analysis of Imperative Programs
- Set-based Analysis of Functional Programs
- Set-based Analysis of Logic Programs
- Horn Clauses for CCC

1. Dataflow Analysis

Idea

- Dataflow analysis computes for every program point a finite number of facts.
- The result can be represented as a recursively defined finite relation.

Idea

- Dataflow analysis computes for every program point a finite number of facts.
- The result can be represented as a recursively defined finite relation.
- Recursive finite relations can be conveniently specified in Datalog and efficiently tabulated through standard fixpoint techniques.
- Structured terms are not essential but convenient ...

Related Work

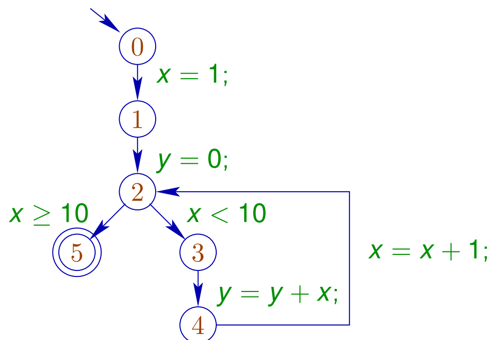
Datalog	...
Tabeled Resolution	...
Ganzinger/McAllester	2001
Nielson/Riis-Nielson/S.	2002

Representation of Programs

Constrol-flow Graph $\Leftrightarrow$ Relation

Representation of Programs

Constrol-flow Graph $\Leftrightarrow$ Relation



Representation of Programs

Constrol-flow Graph $\Leftrightarrow$ Relation

$$
\begin{array}{ll}
\text{var}(x) & \Leftarrow \\
\text{var}(y) & \Leftarrow \\
\\
\text{edge}(0, \mathit{assign}(x, 1), 1) & \Leftarrow \\
\text{edge}(1, \mathit{assign}(y, 0), 2) & \Leftarrow \\
\text{edge}(2, \mathit{test}(\mathit{op}(\mathit{less}, x, 10)), 3) & \Leftarrow \\
\text{edge}(3, \mathit{assign}(y, \mathit{op}(\mathit{plus}, y, x), 4) & \Leftarrow \\
\text{edge}(4, \mathit{assign}(x, \mathit{op}(\mathit{plus}, x, 1), 2) & \Leftarrow \\
\text{edge}(2, \mathit{test}(\mathit{op}(\mathit{geq}, x, 10)), 5) & \Leftarrow
\end{array}
$$

Clauses for Liveness

$$\begin{array}{lcl} \text{live}(U,X) & \Leftarrow & \text{edge}(U,Lab,V), \text{used}(X,Lab) \\ \text{live}(U,X) & \Leftarrow & \text{edge}(U,Lab,V), \text{live}(V,X), \\ & & \text{not}(\text{def}(X,Lab)) \end{array}$$

Clauses for Liveness

$$
\begin{array}{lcl}
\text{live}(U, X) & \Leftarrow & \text{edge}(U, \mathit{Lab}, V), \text{used}(X, \mathit{Lab}) \\
\text{live}(U, X) & \Leftarrow & \text{edge}(U, \mathit{Lab}, V), \text{live}(V, X), \\
 & & \text{not}(\text{def}(X, \mathit{Lab})) \\
\text{def}(X, \mathit{assign}(X, _)) & \Leftarrow & \\
\text{used}(X, \mathit{assign}(_, E)) & \Leftarrow & \mathit{occurs}(X, E) \\
\text{used}(X, \mathit{test}(E)) & \Leftarrow & \mathit{occurs}(X, E)
\end{array}
$$

Clauses for Liveness

$$
\begin{array}{lcl}
\mathrm{live}(U, X) & \Leftarrow & \mathrm{edge}(U, Lab, V), \mathrm{used}(X, Lab) \\
\mathrm{live}(U, X) & \Leftarrow & \mathrm{edge}(U, Lab, V), \mathrm{live}(V, X), \\
 & & \mathrm{not}(\mathrm{def}(X, Lab)) \\
\mathrm{def}(X, assign(X, _)) & \Leftarrow & \\
\mathrm{used}(X, assign(_, E)) & \Leftarrow & occurs(X, E) \\
\mathrm{used}(X, test(E)) & \Leftarrow & occurs(X, E) \\
occurs(X, X) & \Leftarrow & var(X) \\
occurs(X, op(_, E)) & \Leftarrow & occurs(X, E) \\
occurs(X, op(_, E1, E2)) & \Leftarrow & occurs(X, E1) \\
occurs(X, op(_, E1, E2)) & \Leftarrow & occurs(X, E2)
\end{array}
$$

Discussion

- The formulation requires the predicate $\mathsf{not}/1$

$\Longrightarrow$ stratified negation

Discussion

- The formulation requires the predicate not/1

$\Longrightarrow$ stratified negation

- The auxiliary predicate occurs/2 is not finite

$\Longrightarrow$ bottom-up tabulation will not terminate !

Discussion

- The formulation requires the predicate $\text{not}/1$

$\Longrightarrow$ stratified negation

- The auxiliary predicate $\text{occurs}/2$ is not finite

$\Longrightarrow$ bottom-up tabulation will not terminate !

$\Longrightarrow$ Topdown solving

Result for Liveness

$$
\begin{array}{llll}
\text{live}(1, x) & \Leftarrow & & \\
\text{live}(2, x) & \Leftarrow & \text{live}(2, y) & \Leftarrow \\
\text{live}(3, x) & \Leftarrow & \text{live}(3, y) & \Leftarrow \\
\text{live}(4, x) & \Leftarrow & \text{live}(4, y) & \Leftarrow
\end{array}
$$

Summary

- Horn clauses (with stratified negation) provide a convenient tool for dataflow analyses.
- The specification is program independent :-)
- Topdown solving provides an efficient algorithm for computing all valid dataflow facts.

2. Functional Programs

Idea

- Determine for every sub-expression e in the program a (safe super-) set of the terms to which e evaluates.

Idea

- Determine for every sub-expression e in the program a (safe super-) set of the terms to which e evaluates.
- Describe these sets by means of regular tree grammars Jones 1987

Idea

- Determine for every sub-expression e in the program a (safe super-) set of the terms to which e evaluates.
- Describe these sets by means of regular tree grammars — Jones 1987
- Use set constraints instead — Heintze 1994

Idea

- Determine for every sub-expression e in the program a (safe super-) set of the terms to which e evaluates.
- Describe these sets by means of regular tree grammars Jones 1987
- Use set constraints instead Heintze 1994
- Why not just Horn clauses ???

The append Function

$$
\begin{array}{rcl}
\textbf{let rec } a \ = \ \textbf{fun } x & \rightarrow & \textbf{match } x \textbf{ with} \\
[\,] & \rightarrow & \textbf{fun } y \rightarrow y \\
h{::}t & \rightarrow & \textbf{fun } y \rightarrow h :: a\ t\ y
\end{array}
$$

$\textbf{in } a\,[1;2]\ [3]$

The Horn Clauses

$$
\begin{array}{lll}
\mathsf{val}_{a}(\mathbf{fun}(x, \mathsf{match})) & \Leftarrow & \\
\mathsf{val}_{\mathsf{match}}(\mathbf{fun}(y, y)) & \Leftarrow & \mathsf{val}_x([\,]) \\
\mathsf{val}_{\mathsf{match}}(\mathbf{fun}(y, \mathsf{cons})) & \Leftarrow & \mathsf{val}_x(_ :: _) \\
\mathsf{val}_h(Z) & \Leftarrow & \mathsf{val}_x(Z :: _) \\
\mathsf{val}_{\mathsf{cons}}(H :: T) & \Leftarrow & \mathsf{val}_h(H), \mathsf{val}_{\mathsf{a_t_y}}(T) \\
\mathsf{val}_{\mathsf{a_t_y}}(Z) & \Leftarrow & \mathsf{val}_{\mathsf{a_t}}(\mathbf{fun}(x, \mathsf{match})), \mathsf{val}_{\mathsf{match}}(Z) \\
 & \ldots &
\end{array}
$$

Discussion

- All heads of the generated constraints are linear !
  No variable occurs repeatedly in bodies !!

$\Longrightarrow \quad H_3$

Discussion

- All heads of the generated constraints are linear !
  No variable occurs repeatedly in bodies !!

$\Longrightarrow \quad H_3$

- For every set of H_3-clauses, there is an equivalent set of automata clauses.

Discussion

- All heads of the generated constraints are linear !
  No variable occurs repeatedly in bodies !!

$\Longrightarrow$ H_3

- For every set of H_3-clauses, there is an equivalent set of automata clauses.
- These can be found in polynomial time :-)

The Result

$$
\begin{array}{lcl}
\mathsf{val}_{\mathsf{let}}(Z_1 :: Z_2) & \Leftarrow & \mathsf{val}_3(Z_1), \mathsf{val}_{[\,]}(Z_2) \\
\mathsf{val}_{\mathsf{let}}(Z_1 :: Z_2) & \Leftarrow & \mathsf{val}_h(Z_1), \mathsf{val}_{\mathsf{a_t_y}}(Z_2) \\
\mathsf{val}_{\mathsf{a_t_y}}(Z_1 :: Z_2) & \Leftarrow & \mathsf{val}_3(Z_1), \mathsf{val}_{[\,]}(Z_2) \\
\mathsf{val}_{\mathsf{a_t_y}}(Z_1 :: Z_2) & \Leftarrow & \mathsf{val}_h(Z_1), \mathsf{val}_{\mathsf{a_t_y}}(Z_2) \\
\mathsf{val}_h(1) & \Leftarrow & \\
\mathsf{val}_h(2) & \Leftarrow & \\
\mathsf{val}_3(3) & \Leftarrow & \\
\mathsf{val}_{[\,]}([\,]) & \Leftarrow &
\end{array}
$$

Discussion

- The analysis finds:

$$\{[x_1; \ldots; x_k; 3] \mid x_i \in \{1, 2\}\}$$

- The analysis does not discover that the set of all occurring lists is finite :-(

Discussion

- The analysis finds:

$$\{[x_1; \ldots; x_k; 3] \mid x_i \in \{1, 2\}\}$$

- The analysis does not discover that the set of all occurring lists is finite :-(
- The clauses are program-specific!

The analysis consists of two steps:

(1) Constraint generation

(2) Constraint solving

Alternative: Set Constraints

Heintze 1994

$$
\begin{array}{lll}
\mathsf{val}_a & \supseteq & \{\mathbf{fun}(x, \mathsf{match})\} \\
\mathsf{val}_{\mathsf{match}} & \supseteq & (\{[\,]\} \cap \mathsf{val}_x \neq \emptyset);\; \{\mathbf{fun}(y, y)\} \\
\mathsf{val}_{\mathsf{match}} & \supseteq & (_::_ \cap \mathsf{val}_x \neq \emptyset);\; \{\mathbf{fun}(y, \mathsf{cons})\} \\
\mathsf{val}_h & \supseteq & \pi_{::,1}\, \mathsf{val}_x \\
\mathsf{val}_{\mathsf{cons}} & \supseteq & \mathsf{val}_h :: \mathsf{val}_{\mathsf{a_t_y}} \\
\mathsf{val}_{\mathsf{a_t_y}} & \supseteq & (\{\mathbf{fun}(x, \mathsf{match})\} \cap \mathsf{val}_{\mathsf{a_t}} \neq \emptyset);\; \mathsf{val}_{\mathsf{match}} \\
 & \ldots &
\end{array}
$$

Discussion

- The analysis requires the set operators:

Projection	$\pi_{::,i}\, X$
Constructor application	$X :: Y$
Check	$(e \cap X \neq \emptyset);\, Y$

Discussion

- The analysis requires the set operators:

Projection	$\pi_{::,i}\, X$
Constructor application	$X :: Y$
Check	$(e \cap X \neq \emptyset);\, Y$

- All these can be modelled by H_3-clauses :-)

Program-independent Formulation

$$
\begin{array}{lll}
\mathsf{val}(A, Z) & \Leftarrow & \\
& & \mathsf{val}(E, Z) \\
& \ldots &
\end{array}
$$

Program-independent Formulation

$$
\begin{array}{lll}
\text{val}(A, Z) & \Leftarrow & \\
 & & \text{val}(E, Z) \\
 & \ldots & \\
\text{val}(\textbf{fun}(X, E), \textbf{fun}(X, E)) & \Leftarrow & \\
\text{val}(H :: T, Z_1 :: Z_2) & \Leftarrow & \\
 & & \text{val}(H, Z_1), \text{val}(T, Z_2)
\end{array}
$$

Program-independent Formulation

$$\mathsf{reach}(\mathbf{let}(\mathsf{a}, \ldots, \mathbf{app}(\mathbf{app}(\mathsf{a}, [1;2]), [3]))) \Leftarrow$$

$$\begin{array}{lll}
\mathsf{reach}(E) & \Leftarrow & \mathsf{reach}(\mathbf{let}(_, E, _)) \\
\mathsf{val}(A, Z) & \Leftarrow & \\
 & & \mathsf{val}(E, Z) \\
 & \ldots & \\
\mathsf{val}(\mathbf{fun}(X, E), \mathbf{fun}(X, E)) & \Leftarrow & \\
\mathsf{val}(H :: T, Z_1 :: Z_2) & \Leftarrow & \\
 & & \mathsf{val}(H, Z_1), \mathsf{val}(T, Z_2)
\end{array}$$

Program-independent Formulation

$$\text{reach}(\textbf{let}(\text{a}, \ldots, \textbf{app}(\textbf{app}(\text{a}, [1;2]), [3]))) \Leftarrow$$

$$
\begin{array}{lcl}
\text{reach}(E) & \Leftarrow & \text{reach}(\textbf{let}(_, E, _)) \\
\text{val}(A, Z) & \Leftarrow & \text{reach}(\textbf{let}(A, E, _)), \\
 & & \text{val}(E, Z) \\
 & \ldots & \\
\text{val}(\textbf{fun}(X, E), \textbf{fun}(X, E)) & \Leftarrow & \text{reach}(\textbf{fun}(X, E)) \\
\text{val}(H :: T, Z_1 :: Z_2) & \Leftarrow & \text{reach}(H :: T), \\
 & & \text{val}(H, Z_1), \text{val}(T, Z_2)
\end{array}
$$

Discussion

- The rules both for functions and constructor applications violate the restrictions of our favorite class H_1 :-(

Discussion

- The rules both for functions and constructor applications violate the restrictions of our favorite class H_1 :-(
- Finiteness analysis of the predicate reach/1 on the other hand, allows to instantiate variables with finite range !

Discussion

- The rules both for functions and constructor applications violate the restrictions of our favorite class H_1 :-(
- Finiteness analysis of the predicate reach/1 on the other hand, allows to instantiate variables with finite range !
- Instantiation will recover the extensive form :-)

Summary

- Horn clauses provide a convenient tool for approximating the collecting semantics :-)

Summary

- Horn clauses provide a convenient tool for approximating the collecting semantics :-)
- A program-dependent specification can be obtained via H_3-clauses — which can be recovered from general Horn clauses through instantiation of finite predicates :-)

Summary

- Horn clauses provide a convenient tool for approximating the collecting semantics :-)
- A program-dependent specification can be obtained via H_3-clauses — which can be recovered from general Horn clauses through instantiation of finite predicates :-)
- A trivial finiteness analysis may suffice.

3. Prolog

Idea

- Approximate the least model of a program via set constraints — Heintze/Jaffar 1990

Idea

- Approximate the least model of a program via set constraints
  Heintze/Jaffar 1990
- Use uniform Horn clauses instead!
  Frühwirth et al. 1991

Idea

- Approximate the least model of a program via set constraints
  Heintze/Jaffar 1990
- Use uniform Horn clauses instead!
  Frühwirth et al. 1991
- Why not approximate with H_1 ?

Weidenbach 1999

Nielson/Riis-Nielson/S. 2002

Goubault-Larrecq 2005

The Class of H_1-Clauses

H_1-Clauses are of the form:

$$p(X_1, \ldots, X_k) \Leftarrow any \quad \text{or} \quad p(a(X_1, \ldots, X_k)) \Leftarrow any$$

The Class of H_1-Clauses

H_1-Clauses are of the form:

$$p(X_1, \ldots, X_k) \Leftarrow any \quad \text{or} \quad p(a(X_1, \ldots, X_k)) \Leftarrow any$$

For any set of H_1-clauses an equivalent set of automata clauses can be constructed in exponential time :-)

The Class of H_1-Clauses

H_1-Clauses are of the form:

$$p(X_1, \ldots, X_k) \Leftarrow \mathit{any} \quad \text{or} \quad p(a(X_1, \ldots, X_k)) \Leftarrow \mathit{any}$$

For any set of H_1-clauses an equivalent set of automata clauses can be constructed in exponential time :-)

Any set of Horn clauses can be approximated by a set of H_1-clauses :-)

Example: append

$$
\begin{array}{lcl}
\text{app}([\,], Y, Y) & \Leftarrow & \\
\text{app}([H|T], Y, [H|Z]) & \Leftarrow & \text{app}(T, Y, Z) \\
 & \Leftarrow & \text{app}([1,2], [3], Z)
\end{array}
$$

... results in:

The H_1-Approximation

$$\begin{array}{lcl} \mathrm{app}(Z, Y, Y') & \Leftarrow & \mathrm{aux}_{[\,]}(Z) \\ \mathrm{aux}_{[\,]}([\,]) & \Leftarrow & \end{array}$$

The H_1-Approximation

$$\begin{array}{lcl} \text{app}(Z, Y, Y') & \Leftarrow & \text{aux}_{[\,]}(Z) \\ \text{aux}_{[\,]}([\,]) & \Leftarrow & \\ & & \\ \text{app}(Z_1, Y, Z_2) & \Leftarrow & \text{aux}_{[H|T]}(Z_1), \text{aux}_{[H|Z]}(Z_2), \text{app}(T, Y, Z) \end{array}$$

The H_1-Approximation

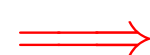
$$
\begin{array}{lll}
\text{app}(Z, Y, Y') & \Leftarrow & \text{aux}_{[\,]}(Z) \\
\text{aux}_{[\,]}([\,]) & \Leftarrow & \\
 & & \\
\text{app}(Z_1, Y, Z_2) & \Leftarrow & \text{aux}_{[H|T}(Z_1), \text{aux}_{[H|Z}(Z_2), \text{app}(T, Y, Z) \\
 & & \\
\text{aux}_{[H|T]}([H|T]) & \Leftarrow & \text{app}(T, Y, Z) \\
\text{aux}_{[H|Z]}([H|Z]) & \Leftarrow & \text{app}(T, Y, Z) \\
 & \ldots &
\end{array}
$$

... with the Result:

$$
\begin{array}{lcl}
\mathsf{val}_Z([Z_1|Z_2]) & \Leftarrow & \mathsf{top}(Z_1), \mathsf{aux}(Z_2) \\
\mathsf{val}_Z([Z_1|Z_2]) & \Leftarrow & \mathsf{top}(Z_1), \mathsf{top}(Z_2) \\
\mathsf{aux}([\,]) & \Leftarrow & \\
\mathsf{aux}([Z_1|Z_2]) & \Leftarrow & \mathsf{top}(Z_1), \mathsf{aux}(Z_2)
\end{array}
$$

... with the Result:

$$
\begin{array}{lll}
\text{val}_Z([Z_1|Z_2]) & \Leftarrow & \text{top}(Z_1), \text{aux}(Z_2) \\
\text{val}_Z([Z_1|Z_2]) & \Leftarrow & \text{top}(Z_1), \text{top}(Z_2) \\
\text{aux}([\,]) & \Leftarrow & \\
\text{aux}([Z_1|Z_2]) & \Leftarrow & \text{top}(Z_1), \text{aux}(Z_2)
\end{array}
$$



Listness and all information about the content of the result is lost :-(

Discussion

- One source of imprecision is the non-linear rule:

$$\mathsf{app}([\,], Y, Y) \quad \Leftarrow$$

Discussion

- One source of imprecision is the non-linear rule:

$$\text{app}([\,], Y, Y) \quad \Leftarrow$$

- Better results can be obtained if additionally call patterns are tracked !

$\Longrightarrow$ Magic Set Transformation

Magic Sets

- For every predicate p/k, we introduce a new predicate called_p/k with the clauses

$$\mathrm{called}_p(\underline{t}) \Leftarrow \quad \text{for the query} \quad \Leftarrow \mathrm{p}(\underline{t})$$

Magic Sets

- For every predicate p/k, we introduce a new predicate called_p/k with the clauses

$$\mathrm{called}_p(\underline{t}) \Leftarrow \quad \text{for the query} \quad \Leftarrow \mathrm{p}(\underline{t})$$

-

$$\begin{array}{rcl} \mathrm{called}_{p_i}(\underline{t_i}) & \Leftarrow & \mathrm{called}_p(\underline{t}), \mathrm{p}_1(\underline{t_1}), \ldots, \mathrm{p}_{i-1}(\underline{t_{i-1}}) \\ p_i(\underline{t}) & \Leftarrow & \mathrm{called}_p(\underline{t}), \mathrm{p}_1(\underline{t_1}), \ldots, \mathrm{p}_{i-1}(\underline{t_m}) \end{array}$$

for every clause:

$$\mathrm{p}(\underline{t}) \Leftarrow \mathrm{p}_1(\underline{t_1}), \ldots, \mathrm{p}_m(\underline{t_m})$$

Example: append (Cont.)

$$
\begin{array}{lcl}
\text{app}([\,], Y, Y) & \Leftarrow & \text{called}([\,], Y, Y) \\
\text{app}([H|T], Y, [H|Z]) & \Leftarrow & \text{called}([H|T], Y, [H|Z]), \\
 & & \text{app}(T, Y, Z) \\
\text{called}(T, Y, Z) & \Leftarrow & \text{called}([H|T], Y, [H|Z]) \\
\text{called}([1, 2], [3], Z) & \Leftarrow &
\end{array}
$$

The H_1-Approximation (Cont.)

$$
\begin{array}{lcl}
\text{val}_Z([Z_1|Z_2]) & \Leftarrow & \text{val}_2(Z_1), \text{aux}_1(Z_2) \\
\text{val}_Z([Z_1|Z_2]) & \Leftarrow & \text{val}_1(Z_1), \text{aux}_2(Z_2) \\
\text{aux}_1([Z_1|Z_2]) & \Leftarrow & \text{val}_3(Z_1), \text{val}_{[\,]}(Z_2) \\
\text{aux}_2([Z_1|Z_2]) & \Leftarrow & \text{val}_2(Z_1), \text{aux}_1(Z_2) \\
\text{aux}_2([Z_1|Z_2]) & \Leftarrow & \text{val}_1(Z_1), \text{aux}_2(Z_2) \\
\text{val}_i(i) & \Leftarrow & \\
\text{val}_{[\,]}([\,]) & \Leftarrow &
\end{array}
$$

Alternative: Set Constraints

Heintze/Jaffar 1990

- The analysis requires the set operators:

Projection	$\pi_{::,i}\, X$
Constructor application	$X :: Y$
Check	$(e \cap X \neq \emptyset);\, Y$
Intersection	$X \cap Y$

Alternative: Set Constraints

Heintze/Jaffar 1990

- The analysis requires the set operators:

Projection	$\pi_{::,i}\, X$
Constructor application	$X :: Y$
Check	$(e \cap X \neq \emptyset);\, Y$
Intersection	$X \cap Y$

- All these can be modelled by H_1-clauses :-)

Alternative: Set Constraints

Heintze/Jaffar 1990

- The analysis requires the set operators:

Projection	$\pi_{::,i}\, X$
Constructor application	$X :: Y$
Check	$(e \cap X \neq \emptyset); Y$
Intersection	$X \cap Y$

- All these can be modelled by H_1-clauses :-)
- Set constraints still are less precise ...

Precision

The H_1-clause:

$$p(a(X, Y)) \Leftarrow q(b(X, Y))$$

is approximated by the set constraint:

$$\text{val}_p \;\supseteq\; a(\pi_{b,1}\text{val}_q, \pi_{b,2}\text{val}_q)$$

Precision

The H_1-clause:

$$p(a(X,Y)) \Leftarrow q(b(X,Y))$$

is approximated by the set constraint:

$$\text{val}_p \supseteq a(\pi_{b,1}\text{val}_q, \pi_{b,2}\text{val}_q)$$

... which for

$$\text{val}_q = \{b(1,2), b(2,1)\}$$

returns:

$$\text{val}_p = \{a(1,1), a(1,2), a(2,1), a(2,2)\}$$

Summary

- Horn clauses provide a convenient tool for approximating the collecting semantics :-)

Summary

- Horn clauses provide a convenient tool for approximating the collecting semantics :-)

- The approximation of arbitrary Horn clauses with H_1-clauses may result in depressing results :-(

Summary

- Horn clauses provide a convenient tool for approximating the collecting semantics :-)
- The approximation of arbitrary Horn clauses with H_1-clauses may result in depressing results :-(
- Better results can be hoped for by taking the query into account via Magic Sets :-)

4. Concurrent

Cryprographic C

The Idea Goubault-Larrecq 2005

- Do not just analyze cryptographic protocols — but their implementation in a standard programming language !!!

The Idea Goubault-Larrecq 2005

- Do not just analyze cryptographic protocols — but their implementation in a standard programming language !!!
- For the protocol assume perfect cryptography which is supported by a datatype msg and language primitives.

The Idea — Goubault-Larrecq 2005

- Do not just analyze cryptographic protocols — but their implementation in a standard programming language !!!
- For the protocol assume perfect cryptography which is supported by a datatype msg and language primitives.
- Programs write and read from a public channel which is controlled by the attacker.

Example

$$
\begin{array}{l}
\ldots \\
\textbf{while}\ (\textbf{true})\ \ \{ \\
\qquad x_1 \leftarrow \mathsf{recv}(); \\
\qquad x_2 \leftarrow \mathsf{enc}(\mathsf{second}(x_1), \mathsf{pub}_A); \\
\qquad \textbf{if}\ (x_2 = \mathsf{hash}(\mathsf{first}(x_1)))\ \ \{ \\
\qquad\qquad x_3 \leftarrow \mathsf{enc}\,(\mathsf{code}, \mathsf{pub}_A); \\
\qquad\qquad \mathsf{send}(x_3); \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

Approach

- A predicate reach$/1$ checks reachability of program points u.
- A predicate $\text{val}_{u,x}/1$ collects all values of x when reaching u.
- Decryption is implemented through pattern matching.
- The capabilities of the attacker are modelled with Horn clauses ...

Some Constraints

$$
\begin{array}{rcl}
\text{known}(Z) & \Leftarrow & \text{chan}(Z) \\
\text{chan}(Z) & \Leftarrow & \text{known}(Z) \\
\text{val}_{x_1}(Z) & \Leftarrow & \text{reach}(1), \text{chan}(Z) \\
\text{val}_{x_2}(Z) & \Leftarrow & \text{reach}(2), \text{val}_{x_1}(\mathbf{pair}(_, \text{enc}(Z, \text{secr}_A))) \\
\text{reach}(2) & \Leftarrow & \text{reach}(1) \\
 & \ldots & \\
\text{reach}(3) & \Leftarrow & \text{reach}(2), \text{val}_{x_1}(\mathbf{pair}(Z_1, _)), \text{val}_{x_2}(\text{hash}(Z_1)) \\
 & \ldots &
\end{array}
$$

Discussion

- The Horn clauses are program specific.
- The generated clauses are not necessarily H_1

Discussion

- The Horn clauses are program specific.
- The generated clauses are not necessarily H_1

$\Longrightarrow$ further approximation is necessary !

Discussion

- The Horn clauses are program specific.
- The generated clauses are not necessarily H_1

$\Longrightarrow$ further approximation is necessary !

$\Longrightarrow$ precision may be gained through instantiation ...

Discussion

- The Horn clauses are program specific.
- The generated clauses are not necessarily H_1

$\Longrightarrow$ further approximation is necessary !

$\Longrightarrow$ precision may be gained through instantiation ...

$\Longrightarrow$ or magic sets :-)

Conclusion

- We have seen that Horn clauses are a decent formalism for the specification of program analyses.
- Topdown solving allowed for program independent specifications of dataflow problems.

Conclusion

- We have seen that Horn clauses are a decent formalism for the specification of program analyses.
- Topdown solving allowed for program independent specifications of dataflow problems.
- Instantiation allowed for for program independent specifications also for functional programs.

Conclusion

- We have seen that Horn clauses are a decent formalism for the specification of program analyses.
- Topdown solving allowed for program independent specifications of dataflow problems.
- Instantiation allowed for for program independent specifications also for functional programs.
- Magic sets enable decent results also for Prolog.

Conclusion

- We have seen that Horn clauses are a decent formalism for the specification of program analyses.
- Topdown solving allowed for program independent specifications of dataflow problems.
- Instantiation allowed for for program independent specifications also for functional programs.
- Magic sets enable decent results also for Prolog.
- The last two may also be useful for CCC.