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:

\[ A_1 \land \ldots \land A_n \land c \rightarrow A_0 \]

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:

\[ A_0 : - \ c, A_1, \ldots, A_n. \]
CHCs for Program Verification

Program *sumupto*: summing the first $n$ integers

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

**Specifiication**

\{\text{\{n\geq 1\}} \text{SumUpto } \{y\geq x\}\}

**Translation**

\[
\begin{align*}
\text{false} :& \quad N \geq 1, \ X=0, \ Y=0, \ p(X, Y, N). & \%\text{Init} \\
p(X, Y, N) :& \quad X<N, \ X1=X+1, \ Y1=X+Y, \ p(X1, Y1, N). & \%\text{Loop} \\
p(X, Y, N) :& \quad X \geq N, \ Y<X. & \%\text{Exit}
\end{align*}
\]

- 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.

- 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 $S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_n$ Final clauses

where $\rightarrow$ is an application of a transformation rule.
R1. **Definition.** Introduce a new predicate definition

introduce \( C : \text{newp}(X) :- c, G \)

\[ S_{i+1} = S_i \cup \{C\} \quad \text{Defs} := \text{Defs} \cup \{C\} \]
Transformation Rules for CHCs

Initial clauses: \( S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \) Final clauses

where '→' is an application of a transformation rule.

R1. **Definition.** Introduce a new predicate definition

\[
C: \text{newp}(X) : c, \ G
\]

\[
S_{i+1} = S_i \cup \{C\} \quad \text{Defs := Defs} \cup \{C\}
\]

R2. **Unfolding.** Apply a Resolution step

given \( C: H : c, A, G \quad A : d_{1, G_1} \ldots A : d_{m, G_m} \) in \( S_i \)

derive \( S_i = \{ H : c, d_{1, G_1}, G \ldots H : c, d_{m, G_m}, G \} \)

\[
S_{i+1} = (S_i - \{C\}) \cup S
\]
... Transformation rules for CHCs

R3. **Folding.** Replace a conjunction by a new predicate

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

derive \( D: H : d, \text{newp}(X), G. \)

\( S_{i+1} = (S_i - \{C\}) \cup \{D\} \)
... Transformation rules for CHCs

R3. **Folding.** Replace a conjunction by a new predicate

given \( C: H \quad : \quad d, B, G \) in \( S_i \) \hspace{1cm} newp(X) :- c, B. with \( d \rightarrow c \) in Defs

derive \hspace{1cm} 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 \( C: H \quad : \quad c, B, G \) in \( S_i \) with \( Th \models c \leftrightarrow d \)

derive \hspace{1cm} 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 \( S_i \) \hspace{1cm} \text{newp}(X) :- c, B. \) with \( d \rightarrow c \) in Defs
derive \( D: H : - d, \text{newp}(X), G. \)

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

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

given \( C: H : - c, B, G \) in \( S_i \) with \( \text{Th} \models c \iff d \)
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  \( C : H : \neg d, B, G \) in \( S_i \)  
\[ \text{newp}(X) : - c, B. \]  
with  \( d \rightarrow c \) in Defs

Derive  \( D : H : \neg d, \text{newp}(X), G. \)

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

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

Given  \( C : H : \neg c, B, G \) in \( S_i \) with  \( \text{Th} \models c \iff d \)

Derive  \( D : H : \neg d, B, G \)

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

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 \( S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \) 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 $\text{false} : - c, A$, where $A$ is atomic, transform $S$ into a set of clauses $S_{\text{SP}}$ such that

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

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

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

∀X. p(X,b) → X≥0

% the X-th Turing machine halts on X
Specialization Strategy: An Example

Define: \( q(X) := X<0, p(X,b). \)

\[
\begin{align*}
\text{false} & :- X<0, p(X,b). \\
p(X,C) & :- X=Y+1, p(Y,C). \\
p(X,a). \\
p(X,b) & :- X\geq 0, \text{tm_halts}(X).
\end{align*}
\]

% \( \forall X. p(X,b) \rightarrow X\geq 0 \) \hspace{1cm} S_0

% the X-th Turing machine halts on X

% \( q(X) \) is a specialization of \( p(X,C) \) \hspace{1cm} 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
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).

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

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

Unfold: q(X) :- X<0, X=Y+1, p(Y,b).
q(X) :- X<0, X≥0, tm_halts(X).
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\geq0, tm_halts(X).

% \forall X. p(X,b) \rightarrow X\geq0

% 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)
% to a specific constraint on X and value of C

Unfold: \ q(X) :- X<0, X=Y+1, p(Y,b).
\ q(X) :- X<0, X\geq0, 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\geq0, \text{tm\_halts}(X). % the X-th Turing machine halts on X

% \forall X. p(X,b) \rightarrow X\geq0

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

Unfold: q(X) :- X<0, X=Y+1, p(Y,b).
q(X) :- X<0, X\geq0, \text{tm\_halts}(X). % clause removal

Fold: false :- X<0, q(X).
q(X) :- X<0, X=Y+1, q(Y).

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”

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:

amax(A,I,N,M) → ∃K.(I≤K<N, read(A,K,M)) → S≥M

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

\[ \text{false} : - S < M, \ 0 \leq I < N, \ \text{asum}(A, I, N, S), \ \text{amax}(A, I, N, M). \]  \quad S_0
Tupling Strategy: An Example

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

Tupling Strategy: An Example


Unfold: asummax(A,I,N,S,M) :- I=N-1, S=0, M=0.

& 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


Define:  

Unfold:  
assummax(A,I,N,S,M) :- I=N-1, S=0, M=0.
& CR  
assummax(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:  
assummax(A,I,N,S,M) :- I=N-1, S=0, M=0.
assummax(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)), assummax(A,I1,N,S1,M1).

false :- S<M, 0≤I<N, assummax(A,I,N,S,M).

Proof of satisfiability:  
assummax(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

1. Define
2. Unfold
3. Replace Constraints
4. Remove Clauses
5. Fold?

If no: Return to step 1
If yes: Move to next step

$S_0$ → $S_n$
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

Program P in L → CHC Specializer → VC
Property F → CHC Specializer → VC
Interp_L → CHC Specializer → VC

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: \texttt{at(Label, Cmd)}.

\begin{verbatim}
x=0;  0. x=0;
y=0;  1. y=0;
while (x<n) {
    x=x+1;  2. if (x<n) 3 else 6;
    y=x+y
}  3. x=x+1;
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
\end{verbatim}
A Small-Step Operational Semantics

• The operational semantics is a one-step transition relation between configurations
  \[ <n_1: cmd_1, \text{env}_1> \Rightarrow <n_2: cmd_2, \text{env}_2> \]
  where: \( n: cmd \) is a labelled command and \( \text{env} \) is an environment mapping variable identifiers to values;

• Assignment
  \[ <n: x=e, \text{env}> \Rightarrow <\text{next}(n), \text{update}(\text{env}, x, [e]\text{env})> \]
  where: \( \text{next}(n) \) is the next labelled command and \( \text{update}(\text{env}, x, [e]\text{env}) \) updates the value of \( x \) in \( \text{env} \) to the value of expression \( e \) in \( \text{env} \);

• Conditional
  \[ <n: \text{if } (e) n_1 \text{ else } n_2, \text{env}> \Rightarrow <\text{next}(n_1), \text{env}> \quad \text{if } [e]\delta \neq 0 \]
  \[ <n: \text{if } (e) n_1 \text{ else } n_2, \text{env}> \Rightarrow <\text{next}(n_2), \text{env}> \quad \text{if } [e]\delta = 0 \]

• Jump
  \[ <n: \text{goto } n_1, \text{env}> \Rightarrow <\text{next}(n_1), \text{env}> \]
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), where 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:
  \( \text{tr}( \text{cf}(LC1,Env1), \text{cf}(LC2,Env2) ) \)
\textbf{assignment} \hspace{5mm} x=e;

\textbf{source configuration} \hspace{5mm} \textbf{target configuration}

\texttt{tr(} \hspace{5mm} \texttt{cf(cmd(L, asgn(X,E)), Env1)}, \hspace{5mm} \texttt{cf(cmd(L1, C), Env2)) :-}
\texttt{nextlab(L,L1),} \hspace{5mm} % \texttt{next label}
\texttt{at(L1,C),} \hspace{5mm} % \texttt{next command}
\texttt{eval(E,Env1,V),} \hspace{5mm} % \texttt{evaluate expression}
\texttt{update(Env1,X,V,Env2).} \hspace{5mm} % \texttt{update environment}

More clauses for predicate \texttt{tr} to encode the semantics of the other commands.
Encoding Partial Correctness Properties

- **Partial correctness** specification (Hoare triple):
  \[
  \{ \varphi \} \text{prog} \{ \psi \}
  \]
  
  If the initial values of the program variables satisfy the precondition \( \varphi \) and \( \text{prog} \) terminates, then the final values of the program variables satisfy the postcondition \( \psi \).

- **CHC encoding** of partial correctness:

  ```prolog
  false :- initConf(Cf), reach(Cf).
  reach(Cf1) :- tr(Cf1,Cf2), reach(Cf2).
  reach(Cf) :- errorConf(Cf).
  initConf(cf(C, Env)) :- at(0,C), \varphi(Env).
  errorConf(cf(C, Env)) :- at(h,C), \neg \psi(Env).
  tr(cf1,cf2) :- ... 
  ```

  \textit{PC-prop}

  - \( \{ \varphi \} \text{prog} \{ \psi \} \) is **valid** iff **PC-prop** is satisfiable.
VCGen: Generating Verification Conditions

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

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

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

- VCGen is parametric wrt \( \text{Interp}_L \) (to a large extent).

- If \( \text{PC-prop} \xrightarrow{VCGen} \text{VC} \) then **PC-prop** is **satisfiable** iff **VC** is **satisfiable**
Generating Verification Conditions: An Example

PC property:

\{n \geq 1\} \text{SumUpto} \ \{y > x\}

CHC encoding:

\begin{align*}
\text{false} & :- \text{initConf}(\text{Cf}), \text{reach}(\text{Cf}). \\
\text{reach}(\text{Cf1}) & :- \text{tr}(\text{Cf1, Cf2}), \text{reach}(\text{Cf2}). \\
\text{reach}(\text{Cf}) & :- \text{errorConf}(\text{Cf}). \\
\text{initConf}(\text{cf}(\text{C, [(x, X, y, Y), (n, N)]})) & :- \text{at}(0, C), \ N \geq 1. \\
\text{errorConf}(\text{cf}(\text{C, [(x, X, y, Y), (n, N)]})) & :- \text{at}(6, C), \ Y \leq X. \\
\text{tr}(\text{cf1, cf2}) & :- \ldots \\
\ldots & \\
\text{at}(0, \text{asgn}(\text{int}(x), \text{int}(0))).
\end{align*}

VCGen

Verification Conditions:

\begin{align*}
\text{false} & :- \ N \geq 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 \geq N, \ Y < X.
\end{align*}
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
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

Eindhoven, April 3rd, 2016
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.

\[
\text{false} :- X<0, p(X,b).
\]

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

\[
p(X,a).
\]

\[
p(X,b) :- X \geq 0, \text{tm_halts}(X).
\]

\[
\text{false} :- X<0, q(X).
\]

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

\[
p(X,a).
\]

\[
p(X,b) :- X \geq 0, \text{tm_halts}(X).
\]

\[
\text{false} :- X<0, q(X).
\]

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

\[
p(X,a).
\]

\[
p(X,b) :- X \geq 0, \text{tm_halts}(X).
\]

\[
\text{false} :- X<0, q(X).
\]

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

\[
p(X,a).
\]

\[
p(X,b) :- X \geq 0, \text{tm_halts}(X).
\]
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).

$\text{false} \leftarrow a(X), p(X)$.  
$p(X) \leftarrow c(X,Y), p(Y)$.  
$p(X) \leftarrow b(X)$.  

\[
\text{Reversal}
\]

$\text{p(X)} \leftarrow a(X)$.  
$p(Y) \leftarrow c(X,Y), p(X)$.  
false $\leftarrow b(X), p(X)$.

\[
\text{VC is satisfiable iff } VC' \text{ is satisfiable}
\]

\[
\begin{array}{cccccc}
VCTransf & VCTransf & VCTransf & \cdots & VCTransf \\
VC_0 & \rightarrow & VC_1 & \rightarrow & VC_2 & \rightarrow & VC_3 & \rightarrow & \cdots & \rightarrow & VC_n \\
\end{array}
\]
Iterated CHC Specialization: *SumUpto* Example

false :- \( N \geq 1 \), \( X = 0 \), \( Y = 0 \), \( p(X, Y, N) \).

\[ VC_0 \]

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

\( p(X, Y, N) :- X \geq N \), \( Y < X \).
Iterated CHC Specialization: *SumUpto* Example

\[
\text{false :- } N \geq 1, X=0, Y=0, p(X, Y, N). \quad \text{VC}_0
\]

\[
p(X, Y, N) :- X < N, X_1=X+1, Y_1=Y+2, p(X_1, Y_1, N).
\]

\[
p(X, Y, N) :- X \geq N, Y < X.
\]

\[
\text{false :- } N \geq 1, X_1=1, Y_1=1, \text{new2}(X_1, Y_1, N). \quad \text{VC}_1
\]

\[
\text{new2}(X, Y, N) :- X=1, Y=1, N>1, X_1=2, Y_1=3, \text{new3}(X_1, Y_1, N).
\]

\[
\text{new3}(X, Y, N) :- X_1 \geq 1, Y_1 \geq X_1, X < N, X_1=X+1, Y_1=X_1+Y, \text{new3}(X_1, Y_1, N).
\]

\[
\text{new3}(X, Y, N) :- Y \geq 1, N \geq 1, X \geq N, Y < X.
\]
Iterated CHC Specialization: *SumUpto* Example

\[
\text{false} :- \; N \geq 1, \; X=0, \; Y=0, \; p(X, \; Y, \; N). \quad \text{VC}_0
\]
\[
p(X, \; Y, \; N) :- \; X < N, \; X1=X+1, \; Y1=Y+2, \; p(X1, \; Y1, \; N).
\]
\[
p(X, \; Y, \; N) :- \; X \geq N, \; Y<X.
\]

\[
\text{false} :- \; N \geq 1, \; X1=1, \; Y1=1, \; \text{new2}(X1, \; Y1, \; N). \quad \text{VC}_1
\]
\[
\text{new2}(X, \; Y, \; N) :- \; X=1, \; Y=1, \; N>1, \; X1=2, \; Y1=3, \; \text{new3}(X1, \; Y1, \; N).
\]
\[
\text{new3}(X, \; Y, \; N) :- \; X1 \geq 1, \; Y1 \geq X1, \; X < N, \; X1=X+1, \; Y1=X1+Y, \; \text{new3}(X1, \; Y1, \; N).
\]
\[
\text{new3}(X, \; Y, \; N) :- \; Y \geq 1, \; N \geq 1, \; X \geq N, \; Y<X.
\]

\[
\text{false} :- \; N \geq 1, \; X1=1, \; Y1=1, \; \text{new2}(X1, \; Y1, \; N). \quad \text{VC}_2
\]
\[
\text{new2}(X1, \; Y1, \; N) :- \; N \geq 1, \; X1=1, \; Y1=1.
\]
\[
\text{new3}(X1, \; Y1, \; N) :- \; X=1, \; Y=1, \; N>1, \; X1=2, \; Y1=3, \; \text{new2}(X, \; Y, \; N).
\]
\[
\text{new3}(X1, \; Y1, \; N) :- \; X1 \geq 1, \; Y1 \geq X1, \; X < N, \; X1=X+1, \; Y1=X1+Y, \; \text{new3}(X, \; Y, \; N).
\]
\[
\text{false} :- \; N \geq 1, \; X \geq N, \; Y<X, \; \text{new3}(X, \; Y, \; N).
\]
Iterated CHC Specialization: *SumUpto* Example

\[
\begin{align*}
\text{false} &: - N \geq 1, X=0, Y=0, p(X, Y, N). & \text{VC}_0 \\
p(X, Y, N) &: - X < N, X1=X+1, Y1=Y+2, p(X1, Y1, N). \quad & \\
p(X, Y, N) &: - X \geq N, Y < X. \\
\end{align*}
\]

\[\text{VCTransf}\]

\[
\begin{align*}
\text{false} &: - N \geq 1, X1=1, Y1=1, \text{new2}(X1, Y1, N). & \text{VC}_1 \\
\text{new2}(X, Y, N) &: - X=1, Y=1, N > 1, X1=2, Y1=3, \text{new3}(X1, Y1, N). \\
\text{new3}(X, Y, N) &: - X1 \geq 1, Y1 \geq X1, X < N, X1=X+1, Y1=X1+Y, \text{new3}(X1, Y1, N). \quad & \\
\text{new3}(X, Y, N) &: - Y \geq 1, N \geq 1, X \geq N, Y < X. \\
\end{align*}
\]

\[\text{Reversal}\]

\[
\begin{align*}
\text{new2}(X1, Y1, N) &: - N \geq 1, X1=1, Y1=1. & \text{VC}_2 \\
\text{new3}(X1, Y1, N) &: - X=1, Y=1, N > 1, X1=2, Y1=3, \text{new2}(X, Y, N). \\
\text{new3}(X1, Y1, N) &: - X1 \geq 1, Y1 \geq X1, X < N, X1=X+1, Y1=X1+Y, \text{new3}(X, Y, N). \\
\text{false} &: - N \geq 1, Y \geq 1, X \geq N, Y < X, \text{new3}(X, Y, N). \quad & \\
\end{align*}
\]

\[\text{VCTransf}\]

\[
\begin{align*}
\text{false} &: - N \geq 1, Y \geq 1, X \geq N, Y < X, \text{new4}(X, Y, N). & \text{VC}_3 \\
\end{align*}
\]

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]

<table>
<thead>
<tr>
<th></th>
<th>VeriMAP ($Gen_{ph}$)</th>
<th>ARMC</th>
<th>HSF(C)</th>
<th>TRACER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 correct</td>
<td>185</td>
<td>138</td>
<td>159</td>
<td>91</td>
</tr>
<tr>
<td>2 safe</td>
<td>154</td>
<td>112</td>
<td>137</td>
<td>74</td>
</tr>
<tr>
<td>3 unsafe</td>
<td>31</td>
<td>26</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>4 incorrect</td>
<td>0</td>
<td>9</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>5 false</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>6 missed</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7 errors</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>8 timed-out</td>
<td>31</td>
<td>51</td>
<td>52</td>
<td>92</td>
</tr>
<tr>
<td>9 total score</td>
<td>339 (0)</td>
<td>210 (-40)</td>
<td>268 (-28)</td>
<td>113 (-52)</td>
</tr>
<tr>
<td>10 total time</td>
<td>10717.34</td>
<td>15788.21</td>
<td>15770.33</td>
<td>27757.46</td>
</tr>
<tr>
<td>11 average time</td>
<td>57.93</td>
<td>114.41</td>
<td>99.18</td>
<td>305.03</td>
</tr>
</tbody>
</table>

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

\[\text{fibonacci: while } (n > 0) \{\]
\[\quad t = u;\]
\[\quad u = u + v;\]
\[\quad v = t;\]
\[\quad n = n - 1\]
\[\}\]
\[\{n = N, N \geq 0, u = 1, v = 0, t = 0}\text{ fibonacci} \{\text{fib}(N, u)\}\]

• The postcondition of \text{fibonacci} cannot be specified by using linear arithmetic constraints \textit{only}.
• Recursive Horn specifications:
  \{z_1 = P_1, \ldots, z_s = P_s, \textbf{pre}(P_1, \ldots, P_s)\} \textit{prog} \{\textbf{post}(P_1, \ldots, P_s, z)\}

where:
- \(z_1, \ldots, z_s\) are global variables of \textit{prog};
- \(P_1, \ldots, P_s\) are parameters;
- \(z\) is a variable in \{\(z_1, \ldots, z_s\}\};
- \textbf{pre} and \textbf{post} are defined by a set of CHCs;
- \textbf{post} is a functional relation: \(z = F(P_1, \ldots, P_s)\) for some function \(F\) defined for all \(P_1, \ldots, P_s\) that satisfy \textbf{pre}.

• All computable functions on integers can be specified by sets of CHCs.
Recursive Horn Specification for Fibonacci

Fibonacci specification:

\{n=N, N\geq 0, u=1, v=0, t=0\} fibonacci \{fib(N,u)\}

where:

\textit{Fib}:

\begin{align*}
\text{fib}(0,F) & : - F=1. \\
\text{fib}(1,F) & : - F=1. \\
\text{fib}(N3,F3) & : - N1\geq 0, N2=N1+1, N3=N2+1, F3=F1+F2, \text{fib}(N1,F1), \text{fib}(N2,F2).
\end{align*}
Translating Partial Correctness into CHCs

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

Step 1. Translate the operational semantics into CHCs;

Step 2. Generate verification conditions as a set of CHCs.
Translating the Operational Semantics

• Define a relation \textit{fibonacci\_prog}(N,U) such that, for all integers N, if the program variables satisfy the precondition

\[
n=N, \text{ } N \geq 0, \text{ } u=1, \text{ } v=0, \text{ } t=0
\]

then the final value of u computed by program \textit{fibonacci} is U.

• \textit{fibonacci\_prog} is defined by a set \textit{OpSem} of clauses that encode the operational semantics:

\[
\text{fibonacci\_prog}(N,U) :\text{initConf}(Cf0,N), \text{reach}(Cf0,Cfh), \text{finalConf}(Cfh,U).
\]

\[
\text{initConf(cf(LC, [(n,N),(u,U),(v,V),(t,T)]), N)} :\text{N} \geq 0, \text{ } U=1, \text{ } V=0, \text{ } T=0, \text{ } at(0,LC).
\]

\[
\text{finalConf(cf(LC, [(n,N),(u,U),(v,V),(t,T)]), U)} :\text{at(last,LC)}.
\]

\[
\text{reach(Cf,Cf)}.
\]

\[
\text{reach(Cf0,Cf2)} :\text{tr(Cf0,Cf1), reach(Cf1,Cf2)}.
\]

\text{tr(Cf0,Cf1)} is the interpreter of the imperative language.
Specializing the Operational Semantics

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

- $OpSem_{SP}$:

  - $fibonacci\_prog(N,F)$ :-
    - $N\geq0$, $U=1$, $V=0$, $T=0$,
    - $r(N,U,V,T,\ N1,F,V1,T1)$.

  - $r(N,U,V,T, \ N2,U2,V2,T2)$ :-
    - $N\geq1$, $N1=N-1$, $U1=U+V$, $V1=U$, $T1=U$,
    - $r(N1,U1,V1,T1,\ N2,U2,V2,T2)$.

  - $r(N,U,V,T,\ N,U,V,T)$ :- $N<0$.

- For every query false :- $G$, $OpSem \cup \{\text{false} :- G\}$ is satisfiable iff $OpSem_{SP} \cup \{\text{false} :- G\}$ is satisfiable.

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

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

plus clauses for fibonacci_progm (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 $\text{post}(P_1, \ldots, P_s, Z) :- B$ defining the postcondition, into a query for $\text{prog}$:
  1. Replace $\text{post}$ by $\text{prog}$ in the head and in the body
     $$\text{prog}(P_1, \ldots, P_s, Z) :- B'$$
  2. Move the conclusion to the premise (exploiting functionality of $\text{prog}$):
     $$\text{false} :- Y \neq Z, \text{prog}(P_1, \ldots, P_s, Y), B'$$
     where $Y$ is a new variable
  3. Case split:
     $$\text{false} :- Y > Z, \text{prog}(X_1, \ldots, X_s, Z), B'$$
     $$\text{false} :- Y < Z, \text{prog}(X_1, \ldots, X_s, Z), B'$$

- If for all generated queries $\text{false} :- G$, $\text{OpSem}_{sp} \cup \{\text{false} :- G\}$ is satisfiable, then
  $$\{z_1= P_1, \ldots, z_s= P_s, \text{pre}(P_1, \ldots, P_s)\} \text{prog} \{\text{post}(P_1, \ldots, P_s, z)\}$$
  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} U {Qi} is satisfiable.
Satisfiability via LA-Solvability

- Consider constraints $C_{LA}$ in Linear (Integer) Arithmetics (linear equalities and inequalities over the integers). An LA-solution of a set $S$ of CHCs is a mapping
  \[ \Sigma : \text{Atom} \rightarrow C_{LA} \]
  such that, for every clause $A_0 \vdash c, A_1, \ldots, A_n$ in $S$,
  \[ \text{LA} \models \forall (c, \Sigma(A_1) \land \ldots \land \Sigma(A_n) \rightarrow \Sigma(A_0)) \]

- 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. ($C_{LA}$ is r.e. and entailment is decidable.) Complete LA-solving methods exist (e.g., CEGAR).
Limitations of LA-solving

- Program \textit{fibonacci} is partially correct and each \textit{OpSem} U \{Qi\} is satisfiable.

- However, there is no LA-solution for \textit{OpSem} U \{Q5\} (nor for \textit{OpSem} U \{Q6\}).

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

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

- LA-solvers cannot prove the partial correctness of \textit{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} U \{Q5\} into a set of linear recursive clauses.
The Linearization Transformation

Nonlinear queries → Nonlinear clauses
The Linearization Transformation

\[ H \leftarrow c, p_1(X_1), \ldots, p_k(X_k). \]
The Linearization Transformation

Nonlinear clauses

H :- c, p₁(X₁), ..., pₖ(Xₖ).

Unfold using OpSem_RI

H :- d, q₁(X₁), ..., qⱼ(Xⱼ).

H :- e, r₁(X₁), ..., rₘ(Xₘ).
The Linearization Transformation

Define:
\[ \text{newr}(X_1, \ldots, X_m) :\]
\[ \text{r}_1(X_1), \ldots, \text{r}_m(X_m). \]

Define:
\[ \text{newq}(X_1, \ldots, X_j) :\]
\[ \text{q}_1(X_1), \ldots, \text{q}_j(X_j). \]

Nonlinear clauses

Unfold using \( OpSem_{RI} \)

\[ H :\]
\[ \text{c}, \text{p}_1(X_1), \ldots, \text{p}_k(X_k). \]

\[ H :\]
\[ \text{d}, \text{q}_1(X_1), \ldots, \text{q}_j(X_j). \]

\[ H :\]
\[ \text{e}, \text{r}_1(X_1), \ldots, \text{r}_m(X_m). \]
The Linearization Transformation

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

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

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

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

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).

Nonlinear clauses

H :- c, p_1(X_1), \ldots, p_k(X_k).
The Linearization Transformation

Nonlinear clauses

\[ H \leftarrow c, p_1(X_1), \ldots, p_k(X_k). \]

Define:
\[ \text{newr}(X_1, \ldots, X_m) \leftarrow r_1(X_1), \ldots, r_m(X_m). \]

Fold:
\[ H \leftarrow d, \text{newr}(X_1, \ldots, X_m). \]

Define:
\[ \text{newq}(X_1, \ldots, X_j) \leftarrow q_1(X_1), \ldots, q_j(X_j). \]

Fold:
\[ H \leftarrow d, \text{newq}(X_1, \ldots, X_j). \]

Unfold using OpSem\textsubscript{RI}

\[ H \leftarrow d, q_1(X_1), \ldots, q_j(X_j). \]

\[ H \leftarrow e, r_1(X_1), \ldots, r_m(X_m). \]

Linear clauses
The Linearization Transformation

Nonlinear clauses

H :- c, p₁(X₁), ..., pₖ(Xₖ).

Unfold using OpSemₐₙₙ

H :- d, q₁(X₁), ..., qₗ(Xₗ).

Define:

newq(X₁, ..., Xₗ) :-
q₁(X₁), ..., qₗ(Xₗ).

Fold:

H :- d, newq(X₁, ..., Xₗ).

H :- e, r₁(X₁), ..., rₘ(Xₘ).

Define:

newr(X₁, ..., Xₘ) :-
r₁(X₁), ..., rₘ(Xₘ).

Fold:

H :- e, newr(X₁, ..., Xₘ).

Linear clauses
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).


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 $\text{false} : - G$, linearization terminates for the input $OpSem_{sp} \cup \{\text{false} : - G\}$.

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

• If $OpSem_{sp} \cup \{\text{false} : - G\}$ is LA-solvable, then $TransfCls$ is LA-solvable. Not vice versa: LA-solvability can be increased.
## Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>VCG</th>
<th>LA-solving-1</th>
<th>LIN</th>
<th>LA-solving-2: VeriMAP &amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Z3</td>
<td>Eldarica</td>
<td></td>
</tr>
<tr>
<td>1. binary_division</td>
<td>0.02</td>
<td>4.16</td>
<td>TO</td>
<td>0.04</td>
</tr>
<tr>
<td>2. fast_multiplication_2</td>
<td>0.02</td>
<td>TO</td>
<td>3.71</td>
<td>0.01</td>
</tr>
<tr>
<td>3. fast_multiplication_3</td>
<td>0.03</td>
<td>TO</td>
<td>4.56</td>
<td>0.02</td>
</tr>
<tr>
<td>4. fibonacci</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>5. Dijkstra_fusc</td>
<td>0.01</td>
<td>1.02</td>
<td>3.80</td>
<td>0.05</td>
</tr>
<tr>
<td>6. greatest_common_divisor</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>7. integer_division</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>8. 91-function</td>
<td>0.01</td>
<td>1.27</td>
<td>TO</td>
<td>0.06</td>
</tr>
<tr>
<td>9. integer_multiplication</td>
<td>0.02</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>10. remainder</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>11. sum_first_integers</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>12. lucas</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>13. padovan</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>14. perrin</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.02</td>
</tr>
<tr>
<td>15. hanoi</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>16. digits10</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.01</td>
</tr>
<tr>
<td>17. digits10-itmd</td>
<td>0.06</td>
<td>TO</td>
<td>TO</td>
<td>0.04</td>
</tr>
<tr>
<td>18. digits10-opt</td>
<td>0.08</td>
<td>TO</td>
<td>TO</td>
<td>0.10</td>
</tr>
<tr>
<td>19. digits10-opt100</td>
<td>0.01</td>
<td>TO</td>
<td>TO</td>
<td>0.02</td>
</tr>
</tbody>
</table>
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