Transforming Constrained Horn Clauses for Program Verification

Maurizio Proietti (IASI-CNR, Rome, Italy)

Joint work with Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi

Overview

1. Rule-based transformations: Fold/Unfold transformations, CHC specialization, predicate tupling
2. Generating verification conditions via CHC specialization
3. CHC specialization as CHC solving
4. Increasing the power of CHC solving via predicate tupling

Constrained Horn Clauses (CHCs)

- First order formulas of the form:

  $$A_1 \wedge \ldots \wedge A_n \wedge c \rightarrow A_0$$

  where $A_0, A_1, \ldots, A_n$ are atomic formulas and c is a formula in a theory *Th* of constraints (any first-order theory). All variables are assumed to be universally quantified in front.

- Prolog-like syntax:

  $$A_0 \text{ :- } c, A_1, \ldots, A_n.$$

CHCs for Program Verification

Program *sumupto*: summing the first n integers

x=0; y=0; while (x<n) { x=x+1; y=x+y}

Specification
{n≥1} *SumUpto* {y≥x}

Translation ⟹

Constrained Horn Clauses

false :- N≥1, X=0, Y=0, p(X, Y, N). %Init
p(X, Y, N) :- X<N, X1=X+1, Y1=X+Y, p(X1, Y1, N). %Loop
p(X, Y, N) :- X≥ N, Y<X. %Exit

- The program satisfies the specification iff the set of CHCs is satisfiable.
- Satisfiability of CHCs is undecidable: no ultimate verifier exists.
- Two ways for improving the effectiveness of CHC-based verification:

  1. Designing smart heuristics for satisfiability solvers;
  2. Transforming difficult CHC problems into equisatisfiable, easy ones.

  (1 and 2 not mutually exclusive)

1. Rule-based Transformations

Transformations of Functional and Logic Programs

Main idea of this talk: Transformation techniques introduced for improving functional and logic programs [Burstall-Darlington 1977, Tamaki-Sato 1984] can be adapted to ease satisfiability proofs for CHCs.

Initial program	$P_0 \rightarrow P_1 \rightarrow \ldots \rightarrow P_n$	Final program

where '$\rightarrow$' is an application of a transformation rule.

- Each rule application preserves the semantics:
  $M(P_0) = M(P_1) = \ldots = M(P_n)$
- The application of the rules is guided by a strategy that guarantees that P_n is more efficient than P_0.

Transformation Rules for CHCs

Initial clauses $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ Final clauses

where '$\rightarrow$' is an application of a transformation rule.

Transformation Rules for CHCs

Initial clauses $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ Final clauses

where '$\rightarrow$' is an application of a transformation rule.

R1. Definition. Introduce a new predicate definition

introduce C: newp(X) :- c, G

$S_{i+1} = S_i \cup \{C\}$ Defs := Defs $\cup$ {C}

Transformation Rules for CHCs

Initial clauses $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ Final clauses

where '$\rightarrow$' is an application of a transformation rule.

R1. Definition. Introduce a new predicate definition

C: newp(X) :- c, G

$S_{i+1} = S_i \cup \{C\}$ Defs := Defs $\cup$ {C}

R2. Unfolding. Apply a Resolution step

given C: H :- c,A,G A :- d_1,G_1 ... A :- d_m,G_m in S_i

derive S_i = { H :- c,d_1,G_1,G ... H :- c,d_m,G_m,G }

$S_{i+1} = (S_i - \{C\}) \cup S$

… Transformation rules for CHCs

R3. Folding. Replace a conjunction by a new predicate

given C: H :- d,B,G in S_i newp(X) :- c,B. with $d \rightarrow c$ in Defs

derive D: H :- d,newp(X),G.

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

… Transformation rules for CHCs

R3. Folding. Replace a conjunction by a new predicate

given $\quad$ C: H :- d,B,G $\quad$ in S_i $\quad$ newp(X) :- c,B. with $d \rightarrow c$ in Defs

derive $\quad$ D: H :- d,newp(X),G.

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R4. Constraint replacement. Replace a constraint by an equivalent one

given $\quad$ C: H :- c,B,G $\quad$ in S_i $\quad$ with $Th \vDash c \leftrightarrow d$

derive $\quad$ D: H :- d,B,G

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

… Transformation rules for CHCs

R3. Folding. Replace a conjunction by a new predicate

given C: H :- d,B,G in S_i newp(X) :- c,B. with $d \rightarrow c$ in Defs

derive D: H :- d,newp(X),G.

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R4. Constraint replacement. Replace a constraint by an equivalent one

given C: H :- c,B,G in S_i with $Th \vDash c \leftrightarrow d$

derive D: H :- d,B,G

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R5. Clause Removal. Remove a clause C with unsatisfiable constraint or subsumed by another

$S_{i+1} = (S_i - \{C\})$

… Transformation rules for CHCs

R3. Folding. Replace a conjunction by a new predicate

given C: H :- d,B,G in S_i newp(X) :- c,B. with $d \rightarrow c$ in Defs

derive D: H :- d,newp(X),G.

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R4. Constraint replacement. Replace a constraint by an equivalent one

given C: H :- c,B,G in S_i with $Th \vDash c \leftrightarrow d$

derive D: H :- d,B,G

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R5. Clause Removal. Remove a clause C with unsatisfiable constraint or subsumed by another

$S_{i+1} = (S_i - \{C\})$

Theorem [Tamaki-Sato 84,Etalle-Gabbrielli 96]: If every new definition is unfolded at least once in $S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_n$ then S_0 satisfiable iff S_n satisfiable.

Transformation Strategies

- Transformation rules need to be guided by suitable strategies.
- Main idea: exploit some knowledge about the query to produce a customized, easier to verify set of clauses.
- Specialization [Gallagher,Leuschel,FPP,…]: Given a set of clauses S and a query **false :- c,A**, where **A** is atomic, transform S into a set of clauses S_{SP} such that

  $$S \cup \{\text{false :- c,A}\} \text{ satisfiable iff } S_{SP} \cup \{\text{false :- c,A}\} \text{ satisfiable.}$$

- Predicate Tupling (also known as Conjunctive Partial Deduction) [PP, Leuschel,…]: Given a set of clauses S and a query **false :- c,G**, where **G** is a (non-atomic) conjunction, introduce a new predicate **newp(X) :- G** and transform set of clauses S_T such that

  $$S \cup \{\text{false :- c,G}\} \text{ satisfiable iff } S_T \cup \{\text{false :- c,newp(X)}\} \text{ satisfiable.}$$

Specialization Strategy: An Example

false :- X<0, p(X,b). % $\forall$X. p(X,b) $\rightarrow$ X$\geq$0 S_0

p(X,C) :- X=Y+1, p(Y,C).

p(X,a).

p(X,b) :- X$\geq$0, tm_halts(X). % the X-th Turing machine halts on X

Specialization Strategy: An Example

false :- X<0, p(X,b). % $\forall$X. p(X,b) $\rightarrow$ X$\geq$0 S_0

p(X,C) :- X=Y+1, p(Y,C).

p(X,a).

p(X,b) :- X$\geq$0, tm_halts(X). % the X-th Turing machine halts on X

Define: q(X) :- X<0, p(X,b). % q(X) is a specialization of p(X,C) S_1

% to a specific constraint on X and value of C

Specialization Strategy: An Example

```
false :- X<0, p(X,b).                  % ∀X. p(X,b) → X≥0                  S0
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X≥0, tm_halts(X).           % the X-th Turing machine halts on X
```

Define: q(X) :- X<0, p(X,b). % q(X) is a specialization of p(X,C) S_1

% to a specific constraint on X and value of C

Unfold: q(X) :- X<0, X=Y+1, p(Y,b). S_2

q(X) :- X<0, X≥0, tm_halts(X).

Specialization Strategy: An Example

false :- X<0, p(X,b). % $\forall$X. p(X,b) $\rightarrow$ X$\geq$0 S_0
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X$\geq$0, tm_halts(X). % the X-th Turing machine halts on X

Define: q(X) :- X<0, p(X,b). % q(X) is a specialization of p(X,C) S_1
% to a specific constraint on X and value of C

Unfold: q(X) :- X<0, X=Y+1, p(Y,b). S_2
~~q(X) :- X<0, X≥0, tm_halts(X)~~. % clause removal

Specialization Strategy: An Example

```
false :- X<0, p(X,b).
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X≥0, tm_halts(X).
```

% $\forall$X. p(X,b) $\rightarrow$ X$\geq$0 — S_0

% the X-th Turing machine halts on X

Define: q(X) :- X<0, p(X,b). % q(X) is a specialization of p(X,C) — S_1

% to a specific constraint on X and value of C

Unfold: q(X) :- X<0, X=Y+1, p(Y,b). — S_2

~~q(X) :- X<0, X≥0, tm_halts(X)~~. % clause removal

Fold: false :- X<0, q(X).

q(X) :- X<0, X=Y+1, q(Y). — S_3

Satisfiability of S_3 is easy to check: q(X) $\equiv$ *false* makes all clauses *true* (no facts for q)

Tupling Strategy: An Example

asum(A,I,N,S) : "the sum of the elements of array A from index I to N is S"

amax(A,I,N,M) : "the largest element of array A from index I to N is M"

S_0

false :- S<M, 0≤I<N, asum(A,I,N,S), amax(A,I,N,M).
asum(A,I,N,S) :- I=N, S=0.
asum(A,I,N,S) :- 0≤I<N, I1=I+1, read(A,I,X), X≥0, S=S1+X, asum(A,I1,N,S1).
amax(A,I,N,M) :- I=N, M=0.
amax(A,I,N,M) :- 0≤I<N, I1=I+1, read(A,I,X), X≥0,
((X≥M1, M=X) ∨ (X<M1, M=M1)), amax(A,I1,N,M1).

Proof of satisfiability:

$$\left.\begin{array}{l} \mathrm{asum}(A,I,N,S) \rightarrow \forall K,Y.(I \leq K < N,\ \mathrm{read}(A,K,Y) \rightarrow S \geq Y) \\ \mathrm{amax}(A,I,N,M) \rightarrow \exists K.(I \leq K < N,\ \mathrm{read}(A,K,M)) \end{array}\right\} \rightarrow S \geq M$$

The proof uses quantified array constraints.
Eldarica (with the SimpleArray(1) theory) does not solve these clauses.

Tupling Strategy: An Example

false :- S<M, 0≤I<N, asum(A,I,N,S), amax(A,I,N,M). S_0

Tupling Strategy: An Example

false :- S<M, 0≤I<N, asum(A,I,N,S), amax(A,I,N,M). S_0

Define: asummax (A,I,N,S,M) :- asum(A,I,N,S), amax(A,I,N,M). S_1

Tupling Strategy: An Example

false :- S<M, 0≤I<N, asum(A,I,N,S), amax(A,I,N,M). S_0

Define: asummax (A,I,N,S,M) :- asum(A,I,N,S), amax(A,I,N,M). S_1

Unfold: asummax(A,I,N,S,M) :- I=N-1, S=0, M=0. S_2
& CR asummax(A,I,N,S,M) :- 0=<I, I<N-1, X≥0, I1=I+1, S=S1+X, read(A,I,X),
((X≥M1, M=X) ∨ (X<M1, M=M1)),
asum(A,I1,N,S1), amax(A,I1,N,M1).

Tupling Strategy: An Example

false :- S<M, 0≤I<N, asum(A,I,N,S), amax(A,I,N,M). S_0

Define: asummax (A,I,N,S,M) :- asum(A,I,N,S), amax(A,I,N,M). S_1

Unfold: asummax(A,I,N,S,M) :- I=N-1, S=0, M=0. S_2
& CR asummax(A,I,N,S,M) :- 0=<I, I<N-1, X≥0, I1=I+1, S=S1+X, read(A,I,X),
((X≥M1, M=X) ∨ (X<M1, M=M1)),
asum(A,I1,N,S1), amax(A,I1,N,M1).

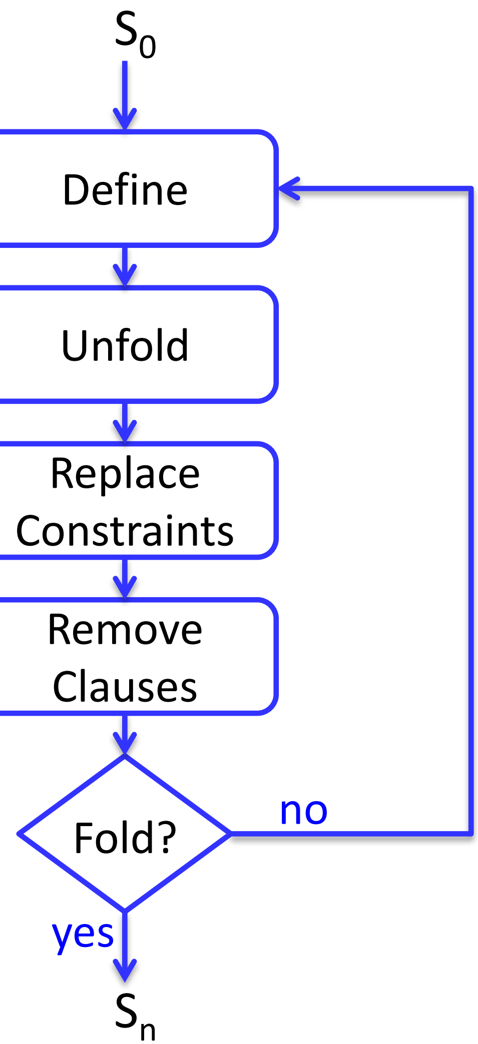
Fold: asummax(A,I,N,S,M) :- I=N-1, S=0, M=0. S_3
asummax(A,I,N,S,M) :- 0=<I, I<N-1, X≥0, I1=I+1, S=S1+X, read(A,I,X),
((X≥M1, M=X) ∨ (X<M1, M=M1)), asummax(A,I1,N,S1,M1).
false :- S<M, 0≤I<N, asummax(A,I,N,S,M).

Proof of satisfiability: asummax(A,I,N,S,M) → S≥M

The proof only uses linear arithmetic constraints.

Eldarica (with the SimpleArray(1) theory) solves these clauses.

A Generic U/F Transformation Strategy

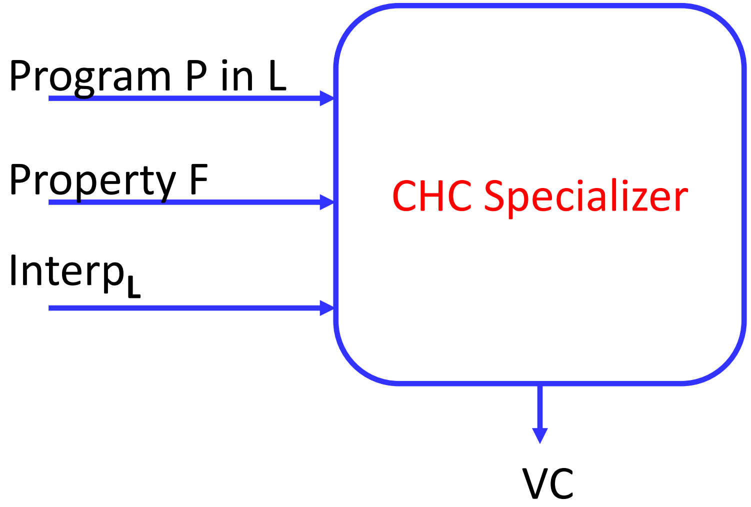


Some Issues About the U/F Strategy

- Unfolding: Which atoms should be unfolded? When to stop?
- Constraint replacement: A suitable constraint reasoner is needed
- Definition: Suitable new predicates need to be introduced to guarantee termination and effectiveness of strategy

2. Generating Verification Conditions via CHC Specialization

CHC Specialization as a Verification Condition Generator



L: Programming language

Interp$_L$: CHC interpreter for L

VC: Verification Conditions, i.e., a set of CHCs independent of L

F holds for P iff VC is satisfiable

The CHC specializer is parametric with respect to the programming language L and the class of properties.

Translating Imperative Programs into CHC

- **C**-like imperative language with assignments, conditionals, jumps. While-loops translated to conditionals and jumps.
- Commands encoded as atomic assertions: at(Label, Cmd).

```
x=0;
y=0;
while (x<n) {
  x=x+1;
  y=x+y
}
```

```
0.  x=0;
1.  y=0;
2.  if (x<n) 3 else 6;
3.  x=x+1;
4.  y=x+y;
5.  goto 2;
6.  halt
```

```
at(0,asgn(int(x), int(0))).
at(1,asgn(int(y), int(0))).
at(2, ite(less(int(x), int(n)), 3, 6)).
at(3, asgn(int(x), plus(int(x), int(1)))).
at(4, asgn(int(y), plus(int(x), int(y)))).
at(5, goto(2)).
at(6, halt).
```

A Small-Step Operational Semantics

- The operational semantics is a one-step transition relation between configurations

  <n1:cmd1, env1> $\Rightarrow$ <n2:cmd2, env2>

  where: n:cmd is a labelled command and env is an environment mapping variable identifiers to values;

- Assignment

  <n: x=e, env> $\Rightarrow$ <next(n), update(env, x, **[**e**]**env)>

  where: next(n) is the next labelled command and update(env, x, **[**e**]**env) updates the value of x in env to the value of expression e in env;

- Conditional

  <n: if (e) n1 else n2, env> $\Rightarrow$ <next(n1), env> if **[**e**]**$\delta\neq 0$
  <n: if (e) n1 else n2, env> $\Rightarrow$ <next(n2), env> if **[**e**]**$\delta= 0$

- Jump

  <n: goto n1, env> $\Rightarrow$ <next(n1), env>

A CHC Interpreter for the Small-Step Semantics

- Configurations: cf(LC, Env)
  where:
  - LC is a labelled command represented by a term of the form cmd(L,C),
    L is a label, C is a command
  - Env is an environment represented as a list of (variable-id,value) pairs:
    [(x,X),(y,Y),(z,Z)]

- One-step transition relation between configurations:
  tr(cf(LC1,Env1), cf(LC2,Env2))

CHC Interpreter (Asgn)

assignment   x=e;

```
     source configuration                target configuration
tr( cf(cmd(L, asgn(X,E)), Env1), cf(cmd(L1, C), Env2) ) :-
    nextlab(L,L1),                        % next label
    at(L1,C),                             % next command
    eval(E,Env1,V),                       % evaluate expression
    update(Env1,X,V,Env2).                % update environment
```

More clauses for predicate tr to encode the semantics of the other commands.

Encoding Partial Correctness Properties

- Partial correctness specification (Hoare triple):

  $\{\varphi\}$ *prog* $\{\psi\}$

  If the initial values of the program variables satisfy the precondition φ and *prog* terminates, **then** the final values of the program variables satisfy the postcondition ψ.

- CHC encoding of partial correctness:

false :- initConf(Cf), reach(Cf).		*PC-prop*
reach(Cf1) :- tr(Cf1,Cf2), reach(Cf2).	PC property	
reach(Cf) :- errorConf(Cf).		
initConf(cf(C, Env)) :- at(0,C), φ(Env).	Initial configuration	
errorConf(cf(C, Env)) :- at(h,C), $\neg\psi$(Env).	Error configuration	
tr(cf1,cf2) :- …	Interp_L	

- $\{\varphi\}$ *prog* $\{\psi\}$ is valid iff *PC-prop* is satisfiable.

VCGen: Generating Verification Conditions

- *VCGen* is a transformation strategy that specializes *PC-prop* to a given $\{\varphi\}$ *prog* $\{\psi\}$, and removes explicit reference to the interpreter (function **cf**, predicates **at**, **tr**, etc.).

- All new definitions are of the form newp(X) :- reach(cf), corresponding to a program point.

- Limited reasoning about constraints at specialization time (satisfiability only).

- VCGen is parametric wrt $\mathrm{Interp}_{\mathbf{L}}$ (to a large extent).

- If $\textit{PC-prop} \xrightarrow{\textit{VCGen}} \textit{VC}$ then *PC-prop* is satisfiable iff *VC* is satisfiable

Generating Verification Conditions: An Example

PC property: {n≥1} *SumUpto* {y>x}

CHC encoding:

PC-prop

```
false :- initConf(Cf), reach(Cf).
reach(Cf1) :- tr(Cf1,Cf2), reach(Cf2).
reach(Cf) :- errorConf(Cf).
initConf(cf(C, [(x,X),(y,Y),(n,N)])) :- at(0,C), N≥1.
errorConf(cf(C, [(x,X),(y,Y),(n,N)])) :- at(6,C), Y≤X.
tr(cf1,cf2) :- …
…
at(0,asgn(int(x), int(0))).
…
```

VCGen

Verification Conditions:

VC

```
false :- N≥1, X=0, Y=0, p(X, Y, N).
p(X, Y, N) :- X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N).
p(X, Y, N) :- X≥ N, Y<X.
```

Experimental evaluation

- Other semantics: multi-step for recursive functions, exceptions, etc.
- Checking the satisfiability of the VCs using QARMC, Z3 (PDR), MathSAT (IC3), Eldarica
- VCGen+QARMC compares favorably to HSF+QARMC

	Small-step (SS_f^s)				Multi-step (MS)				HSF(C)
	QARMC	Z3	MSAT	ELD	QARMC	Z3	MSAT	ELD	
Correct answers	217	208	205	217	210	196	177	182	189
safe problems	161	150	158	158	160	144	147	141	158
unsafe problems	56	58	47	59	50	52	30	41	31
Incorrect answers	5	0	3	2	3	0	1	0	12
false alarms	3	0	1	0	1	0	1	0	3
missed bugs	2	0	2	2	2	0	0	0	9
Timeouts	98	112	112	101	120	124	142	138	119
Total problems	320	320	320	320	320	320	320	320	320
VCG time	221.68	221.68	221.68	221.68	141.85	141.85	141.85	141.85	
Solving time	3656.24	4221.39	2988.86	8809.58	2674.00	2704.95	1896.96	2779.18	
Total time	3877.92	4443.07	3210.54	9031.26	2815.85	2846.80	2038.81	2921.03	
Average Time	17.87	21.36	15.66	41.62	13.41	14.52	11.52	16.05	

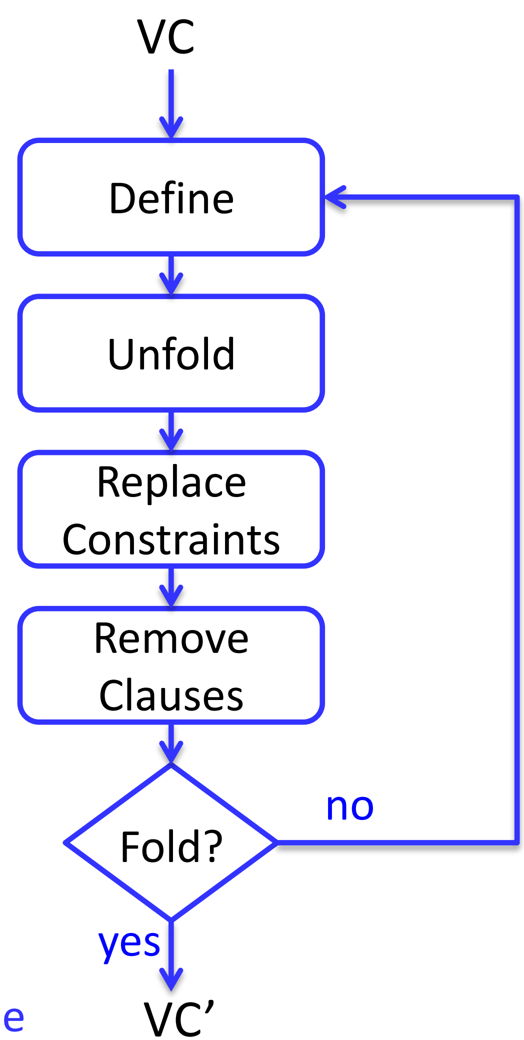
3. CHC Specialization as CHC Solving

VCTransf: Specializing Verification Conditions

false :- c, p(X)

newp(X) :- c, p(X)

apply theory of constraints



Specializing verification conditions by propagating constraints.

Introduction of new predicates by generalization (e.g., widening and convex hull techniques)

VC is satisfiable iff *VC'* is satisfiable

VCTransf as CHC Solving

The effect of applying VCTransf can be:

1. A set VC’ of verification conditions without constrained facts for the predicates on which the queries depend (i.e., no clauses of the form p(X) :- c).
   VC’ is satisfiable.

2. A set VC’ of verification conditions including false :- true.
   VC’ is unsatisfiable.

3. Neither 1 nor 2 (constrained facts of the form p(X) :- c, but not false :- true).
   Satisfiability is unknown.

propagation of constraint X<0 and constant b

VC
```
false :- X<0, p(X,b).
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X≥0, tm_halts(X).
```

VCTransf ⟹

VC’
```
false :- X<0, q(X).
q(X) :- X<0, X=Y+1, q(Y).
```

No constrained facts: VC’ satisfiable

Iterated CHC Specialization

- If the satisfiability of VC' is unknown, *VCTransf* can be iterated.

- Between two applications of *VCTransf* we can apply the *Reversal* transformation (particular case of the query-answer transformation [KafleGallagher 15] for linear programs) that interchanges premises and conclusions of clauses (backward reasoning from queries simulates forward reasoning from facts).

VC		*VC'*
false :- a(X), p(X). p(X) :- c(X,Y), p(Y). p(X) :- b(X).	$\xrightarrow{\textit{Reversal}}$	p(X) :- a(X). p(Y) :- c(X,Y), p(X). false :- b(X), p(X).

VC is satisfiable iff *VC'* is satisfiable

$$VC_0 \xrightarrow{\textit{VCTransf}} VC_1 \xrightarrow{\textit{Reversal}} VC_2 \xrightarrow{\textit{VCTransf}} VC_3 \quad \dots \quad \xrightarrow{\textit{VCTransf}} VC_n$$

Iterated CHC Specialization: *SumUpto* Example

false :- N≥1, X=0, Y=0, p(X, Y, N). *VC_0*
p(X, Y, N) :- X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N).
p(X, Y, N) :- X≥N, Y<X.

Iterated CHC Specialization: *SumUpto* Example

false :- N≥1, X=0, Y=0, p(X, Y, N). *VC_0*
p(X, Y, N) :- X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N).
p(X, Y, N) :- X≥N, Y<X.

VCTransf

false :- N≥1, X1=1, Y1=1, new2(X1, Y1, N). *VC_1*
new2(X, Y, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new3(X1, Y1, N).
new3(X, Y, N) :- X1≥1, Y1≥X1, X<N, X1=X+1, Y1=X1+Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y≥1, N≥1, X≥N, Y<X.

Iterated CHC Specialization: *SumUpto* Example

VC_0

false :- N≥1, X=0, Y=0, p(X, Y, N).
p(X, Y, N) :- X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N).
p(X, Y, N) :- X≥N, Y<X.

↓ *VCTransf*

VC_1

false :- N≥1, X1=1, Y1=1, new2(X1, Y1, N).
new2(X, Y, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new3(X1, Y1, N).
new3(X, Y, N) :- X1≥1, Y1≥X1, X<N, X1=X+1, Y1=X1+Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y≥1, N≥1, X≥N, Y<X.

↓ *Reversal*

VC_2

new2(X1, Y1, N) :- N≥1, X1=1, Y1=1.
new3(X1, Y1, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new2(X, Y, N).
new3(X1, Y1, N) :- X1≥1, Y1≥X1, X<N, X1=X+1, Y1=X1+Y, new3(X, Y, N).
false :- N≥1, Y≥1, X≥N, Y<X, new3(X, Y, N).

Iterated CHC Specialization: *SumUpto* Example

```
false :- N≥1, X=0, Y=0, p(X, Y, N).
p(X, Y, N) :- X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N).
p(X, Y, N) :- X≥N, Y<X.
```
VC_0

VCTransf

```
false :- N≥1, X1=1, Y1=1, new2(X1, Y1, N).
new2(X, Y, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new3(X1, Y1, N).
new3(X, Y, N) :- X1≥1, Y1≥X1, X<N, X1=X+1, Y1=X1+Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y≥1, N≥1, X≥N, Y<X.
```
VC_1

Reversal

```
new2(X1, Y1, N) :- N≥1, X1=1, Y1=1.
new3(X1, Y1, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new2(X, Y, N).
new3(X1, Y1, N) :- X1≥1, Y1≥X1, X<N, X1=X+1, Y1=X1+Y, new3(X, Y, N).
false :- N≥1, Y≥1, X≥N, Y<X, new3(X, Y, N).
```
VC_2

VCTransf

```
false :- N≥1, Y≥1, X≥N, Y<X, new4(X, Y, N).
```
VC_3

No constrained facts. *VC_3 is satisfiable*

Experiments with VeriMAP

216 examples taken from: DAGGER, TRACER, InvGen, and TACAS 2013 Software Verification Competition.

- ARMC [Podelski, Rybalchenko PADL 2007]
- HSF(C) [Grebenshchikov et al. TACAS 2012]
- TRACER [Jaffar, Murali, Navas, Santosa CAV 2012]

		VeriMAP (Gen_{PH})	ARMC	HSF(C)	TRACER	
					SPost	*WPre*
1	*correct answers*	185	138	159	91	103
2	safe problems	154	112	137	74	85
3	unsafe problems	31	26	22	17	18
4	*incorrect answers*	0	9	5	13	14
5	false alarms	0	8	3	13	14
6	missed bugs	0	1	2	0	0
7	*errors*	0	18	0	20	22
8	*timed-out problems*	31	51	52	92	77
9	*total score*	339 (0)	210 (-40)	268 (-28)	113 (-52)	132 (-56)
10	*total time*	10717.34	15788.21	15770.33	27757.46	23259.19
11	*average time*	57.93	114.41	99.18	305.03	225.82

Table 1: Verification results using VeriMAP, ARMC, HSF(C) and TRACER. For each column the sum of the values of lines 1, 4, 7, and 8 is 216, which is the total number of the verification problems we have considered. The timeout limit is five minutes. Times are in seconds.

4. Improving CHC Solving via Predicate Tupling

Limitations of Linear Arithmetic Specifications

- Not very expressive.
- Example: computing Fibonacci numbers

fibonacci: while (n>0) {
t=u;
u=u+v;
v=t;
n=n-1
}

{n=N, N>=0, u=1, v=0, t=0} *fibonacci* {fib(N,u)}

- The postcondition of *fibonacci* cannot be specified by using linear arithmetic constraints *only*.

Recursive Horn Clause Specifications

- Recursive Horn specifications:
  {z1=P1,…,zs=Ps, **pre**(P1, . . . , Ps)} *prog* {**post**(P1,…,Ps,z)}

  where :

  - z1,...,zs are global variables of *prog*;
  - P1,…,Ps are parameters;
  - z is a variable in {z1,...,zs};
  - **pre** and **post** are defined by a set of CHCs;
  - **post** is a functional relation: z=F(P1,…,Ps) for some function F defined for all P1, . . . , Ps that satisfy **pre**.

- All computable functions on integers can be specified by sets of CHCs.

Recursive Horn Specification for Fibonacci

Fibonacci specification:

{n=N, N>=0, u=1, v=0, t=0} *fibonacci* {fib(N,u)}

where:

Fib:

fib(0,F) :- F=1.

fib(1,F) :- F=1.

fib(N3,F3) :- N1>=0, N2=N1+1, N3=N2+1, F3=F1+F2, fib(N1,F1), fib(N2,F2).

Translating Partial Correctness into CHCs

A recursive Horn specification can be translated into CHCs in two steps:

Step1. Translate the operational semantics into CHCs;

Step 2. Generate verification conditions as a set of CHCs.

Translating the Operational Semantics

- Define a relation **fibonacci_prog**(N,U) such that, for all integers N, if the program variables satisfy the precondition

  n=N, N>=0, u=1, v=0, t=0

  then the final value of u computed by program *fibonacci* is U.

- **fibonacci_prog** is defined by a set *OpSem* of clauses that encode the operational semantics:

  fibonacci_prog(N,U) :- initConf(Cf0,N), reach(Cf0,Cfh), finalConf(Cfh,U).
  initConf(cf(LC, [(n,N),(u,U),(v,V),(t,T)]), N) :- N>=0, U=1, V=0, T=0, at(0,LC).
  finalConf(cf(LC, [(n,N),(u,U),(v,V),(t,T)]), U) :- at(last,LC).
  reach(Cf,Cf).
  reach(Cf0,Cf2) :- tr(Cf0,Cf1), reach(Cf1,Cf2).

  tr(Cf0,Cf1) is the interpreter of the imperative language.

Specializing the Operational Semantics

- Apply VCGen and specialize *OpSem* w.r.t. the program *fibonacci*.

- $OpSem_{SP}$:

```
fibonacci_prog(N,F) :-                                  %Initialization
        N>=0, U=1, V=0, T=0,
        r(N,U,V,T,  N1,F,V1,T1).
r(N,U,V,T, N2,U2,V2,T2) :-                              %Loop
        N>=1, N1=N-1, U1=U+V, V1=U, T1=U,
        r(N1,U1,V1,T1, N2,U2,V2,T2).
r(N,U,V,T,  N,U,V,T) :- N=<0.                           %Loop exit
```

- For every query false :- G,
  OpSem ∪ {false :- G} is satisfiable *iff* $OpSem_{SP}$ ∪ {false :- G} is satisfiable.

- $OpSem_{SP}$ is linear recursive (at most one predicate in the premise).

Generating Verification Conditions (Nonlinear recursive)

Q: false :- F1≠F2, fibonacci_prog(N,F1), fib(N,F2).

plus clauses for fibonacci_prog (*OpSem*$_{SP}$) and fib (*Fib*).

Program *fibonacci* is partially correct iff *OpSem*$_{sp}$ ∪ *Fib* ∪ {Q}.

Q in not linear recursive; the clauses in *Fib* are not linear recursive.

Generating Verification Conditions (Almost linear recursive)

- Under suitable assumptions, linear recursive clauses, except for queries.

- Transform each clause **post**(P1, . . . ,Ps, Z) :- B defining the postcondition, into a query for **prog**:

  (1) Replace **post** by **prog** in the head and in the body

  prog(P1, . . . ,Ps, Z) :- B′

  (2) Move the conclusion to the premise (exploiting functionality of **prog**):

  false :- Y≠Z, **prog**(P1, . . . ,Ps, Y), B′   where Y is a new variable

  (3) Case split:

  false :- Y>Z, **prog**(X1, . . . ,Xs, Z), B′

  false :- Y<Z, **prog**(X1, . . . ,Xs, Z), B′

- *If* for all generated queries false :- G, $OpSem_{SP}$ ∪ {false :- G} is satisfiable, *then* {z1=P1,..., zs=Ps, **pre**(P1, . . . , Ps)} *prog* {**post**(P1,..., Ps,z)} is valid.

Verification Conditions for Fibonacci

Generating the verification conditions for *fibonacci*

fib(0,F) :- F=1.

(1) Replace **fib** by **fibonacci_prog** in the head and in the body

fibonacci_prog(0,F) :- F=1.

(2) Move the conclusion to the premise:

false :- F≠1, fibonacci_prog(0,F).

(3) Case split

Q1: false :- F>1, fibonacci_prog(0,F).

Q2: false :- F<1, fibonacci_prog(0,F).

Verification Conditions for Fibonacci

- Verification conditions for *fibonacci*

```
Q1:  false :- F>1, fibonacci_prog(0,F).
Q2:  false :- F<1, fibonacci_prog(0,F).
Q3:  false :- F>1, fibonacci_prog(1,F).
Q4:  false :- F<1, fibonacci_prog(1,F).
Q5:  false :- N1>=0, N2=N1+1, N3=N2+1, F3>F1+F2,
              fibonacci_prog(N1,F1),
              fibonacci_prog(N2,F2),
              fibonacci_prog(N3,F3).
Q6:  false :- N1>=0, N2=N1+1, N3=N2+1, F3<F1+F2,
              fibonacci_prog(N1,F1),
              fibonacci_prog(N2,F2),
              fibonacci_prog(N3,F3).
```

- Program *fibonacci* is partially correct if, for i=1,...,6, $OpSem_{SP} \cup \{Qi\}$ is satisfiable.

Satisfiability via LA-Solvability

- Consider constraints $\mathbf{C}_{\mathbf{LA}}$ in **L**inear (Integer) **A**rithmetics (linear equalities and inequalities over the integers). An LA-solution of a set S of CHCs is a mapping

  Σ: Atom $\rightarrow \mathbf{C}_{\mathbf{LA}}$

  such that, for every clause A_0 :- c, $A_1,\ldots,A_n$ in S,

  $$\mathbf{LA} \models \forall(c, \Sigma(A_1) \wedge \ldots \wedge \Sigma(A_n) \rightarrow \Sigma(A_0))$$

- A set of CHCs is LA-solvable if it has an LA-solution.
- LA-solvability implies satisfiability, but not vice versa.
- Satisfiability is undecidable and not semidecidable.
  LA-solvability is semidecidable. ($\mathbf{C}_{\mathbf{LA}}$ is r.e. and entailment is decidable.)
  Complete LA-solving methods exist (e.g., CEGAR).

Limitations of LA-solving

- Program *fibonacci* is partially correct and each *OpSem* ∪ {Qi} is satisfiable.

- However, there is no LA-solution for *OpSem* ∪ {Q5} (nor for *OpSem* ∪ {Q6}).

  Proof (see details in ICLP-15 paper): there exists no LA constraint c(N,F) which is an LA-solution of the clauses for **r_fibonacci** and:

  $$\mathbf{LA} \models \forall(N1>=0,\ N2=N1+1,\ N3=N2+1,\ F3>F1+F2,\ c(N1,F1),\ c(N2,F2),\ c(N3,F3) \rightarrow \text{false})$$

- LA-solvers cannot prove the partial correctness of *fibonacci.*

Improving LA-solving by Transforming Verification Conditions

- Solution 1: More powerful constraint theories, but decidability of entailment is lost for non-linear polynomials [Matijasevic 70].

- Solution 2: Transform the verification conditions into equisatisfiable CHCs that are (sometimes) LA-solvable.

- Transformation: Linearization via predicate tupling.

Linearization

Q5: false :- N1>=0, N2=N1+1, N3=N2+1, F3>F1+F2,
fibonacci_prog(N1,F1), fibonacci_prog(N2,F2), fibonacci_prog(N3,F3).

- No LA-solution of single **fibonacci_prog** atoms is able to prove that the premise of Q5 is false.

- An "LA-solution" for the conjunction of the three **fibonacci_prog** atoms exists. The conjunction of the three atoms implies the LA-constraint:

  N1>=0, N2=N1+1, N3=N2+1, F3=F1+F2

  which implies satisfiability of Q5.

- Apply predicate tupling and transform conjunctions of atoms into single atoms, i.e., transform $OpSem_{SP} \cup$ {Q5} into a set of linear recursive clauses.

The Linearization Transformation

Nonlinear queries → **Nonlinear clauses**

The Linearization Transformation

Nonlinear clauses

$$H \text{ :- } c,\ p_1(X_1), \ldots, p_k(X_k).$$

The Linearization Transformation

Nonlinear clauses

$$\downarrow$$

H :- c, $p_1(X_1), \ldots, p_k(X_k)$.

Unfold using *$OpSem_{RI}$*

. . .

H :- d, $q_1(X_1), \ldots, q_j(X_j)$.

H :- e, $r_1(X_1), \ldots, r_m(X_m)$.

The Linearization Transformation

Nonlinear clauses

H :- c, $p_1(X_1), \ldots, p_k(X_k)$.

Unfold using *$OpSem_{RI}$*

H :- d, $q_1(X_1), \ldots, q_j(X_j)$. . . . **H :- e, $r_1(X_1), \ldots, r_m(X_m)$.**

Define:

newq($X_1, \ldots, X_j$) :-
$q_1(X_1), \ldots, q_j(X_j)$.

Define:

newr($X_1, \ldots, X_m$) :-
$r_1(X_1), \ldots, r_m(X_m)$.

The Linearization Transformation

Nonlinear clauses

$H \text{ :- } c,\ p_1(X_1), \ldots, p_k(X_k).$

Unfold using $OpSem_{RI}$

$H \text{ :- } d,\ q_1(X_1), \ldots, q_j(X_j).$ . . . $H \text{ :- } e,\ r_1(X_1), \ldots, r_m(X_m).$

Define:

$newq(X_1, \ldots, X_j) \text{ :- } q_1(X_1), \ldots, q_j(X_j).$

Define:

$newr(X_1, \ldots, X_m) \text{ :- } r_1(X_1), \ldots, r_m(X_m).$

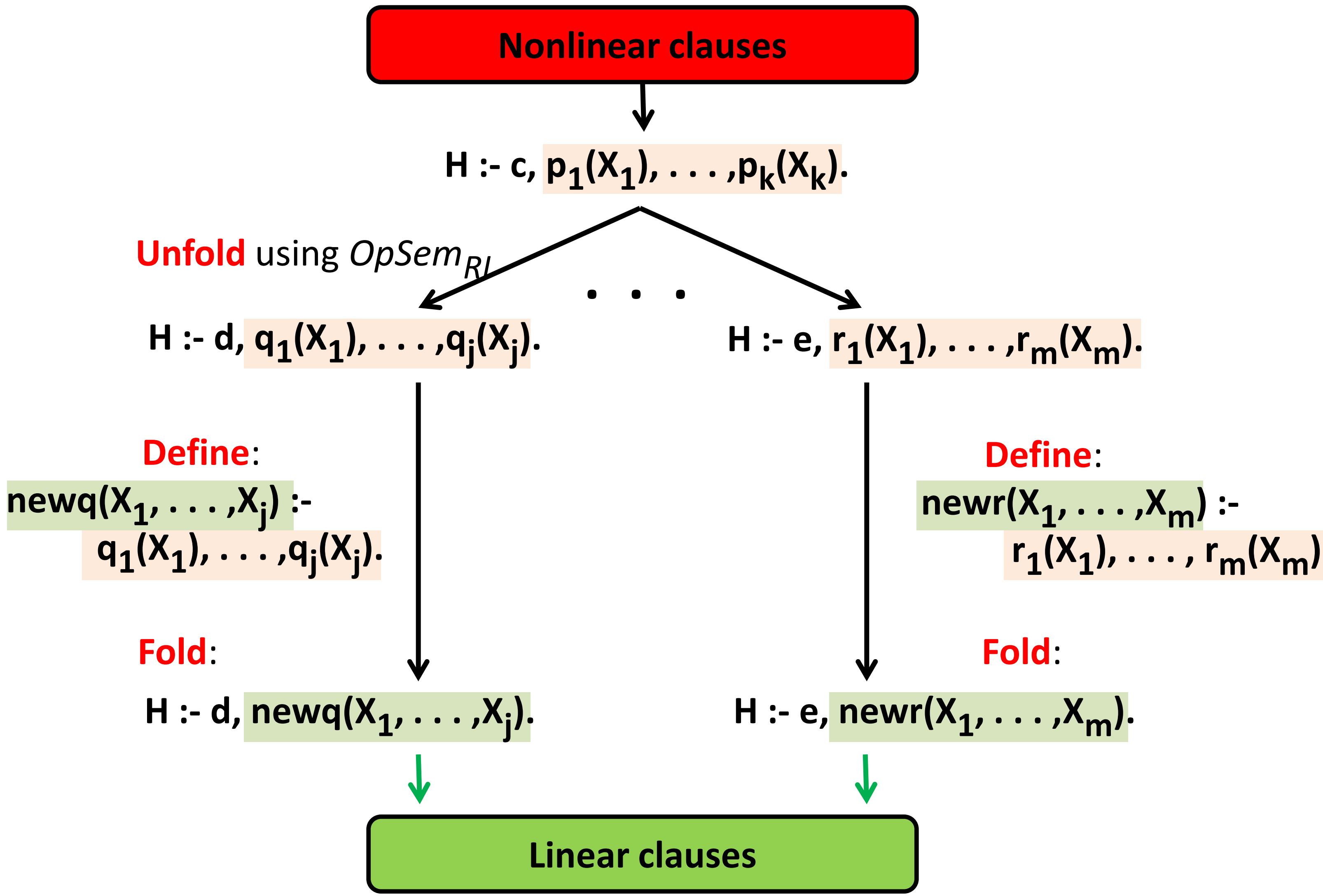
Fold:

$H \text{ :- } d,\ newq(X_1, \ldots, X_j).$

Fold:

$H \text{ :- } e,\ newr(X_1, \ldots, X_m).$

The Linearization Transformation



The Linearization Transformation

Nonlinear clauses

H :- c, $p_1(X_1), \ldots, p_k(X_k)$.

Unfold using $OpSem_{RI}$

H :- d, $q_1(X_1), \ldots, q_j(X_j)$.

. . .

H :- e, $r_1(X_1), \ldots, r_m(X_m)$.

Define:

$newq(X_1, \ldots, X_j)$:- $q_1(X_1), \ldots, q_j(X_j)$.

Define:

$newr(X_1, \ldots, X_m)$:- $r_1(X_1), \ldots, r_m(X_m)$.

Fold:

H :- d, $newq(X_1, \ldots, X_j)$.

Fold:

H :- e, $newr(X_1, \ldots, X_m)$.

Linear clauses

Linearized Fibonacci

false :- N1>= 0, N2=N1+1, N3=N2+1, F3>F1+F2, U=1, V=0, **new1**(N3,U,V,F3,N2,F2,N1,F1).

new1(N1,U,V,U,N2,U,N3,U) :- N1=<0, N2=<0, N3=<0.

new1(N1,U,V,U,N2,U,N3,F3) :- N1=<0, N2=<0, N4=N3-1, W=U+V, N3>=1, **new2**(N4,W,U,F3).

new1(N1,U,V,U,N2,F2,N3,U) :- N1=<0, N4=N2-1, W=U+V, N2>=1, N3=<0, **new2**(N4,W,U,F2).

new1(N1,U,V,U,N2,F2,N3,F3) :- N1=<0, N4=N2-1, N2>=1, N5=N3-1, N3>=1, **new3**(N4,W,U,F2,N5,F3).

new1(N1,U,V,F1,N2,U,N3,U) :- N4=N1-1, W=U+V, N1>=1, N2=<0, N3=<0, **new2**(N4,W,U,F1).

new1(N1,U,V,F1,N2,U,N3,F3) :- N4=N1-1, N1>=1, N2=<0, N5=N3-1, W=U+V, N3>=1, **new3**(N4,W,U,F1,N5,F3).

new1(N1,U,V,F1,N2,F2,N3,U) :- N4=N1-1, N1>=1, N5=N2-1, W=U+V, N2>=1, N3=<0, **new3**(N4,W,U,F1,N5,F2).

new1(N1,U,V,F1,N2,F2,N3,F3) :- N4=N1-1, N1>=1, N5=N2-1, N2>=1, N6=N3-1, W=U+V, N3>=1,
 new1(N4,W,U,F1,N5,F2,N6,F3).

plus linear clauses for **new2** and **new3**.

new1, **new2**, **new3** have been introduced by the following **definitions**:

new1(N1,U,V,F1,N2,F2,N3,F3) :- r(N1,U,V,V,X1,F1,Y1,Z1), r(N2,U,V,V,X2,F2,Y2,Z2), r(N3,U,V,V,X3,F3,Y3,Z3).

new2(N,U,V,F) :- r(N,U,V,V,X,F,Y,Z).

new3(N2,U,V,F2,N1,F1) :- r(N1,U,V,V,X1,F1,Y1,Z1), r(N2,U,V,V,X2,F2,Y2,Z2).

The linearized clauses for *fibonacci* are LA-solvable.

Properties of Linearization

- $OpSem_{SP}$ is a set of linear recursive clauses if no recursive functions in the imperative language.

- For every query **false :- G**, linearization terminates for the input

  $OpSem_{SP} \cup \{$**false :- G**$\}$

- Let *TransfCls* be the output of linearization.
  $OpSem_{SP} \cup \{$**false :- G**$\}$ is satisfiable iff *TransfCls* is satisfiable.

- If $OpSem_{SP} \cup \{$**false :- G**$\}$ is LA-solvable, then *TransfCls* is LA-solvable.
  Not vice versa: LA-solvability can be increased.

Experiments

Program	VCG	*LA*-solving-1		LIN	*LA*-solving-2: VeriMAP &		
		Z3	Eldarica		Z3	MathSAT	Eldarica
1. *binary_division*	0.02	4.16	*TO*	0.04	17.36	17.87	20.98
2. *fast_multiplication_2*	0.02	*TO*	3.71	0.01	1.07	1.97	7.59
3. *fast_multiplication_3*	0.03	*TO*	4.56	0.02	2.59	2.54	9.31
4. *fibonacci*	0.01	*TO*	*TO*	0.01	2.00	47.74	6.97
5. *Dijkstra_fusc*	0.01	1.02	3.80	0.05	2.14	2.80	10.26
6. *greatest_common_divisor*	0.01	*TO*	*TO*	0.01	0.89	1.78	0.04
7. *integer_division*	0.01	*TO*	*TO*	0.01	0.88	1.90	2.86
8. 91-*function*	0.01	1.27	*TO*	0.06	117.97	14.24	*TO*
9. *integer_multiplication*	0.02	*TO*	*TO*	0.01	0.52	14.76	0.54
10. *remainder*	0.01	*TO*	*TO*	0.01	0.87	1.70	3.16
11. *sum_first_integers*	0.01	*TO*	*TO*	0.01	1.79	2.30	6.81
12. *lucas*	0.01	*TO*	*TO*	0.01	2.04	8.39	9.46
13. *padovan*	0.01	*TO*	*TO*	0.01	2.24	*TO*	11.62
14. *perrin*	0.01	*TO*	*TO*	0.02	2.23	*TO*	11.89
15. *hanoi*	0.01	*TO*	*TO*	0.01	1.81	2.07	6.59
16. *digits10*	0.01	*TO*	*TO*	0.01	4.52	3.10	6.54
17. *digits10-itmd*	0.06	*TO*	*TO*	0.04	*TO*	10.26	12.38
18. *digits10-opt*	0.08	*TO*	*TO*	0.10	*TO*	*TO*	15.80
19. *digits10-opt100*	0.01	*TO*	*TO*	0.02	*TO*	58.99	8.98

Conclusions

- CHC transformations are useful for CHC satisfiability
  - For generating verification conditions from the program semantics
  - For proving the satisfiability of CHCs
  - For pre-processing the input of CHC solvers:
    CHC solving < transformation + CHC solving
- Future work:
  - Characterization of the power of fold/unfold: How much improvement?
  - Other applications (more languages, properties, etc.)
  - Integration with CHC solvers