

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

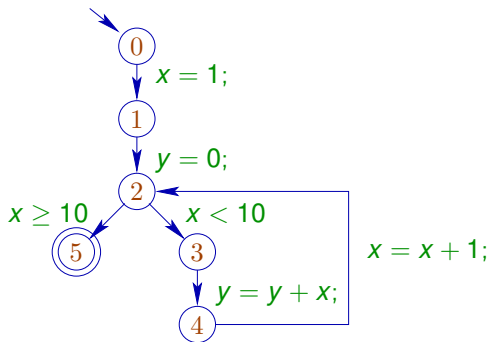
Datalog	...
Tabeled Resolution	...
Ganzinger/McAllester	2001
Nielson/Riis-Nielson/S.	2002

Representation of Programs

Control-flow Graph $\Leftrightarrow$ Relation

Representation of Programs

Control-flow Graph $\Leftrightarrow$ Relation



Representation of Programs

Control-flow Graph $\Leftrightarrow$ Relation

<code>var(<i>x</i>)</code>	$\Leftarrow$
<code>var(<i>y</i>)</code>	$\Leftarrow$
<code>edge(0, <i>assign</i>(<i>x</i>, 1), 1)</code>	$\Leftarrow$
<code>edge(1, <i>assign</i>(<i>y</i>, 0), 2)</code>	$\Leftarrow$
<code>edge(2, <i>test</i>(<i>op</i>(<i>less</i>, <i>x</i>, 10)), 3)</code>	$\Leftarrow$
<code>edge(3, <i>assign</i>(<i>y</i>, <i>op</i>(<i>plus</i>, <i>y</i>, <i>x</i>)), 4)</code>	$\Leftarrow$
<code>edge(4, <i>assign</i>(<i>x</i>, <i>op</i>(<i>plus</i>, <i>x</i>, 1)), 2)</code>	$\Leftarrow$
<code>edge(2, <i>test</i>(<i>op</i>(<i>geq</i>, <i>x</i>, 10)), 5)</code>	$\Leftarrow$

Clauses for Liveness

$\text{live}(U, X) \Leftarrow \text{edge}(U, \text{Lab}, V), \text{used}(X, \text{Lab})$

$\text{live}(U, X) \Leftarrow \text{edge}(U, \text{Lab}, V), \text{live}(V, X),$
 $\text{not}(\text{def}(X, \text{Lab}))$

Clauses for Liveness

$\text{live}(U, X) \Leftarrow \text{edge}(U, \text{Lab}, V), \text{used}(X, \text{Lab})$

$\text{live}(U, X) \Leftarrow \text{edge}(U, \text{Lab}, V), \text{live}(V, X),$
 $\text{not}(\text{def}(X, \text{Lab}))$

$\text{def}(X, \text{assign}(X, _)) \Leftarrow$

$\text{used}(X, \text{assign}(_, E)) \Leftarrow \text{occurs}(X, E)$

$\text{used}(X, \text{test}(E)) \Leftarrow \text{occurs}(X, E)$

Clauses for Liveness

$\text{live}(U, X) \Leftarrow \text{edge}(U, \text{Lab}, V), \text{used}(X, \text{Lab})$

$\text{live}(U, X) \Leftarrow \text{edge}(U, \text{Lab}, V), \text{live}(V, X),$
 $\text{not}(\text{def}(X, \text{Lab}))$

$\text{def}(X, \text{assign}(X, _)) \Leftarrow$

$\text{used}(X, \text{assign}(_, E)) \Leftarrow \text{occurs}(X, E)$

$\text{used}(X, \text{test}(E)) \Leftarrow \text{occurs}(X, E)$

$\text{occurs}(X, X) \Leftarrow \text{var}(X)$

$\text{occurs}(X, \text{op}(_, E)) \Leftarrow \text{occurs}(X, E)$

$\text{occurs}(X, \text{op}(_, E1, E2)) \Leftarrow \text{occurs}(X, E1)$

$\text{occurs}(X, \text{op}(_, E1, E2)) \Leftarrow \text{occurs}(X, E2)$

Discussion

- ▶ The formulation requires the predicate **not/1**
 stratified negation

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 !

Discussion

- ▶ The formulation requires the predicate **not**/1
⇒ stratified negation
- ▶ The auxiliary predicate **occurs**/2 is not finite
⇒ bottom-up tabulation will not terminate !
⇒ Topdown solving

Result for Liveness

live(1, x) $\Leftarrow$

live(2, x) $\Leftarrow$

live(3, x) $\Leftarrow$

live(4, x) $\Leftarrow$

live(2, y) $\Leftarrow$

live(3, y) $\Leftarrow$

live(4, y) $\Leftarrow$

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

```
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

$\text{val}_a(\text{fun}(x, \text{match})) \Leftarrow$

$\text{val}_{\text{match}}(\text{fun}(y, y)) \Leftarrow \text{val}_x([])$

$\text{val}_{\text{match}}(\text{fun}(y, \text{cons})) \Leftarrow \text{val}_x(- :: -)$

$\text{val}_h(Z) \Leftarrow \text{val}_x(Z :: -)$

$\text{val}_{\text{cons}}(H :: T) \Leftarrow \text{val}_h(H), \text{val}_{a.t.y}(T)$

$\text{val}_{a.t.y}(Z) \Leftarrow \text{val}_{a.t}(\text{fun}(x, \text{match})), \text{val}_{\text{match}}(Z)$

...

Discussion

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

$\Rightarrow H_3$

Discussion

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

$\implies H_3$

- ▶ 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 !!

$\implies H_3$

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

The Result

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

$$\text{val}_{\text{let}}(Z_1 :: Z_2) \quad \Leftarrow \quad \text{val}_h(Z_1), \text{val}_{\text{a.t.y}}(Z_2)$$

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

$$\text{val}_{\text{a.t.y}}(Z_1 :: Z_2) \quad \Leftarrow \quad \text{val}_h(Z_1), \text{val}_{\text{a.t.y}}(Z_2)$$

$$\text{val}_h(1) \quad \Leftarrow$$

$$\text{val}_h(2) \quad \Leftarrow$$

$$\text{val}_3(3) \quad \Leftarrow$$

$$\text{val}_{[]} ([]) \quad \Leftarrow$$

Discussion

- ▶ The analysis finds:

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

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

Discussion

- ▶ The analysis finds:

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

- ▶ 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

$$\text{val}_a \supseteq \{\text{fun}(x, \text{match})\}$$

$$\text{val}_{\text{match}} \supseteq (\{[]\} \cap \text{val}_x \neq \emptyset); \{\text{fun}(y, y)\}$$

$$\text{val}_{\text{match}} \supseteq (\text{--} :: \text{--} \cap \text{val}_x \neq \emptyset); \{\text{fun}(y, \text{cons})\}$$

$$\text{val}_h \supseteq \pi_{::,1} \text{val}_x$$

$$\text{val}_{\text{cons}} \supseteq \text{val}_h :: \text{val}_{\text{a.t.y}}$$

$$\text{val}_{\text{a.t.y}} \supseteq (\{\text{fun}(x, \text{match})\} \cap \text{val}_{\text{a.t}} \neq \emptyset); \text{val}_{\text{match}}$$

...

Discussion

- The analysis requires the set operators:

Projection $\pi_{::,j} X$

Constructor application $X :: Y$

Check $(e \cap X \neq \emptyset); Y$

Discussion

- ▶ The analysis requires the set operators:

Projection $\pi_{::,j} X$

Constructor application $X :: Y$

Check $(e \cap X \neq \emptyset); Y$

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

Program-independent Formulation

$\text{val}(A, Z)$

$\Leftarrow$

$\text{val}(E, Z)$

...

Program-independent Formulation

$\text{val}(A, Z) \quad \Leftarrow \quad \text{val}(E, Z)$

$\dots$

$\text{val}(\text{fun}(X, E), \text{fun}(X, E)) \quad \Leftarrow$

$\text{val}(H :: T, Z_1 :: Z_2) \quad \Leftarrow \quad \text{val}(H, Z_1), \text{val}(T, Z_2)$

Program-independent Formulation

reach(**let**(*a*, ..., **app**(**app**(*a*, [1; 2]), [3]))) $\Leftarrow$

reach(*E*) $\Leftarrow$ **reach**(**let**(-, *E*, -))

val(*A*, *Z*) $\Leftarrow$
val(*E*, *Z*)

...

val(**fun**(*X*, *E*), **fun**(*X*, *E*)) $\Leftarrow$

val(*H* :: *T*, *Z*₁ :: *Z*₂) $\Leftarrow$
val(*H*, *Z*₁), **val**(*T*, *Z*₂)

Program-independent Formulation

reach(**let**(*a*, ..., **app**(**app**(*a*, [1; 2]), [3]))) $\Leftarrow$

reach(*E*) $\Leftarrow$ **reach**(**let**(-, *E*, -))

val(*A*, *Z*) $\Leftarrow$ **reach**(**let**(*A*, *E*, -)),
val(*E*, *Z*)

...

val(**fun**(*X*, *E*), **fun**(*X*, *E*)) $\Leftarrow$ **reach**(**fun**(*X*, *E*))

val(*H* :: *T*, *Z*₁ :: *Z*₂) $\Leftarrow$ **reach**(*H* :: *T*),
val(*H*, *Z*₁), **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 :-(
 H_1
- ▶ Finiteness analysis of the predicate $\text{reach}/1$ 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:

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

The Class of H_1 -Clauses

H_1 -Clauses are of the form:

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

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:

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

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 :-)

Example: append

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

... results in:

The H_1 -Approximation

$$\text{app}(Z, Y, Y') \quad \Leftarrow \quad \text{aux}_{[]} (Z)$$
$$\text{aux}_{[]} ([]) \quad \Leftarrow$$

The H_1 -Approximation

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

$$\text{aux}_{[]} ([]) \Leftarrow$$

$$\text{app}(Z_1, Y, Z_2) \Leftarrow \text{aux}_{[H|T]} (Z_1), \text{aux}_{[H|Z]} (Z_2), \text{app}(T, Y, Z)$$

The H_1 -Approximation

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

$$\text{aux}_{[]} ([]) \Leftarrow$$

$$\text{app}(Z_1, Y, Z_2) \Leftarrow \text{aux}_{[H|T]} (Z_1), \text{aux}_{[H|Z]} (Z_2), \text{app}(T, Y, Z)$$

$$\text{aux}_{[H|T]} ([H|T]) \Leftarrow \text{app}(T, Y, Z)$$

$$\text{aux}_{[H|Z]} ([H|Z]) \Leftarrow \text{app}(T, Y, Z)$$

...

... with the Result:

$$\text{val}_Z([Z_1|Z_2]) \Leftarrow \text{top}(Z_1), \text{aux}(Z_2)$$
$$\text{val}_Z([Z_1|Z_2]) \Leftarrow \text{top}(Z_1), \text{top}(Z_2)$$
$$\text{aux}([]) \Leftarrow$$
$$\text{aux}([Z_1|Z_2]) \Leftarrow \text{top}(Z_1), \text{aux}(Z_2)$$

... with the Result:

$$\text{val}_Z([Z_1|Z_2]) \Leftarrow \text{top}(Z_1), \text{aux}(Z_2)$$
$$\text{val}_Z([Z_1|Z_2]) \Leftarrow \text{top}(Z_1), \text{top}(Z_2)$$
$$\text{aux}([]) \Leftarrow$$
$$\text{aux}([Z_1|Z_2]) \Leftarrow \text{top}(Z_1), \text{aux}(Z_2)$$

$\Rightarrow$ Listness and all information about the content of the result is lost :-)

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:

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

- ▶ 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 called_p/k with the clauses

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

Magic Sets

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

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



$$\text{called}_{p_i}(\underline{t_i}) \Leftarrow \text{called}_p(\underline{t}), p_1(\underline{t_1}), \dots, p_{i-1}(\underline{t_{i-1}})$$

$$p_i(\underline{t}) \Leftarrow \text{called}_p(\underline{t}), p_1(\underline{t_1}), \dots, p_{i-1}(\underline{t_m})$$

for every clause:

$$p(\underline{t}) \Leftarrow p_1(\underline{t_1}), \dots, p_m(\underline{t_m})$$

Example: append (Cont.)

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

The H_1 -Approximation (Cont.)

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

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

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

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

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

$$\text{val}_i(i) \Leftarrow$$

$$\text{val}_[]([]) \Leftarrow$$

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 $\pi_{::,j} X$

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 $\pi_{::,j} X$

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 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

```
...  
while (true) {  
     $x_1 \leftarrow \text{recv}();$   
     $x_2 \leftarrow \text{enc}(\text{second}(x_1), \text{pub}_A);$   
    if ( $x_2 = \text{hash}(\text{first}(x_1))$ ) {  
         $x_3 \leftarrow \text{enc}(\text{code}, \text{pub}_A);$   
         $\text{send}(x_3);$   
    }  
}
```

Approach

- ▶ A predicate $\text{reach}/1$ 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

$\text{known}(Z) \Leftarrow \text{chan}(Z)$

$\text{chan}(Z) \Leftarrow \text{known}(Z)$

$\text{val}_{x_1}(Z) \Leftarrow \text{reach}(1), \text{chan}(Z)$

$\text{val}_{x_2}(Z) \Leftarrow \text{reach}(2), \text{val}_{x_1}(\text{pair}(-, \text{enc}(Z, \text{secr}_A)))$

$\text{reach}(2) \Leftarrow \text{reach}(1)$

...

$\text{reach}(3) \Leftarrow \text{reach}(2), \text{val}_{x_1}(\text{pair}(Z_1, -)), \text{val}_{x_2}(\text{hash}(Z_1))$

...

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
 $\implies$ 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
 - ⇒ further approximation is necessary !
 - ⇒ precision may be gained through instantiation ...
 - ⇒ 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.