Horn Clauses for Program Analysis

Helmut Seidl

Technische Universität München

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

Control-flow Graph $\Leftrightarrow$ Relation

\[
\begin{align*}
x &= 1; \\
x &< 10; \\
x &\geq 10; \\
y &= 0; \\
y &= y + x; \\
x &= x + 1;
\end{align*}
\]
Representation of Programs

Control-flow Graph $\iff$ Relation

Control-flow Graph:
- Initial state 0
- Transition $x = 1$ from 0 to 1
- Transition $y = 0$ from 1 to 2
- Transition $x \geq 10$ from 2 to 5
- Transition $x < 10$ from 2 to 3
- Transition $x = x + 1$ from 3 to 4
- Transition $y = y + x$ from 4 to 3
- Transition $y = y + x$ from 3 to 2
- Transition $y = y + x$ from 2 to 1
- Transition $x = x + 1$ from 1 to 0

Relation:
- $x = 1$
- $x < 10$
- $y = 0$
- $x \geq 10$
- $x = x + 1$
- $y = y + x$
Representation of Programs

Control-flow Graph $\iff$ Relation

\[
\begin{align*}
\text{var}(x) & \iff \\
\text{var}(y) & \iff \\
\text{edge}(0, \text{assign}(x, 1), 1) & \iff \\
\text{edge}(1, \text{assign}(y, 0), 2) & \iff \\
\text{edge}(2, \text{test}(\text{op}(\text{less}, x, 10)), 3) & \iff \\
\text{edge}(3, \text{assign}(y, \text{op}(\text{plus}, y, x)), 4) & \iff \\
\text{edge}(4, \text{assign}(x, \text{op}(\text{plus}, x, 1)), 2) & \iff \\
\text{edge}(2, \text{test}(\text{op}(\text{geq}, x, 10)), 5) & \iff \\
\end{align*}
\]
Clauses for Liveness

\[ \text{live}(U, X) \iff \text{edge}(U, \text{Lab}, V), \text{used}(X, \text{Lab}) \]

\[ \text{live}(U, X) \iff \text{edge}(U, \text{Lab}, V), \text{live}(V, X), \text{not}(\text{def}(X, \text{Lab})) \]
Clauses for Liveness

\[
\begin{align*}
\text{live}(U, X) & \iff \text{edge}(U, Lab, V), \text{used}(X, Lab) \\
\text{live}(U, X) & \iff \text{edge}(U, Lab, V), \text{live}(V, X), \neg \text{def}(X, Lab) \\
\text{def}(X, \text{assign}(X, _)) & \iff \\
\text{used}(X, \text{assign}(_, E)) & \iff \text{occurs}(X, E) \\
\text{used}(X, \text{test}(E)) & \iff \text{occurs}(X, E)
\end{align*}
\]
Clauses for Liveness

\[
\text{live}(U, X) \iff \text{edge}(U, Lab, V), \text{used}(X, Lab)
\]

\[
\text{live}(U, X) \iff \text{edge}(U, Lab, V), \text{live}(V, X), \neg \text{def}(X, Lab)
\]

\[
\text{def}(X, \text{assign}(X, \_)) \iff
\]

\[
\text{used}(X, \text{assign}(\_, E)) \iff \text{occurs}(X, E)
\]

\[
\text{used}(X, \text{test}(E)) \iff \text{occurs}(X, E)
\]

\[
\text{occurs}(X, X) \iff \text{var}(X)
\]

\[
\text{occurs}(X, \text{op}(\_, E)) \iff \text{occurs}(X, E)
\]

\[
\text{occurs}(X, \text{op}(\_, E1, E2)) \iff \text{occurs}(X, E1)
\]

\[
\text{occurs}(X, \text{op}(\_, E1, E2)) \iff \text{occurs}(X, E2)
\]
The formulation requires the predicate $\text{not}/1$.

$\Rightarrow$ stratified negation
Discussion

- The formulation requires the predicate \( \text{not} / 1 \)
  \[ \implies \text{stratified negation} \]

- The auxiliary predicate \( \text{occurs} / 2 \) is not finite
  \[ \implies \text{bottom-up tabulation will not terminate} ! \]
Discussion

- The formulation requires the predicate `not/1`
  
  $\implies$ stratified negation

- The auxiliary predicate `occurs/2` is not finite
  
  $\implies$ bottom-up tabulation will not terminate!

  $\implies$ Topdown solving
Result for Liveness

\[
\begin{align*}
\text{live}(1, x) & \iff \\
\text{live}(2, x) & \iff \\
\text{live}(3, x) & \iff \\
\text{live}(4, x) & \iff \\
\text{live}(2, y) & \iff \\
\text{live}(3, y) & \iff \\
\text{live}(4, y) & \iff 
\end{align*}
\]
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

- 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\footnote{Jones 1987}

- Use set constraints instead\footnote{Heintze 1994}

- Why not just Horn clauses?\footnote{???}
The append Function

let rec a = fun x → match x with
            [] → fun y → y
        h::t → fun y → h::a t y
in a [1; 2] [3]
The Horn Clauses

\[ \text{val}_a(\text{fun}(x, \text{match})) \iff \]
\[ \text{val}_{\text{match}}(\text{fun}(y, y)) \iff \text{val}_x([]) \]
\[ \text{val}_{\text{match}}(\text{fun}(y, \text{cons})) \iff \text{val}_x(- :: -) \]
\[ \text{val}_h(Z) \iff \text{val}_x(Z :: -) \]
\[ \text{val}_{\text{cons}}(H :: T) \iff \text{val}_h(H), \text{val}_{a \cdot t \cdot y}(T) \]
\[ \text{val}_{a \cdot t \cdot y}(Z) \iff \text{val}_{a \cdot t}(\text{fun}(x, \text{match})), \text{val}_{\text{match}}(Z) \]

...
Discussion

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

$\iff H_3$
Discussion

- All heads of the generated constraints are linear! No variable occurs repeatedly in bodies!!
  \[ \Rightarrow H_3 \]

- 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!!

  \[ \Rightarrow H_3 \]

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

- These can be found in **polynomial** time :-)

Helmut Seidl
Nov. 18/19, 2008
Cachan
The Result

\[ \text{val}_{\text{let}}(Z_1 :: Z_2) \iff \text{val}_3(Z_1), \text{val}[][Z_2] \]

\[ \text{val}_{\text{let}}(Z_1 :: Z_2) \iff \text{val}_h(Z_1), \text{val}_{\text{a.t.y}}(Z_2) \]

\[ \text{val}_{\text{a.t.y}}(Z_1 :: Z_2) \iff \text{val}_3(Z_1), \text{val}[][Z_2] \]

\[ \text{val}_{\text{a.t.y}}(Z_1 :: Z_2) \iff \text{val}_h(Z_1), \text{val}_{\text{a.t.y}}(Z_2) \]

\[ \text{val}_h(1) \iff \]

\[ \text{val}_h(2) \iff \]

\[ \text{val}_3(3) \iff \]

\[ \text{val}[][[]] \iff \]
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

\[ \begin{align*}
\text{val}_a & \supseteq \{ \text{fun}(x, \text{match}) \} \\
\text{val}_{\text{match}} & \supseteq (\{[]\} \cap \text{val}_x \neq \emptyset); \ \{ \text{fun}(y, y) \} \\
\text{val}_{\text{match}} & \supseteq (\_::\_ \cap \text{val}_x \neq \emptyset); \ \{ \text{fun}(y, \text{cons}) \} \\
\text{val}_h & \supseteq \pi::,1 \ \text{val}_x \\
\text{val}_{\text{cons}} & \supseteq \text{val}_h :: \text{val}_{a \cdot t \cdot y} \\
\text{val}_{a \cdot t \cdot y} & \supseteq (\{ \text{fun}(x, \text{match}) \} \cap \text{val}_{a \cdot t} \neq \emptyset); \ \text{val}_{\text{match}} \\
\ldots 
\end{align*} \]
The analysis requires the set operators:

- **Projection**  
  \[ \pi_{i \in I} X \]

- **Constructor application**  
  \[ X :: Y \]

- **Check**  
  \[ (e \cap X \neq \emptyset); Y \]
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

\[ \text{val}(A, Z) \iff \text{val}(E, Z) \]

\[ \ldots \]
Program-independent Formulation

\[
\text{val}(A, Z) \quad \iff \quad \text{val}(E, Z)
\]

\[
\ldots
\]

\[
\text{val}(\text{fun}(X, E), \text{fun}(X, E)) \quad \iff \quad \text{val}(H :: T, Z_1 :: Z_2)
\]

\[
\text{val}(H :: T, Z_1 :: Z_2) \quad \iff \quad \text{val}(H, Z_1), \text{val}(T, Z_2)
\]
Program-independent Formulation

\[
\text{reach(} \text{let}(a, \ldots, \text{app}(\text{app}(a, [1; 2]), [3])))) \iff \\
\text{reach}(E) \iff \text{reach(} \text{let}(_, E, _)) \\
\text{val}(A, Z) \iff \text{val}(E, Z) \\
\ldots \\
\text{val}(\text{fun}(X, E), \text{fun}(X, E)) \iff \\
\text{val}(H :: T, Z_1 :: Z_2) \iff \text{val}(H, Z_1), \text{val}(T, Z_2)
\]
Program-independent Formulation

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

\[ \text{reach}(E) \iff \text{reach}(\text{let}(\_, E, \_)) \]

\[ \text{val}(A, Z) \iff \text{reach}(\text{let}(A, E, \_)), \text{val}(E, Z) \]

\[ \ldots \]

\[ \text{val}(\text{fun}(X, E), \text{fun}(X, E)) \iff \text{reach}(\text{fun}(X, E)) \]

\[ \text{val}(H :: T, Z_1 :: Z_2) \iff \text{reach}(H :: T), \text{val}(H, Z_1), \text{val}(T, Z_2) \]
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

- 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
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:

$$p(X_1, \ldots, X_k) \iff \text{any} \quad \text{or} \quad p(a(X_1, \ldots, X_k)) \iff \text{any}$$
The Class of $H_1$-Clauses

$H_1$-Clauses are of the form:

\[ p(X_1, \ldots, X_k) \iff \text{any} \quad \text{or} \quad p(a(X_1, \ldots, X_k)) \iff \text{any} \]

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:

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

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 :-)

Helmut Seidl
Nov. 18/19, 2008
Cachan
Example: append

\[
\text{app}([], Y, Y) \leftarrow \\
\text{app}([H\mid T], Y, [H\mid Z]) \leftarrow \text{app}(T, Y, Z) \\
\text{app}([1, 2], [3], Z) \leftarrow
\]

... results in:
The $H_1$-Approximation

\[
\text{app}(Z, Y, Y') \iff \text{aux}_[](Z)
\]

\[
\text{aux}_[]([]) \iff 
\]
The $H_1$-Approximation

\[
\text{app}(Z, Y, Y') \iff \text{aux}_{[]} (Z)
\]

\[
\text{aux}_{[]} ([]) \iff
\]

\[
\text{app}(Z_1, Y, Z_2) \iff \text{aux}_{[H|T]} (Z_1), \text{aux}_{[H|Z]} (Z_2), \text{app}(T, Y, Z)
\]
The $H_1$-Approximation

\[
\text{app}(Z, Y, Y') \iff \text{aux}_Z(Z)
\]
\[
\text{aux}_Z([[]]) \iff
\]
\[
\text{app}(Z_1, Y, Z_2) \iff \text{aux}_{H|T}(Z_1), \text{aux}_{H|Z}(Z_2), \text{app}(T, Y, Z)
\]
\[
\text{aux}_{H|T}([H|T]) \iff \text{app}(T, Y, Z)
\]
\[
\text{aux}_{H|Z}([H|Z]) \iff \text{app}(T, Y, Z)
\]
\[
\ldots
\]
... with the Result:

\[
\begin{align*}
\text{val}_Z([Z_1|Z_2]) & \iff \text{top}(Z_1), \text{aux}(Z_2) \\
\text{val}_Z([Z_1|Z_2]) & \iff \text{top}(Z_1), \text{top}(Z_2) \\
\text{aux}([\,]) & \iff \\
\text{aux}([Z_1|Z_2]) & \iff \text{top}(Z_1), \text{aux}(Z_2)
\end{align*}
\]
... with the Result:

\[
\begin{align*}
\text{val}_Z([Z_1|Z_2]) &\iff \text{top}(Z_1), \text{aux}(Z_2) \\
\text{val}_Z([Z_1|Z_2]) &\iff \text{top}(Z_1), \text{top}(Z_2) \\
\text{aux}([\ ])&\iff \\
\text{aux}([Z_1|Z_2]) &\iff \text{top}(Z_1), \text{aux}(Z_2)
\end{align*}
\]

\[
\Rightarrow \quad \text{Listness and all information about the content of the result is lost } \ :-(\]

Helmut Seidl
Nov. 18/19, 2008
Cachan
Discussion

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

\[
\text{app}([], Y, Y) \leftarrow
\]
Discussion

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

  \[ \text{app}([], Y, Y) \leq \]

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

  ⏫ Magic Set Transformation
Magic Sets

- For every predicate $p/k$, we introduce a new predicate $\text{called}_p/k$ with the clauses

$$\text{called}_p(t) \iff \text{for the query} \iff p(t)$$
Magic Sets

- For every predicate $p/k$, we introduce a new predicate $\text{called}_p/k$ with the clauses

$$\text{called}_p(t) \iff \text{for the query} \iff p(t)$$

- for every clause:

$$p(t) \iff p_1(t_1), \ldots, p_m(t_m)$$
Example:  **append**  (Cont.)

\[
\begin{align*}
\text{app}([\ ], Y, Y) & \iff \text{called}([\ ], Y, Y) \\
\text{app}([H|T], Y, [H|Z]) & \iff \text{called}([H|T], Y, [H|Z]), \text{app}(T, Y, Z) \\
\text{called}(T, Y, Z) & \iff \text{called}([H|T], Y, [H|Z]) \\
\text{called}([1, 2], [3], Z) & \iff
\end{align*}
\]
The $H_1$-Approximation (Cont.)

\[ \text{val}_Z([Z_1|Z_2]) \iff \text{val}_2(Z_1), \text{aux}_1(Z_2) \]

\[ \text{val}_Z([Z_1|Z_2]) \iff \text{val}_1(Z_1), \text{aux}_2(Z_2) \]

\[ \text{aux}_1([Z_1|Z_2]) \iff \text{val}_3(Z_1), \text{val}_{[\ ]}(Z_2) \]

\[ \text{aux}_2([Z_1|Z_2]) \iff \text{val}_2(Z_1), \text{aux}_1(Z_2) \]

\[ \text{aux}_2([Z_1|Z_2]) \iff \text{val}_1(Z_1), \text{aux}_2(Z_2) \]

\[ \text{val}_i(i) \iff \]

\[ \text{val}_{[\ ]}([\ ]\ ]\ ] ) \iff \]
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: $\pi_{i \cdot} X$
- 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:

\[ p(a(X, Y)) \Leftrightarrow q(b(X, Y)) \]

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:

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

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 Cryptographic 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

\[
\ldots
\textbf{while} \ (\text{true}) \ 
\begin{align*}
\ x_1 & \leftarrow \text{recv}(); \\
\ x_2 & \leftarrow \text{enc} (\text{second}(x_1), \text{pub}_A); \\
\textbf{if} \ (x_2 = \text{hash} (\text{first}(x_1))) \ 
\begin{align*}
\ x_3 & \leftarrow \text{enc} (\text{code}, \text{pub}_A); \\
\text{send}(x_3); \\
\end{align*}
\end{align*}
\]
Approach

- A predicate `reach/1` checks reachability of program points `u`.
- A predicate `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

\[
\begin{align*}
\text{known}(Z) & \iff \text{chan}(Z) \\
\text{chan}(Z) & \iff \text{known}(Z) \\
\text{val}_{x_1}(Z) & \iff \text{reach}(1), \text{chan}(Z) \\
\text{val}_{x_2}(Z) & \iff \text{reach}(2), \text{val}_{x_1}(\text{pair}(Z, \text{secr}_A)) \\
\text{reach}(2) & \iff \text{reach}(1) \\
& \text{...} \\
\text{reach}(3) & \iff \text{reach}(2), \text{val}_{x_1}(\text{pair}(Z_1, Z)), \text{val}_{x_2}(\text{hash}(Z_1)) \\
& \text{...}
\end{align*}
\]
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 further approximation is necessary!
Discussion

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

  $\Rightarrow$ further approximation is necessary!

  $\Rightarrow$ precision may be gained through instantiation ...
Discussion

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

  $\Rightarrow$ further approximation is necessary!

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

  $\Rightarrow$ or magic sets :-)

Helmut Seidl Nov. 18/19, 2008 Cachan
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 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.