Program Verification using Constraint Handling Rules and Array Constraint Generalizations

Emanuele De Angelis[1,3], Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1]University of Chieti-Pescara 'G. d'Annunzio', Italy
[2]University of Rome 'Tor Vergata', Italy
[3]CNR - Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy

2nd International Workshop on
Verification and Program Transformation, **VPT 2014**
Vienna, July 17–18, 2014

Proof of Partial Correctness: An Example

Consider a **program** and a **partial correctness triple** for the program:

```
prog : while(x < n) {
          x = x+1 ;
          y = y+2 ;
       }
```

$\{\, x=0 \wedge y=0 \wedge n \geq 1\} \;\; prog \;\; \{y > x\}$

(A) Generate the **Verification Conditions** (VC's)

1.	$x=0 \wedge y=0 \wedge n \geq 1 \rightarrow \mathrm{P}(x,y,n)$	Initialization
2.	$\mathrm{P}(x,y,n) \wedge x < n \rightarrow \mathrm{P}(x+1,y+2,n)$	Loop
3.	$\mathrm{P}(x,y,n) \wedge x \geq n \rightarrow y > x$	Exit

(B) Then, prove **satisfiability** of the VC's.

If the VC's are **satisfiable**, then the **partial correctness triple** holds.

Proof of Satisfiability of VC's

VC's are **satisfiable** if there is an interpretation that makes them true.

In our case,

$$\mathrm{P}(x,y,n) \;\equiv\; (x=0 \land y=0 \land n\geq 1) \lor y>x$$

makes the VC's true. Indeed,

1'. $x=0 \land y=0 \land n\geq 1 \;\rightarrow\; (x=0 \land y=0 \land n\geq 1) \lor y>x$

2'. $((x=0 \land y=0 \land n\geq 1) \lor y>x) \;\land x<n$
$\rightarrow (x+1=0 \land y+2=0 \land n\geq 1) \lor y+2>x+1$

3'. $((x=0 \land y=0 \land n\geq 1) \lor y>x) \;\land x\geq n \;\rightarrow\; y>x$

Thus, $\{x=0 \land y=0 \land n\geq 1\}$ *prog* $\{y>x\}$ holds.

- How to ***automatically*** prove **satisfiability** of the VC's ?

Automatic Proofs of Satisfiability of VC's

Various methods:

- CounterExample Guided Abstraction Refinement (CEGAR), Interpolation, Satisfiability Modulo Theories [Rybalchenko et al., McMillan, Alberti et al.]
- Symbolic execution of Constraint Logic Programs [Jaffar et at.]
- Static Analysis and Transformation of Constraint Logic Programs [Gallagher et al., Albert et al.]

Our CLP Transformation Method

(A) Generate the VC's as a **constraint logic program (a CLP program)**:

V: 1^*. `p(X,Y,N) :- X=0, Y=0, N ≥ 1.` (a constrained fact)

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

3^*. `incorrect :- X≥N, Y≤X, p(X,Y,N).`

THM: The VC's are **satisfiable** iff `incorrect` $\notin$ **the least model** $M(V)$.

(B) Apply transformation rules that **preserve the least model** $M(V)$.

V': 4. `q(X1,Y1,N) :- X<N, X>Y, Y≥0, X1=X+1, Y1=Y+2, q(X,Y,N).`

5. `incorrect :- X≥N, Y≤X, Y≥0 N≥1, q(X,Y,N).`

least model preserved: `incorrect` $\notin M(V)$ iff `incorrect` $\notin M(V')$

no constrained facts for `q`: `incorrect` $\notin M(V')$

Thus, $\{x=0 \wedge y=0 \wedge n\geq 1\}$ *prog* $\{y>x\}$ holds.

Outline

(A) How to generate the VC's, i.e., V?
(B) How to prove satisfiability of the VC's, i.e., transform V into V'?

Basic ideas from [PEPM-13].

Rules and strategies for programs on integers and integer arrays.

- CLP program transformation:
  - Unfold/fold rules: preserving the least model
  - (A) Strategies for VC's generation:
    specialization of the interpreter (getting V)
  - (B) Strategies for VC's satisfiability proof:
    propagation of constraints (getting V')
- Running example: Ascending Array Initialization, e.g., [3, 4, 5, 6]
- Experimental evaluation.

Rules for Transforming CLP Programs

R1. **Definition**. Introducing a new predicate (e.g., a loop invariant)

`newp(X) :- c,A.`

R2. **Unfolding**. A symbolic evaluation step (e.g., a resolution step)

given $\quad$ `H :- c,`$\underline{\mathtt{A}}$`,G.`

$\underline{\mathtt{A}}$ `:-` d_1, G_1. , ..., $\underline{\mathtt{A}}$ `:-` d_m, G_m.

derive $\quad$ `H :- c,`d_1, G_1`,G.` , ..., `H :- c,`d_m, G_m`,G.`

R3. **Folding**. Using a predicate definition

given $\quad$ `H :- d,`$\underline{\mathtt{A}}$`,G.`

`newp(X) :- c,`$\underline{\mathtt{A}}$`.` $\quad$ and $\quad$ $d \rightarrow c$

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

R4. **Clause Removal**. Delete clauses with
(i) unsatisfiable constraint or (ii) subsumed by other clauses

'Rule + Strategies' Program Transformation

$$\begin{tikzcd} \boxed{P} \arrow[d, "r"] \\ P_1 \arrow[d, "r"] \\ P_2 \arrow[d, "r"] \\ \vdots \arrow[d, "r"] \\ \boxed{TransfP} \end{tikzcd}$$

- The transformation **rules**:

  $r \in \{$ Definition, Unfolding, Folding, Clause Removal $\}$

- The rules **preserve** the least model:

  Theorem (Least model preservation)

  $\texttt{incorrect} \in M(P)$ iff $\texttt{incorrect} \in M(TransfP)$

- The rules must be guided by **strategies**.

[Burstall-Darlington 77,Tamaki-Sato 84, Etalle-Gabbrielli 96]

Encoding Partial Correctness into CLP

Consider the triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$.

A program *prog* is incorrect w.r.t. φ_{init} and φ_{error}
if a final configuration satisfying φ_{error}
is reachable from an initial configuration satisfying φ_{init}.

Definition (the interpreter *Int* with the transition predicate `tr(X,Y)`)

```
reach(X) :- initConf(X).
reach(Y) :- tr(X,Y), reach(X).
incorrect :- errorConf(X), reach(X).
```

$+$ clauses for `tr` (i.e., the operat. semantics of the programming language)

Theorem

prog is incorrect iff `incorrect` $\in M(Int)$

A program *prog* is correct iff it is not incorrect.

tr(X,Y): the operational semantics

`L: Id = Expr`	`tr( cf(cmd(L,asgn(Id,Expr)),S), cf(cmd(L1,C1),S1)) :-` `aeval(Expr,S,V),` *evaluate expression* `update(Id,V,S,S1),` *update store* `nextlabel(L,L1),` *next label* `at(L1,C1).` *next command*

`L: if(Expr){`
`L1: ...`
`}`
`else`
`L2: ...`
`}`
	`tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-`
	`beval(Expr,S),` *expression is true*
	`at(L1,C).` *next command*
	`tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-`
	`beval(not(Expr),S),` *expression is false*
	`at(L2,C).` *next command*

`L: goto L1`	`tr( cf(cmd(L,goto(L1)),S), cf(C,S)) :-` `at(L1,C).` *next command*

`tr(X,Y)`: the operational semantics for array assignment

array assignment: L : a[ie] = e

`tr( cf(cmd(L,asgn(elem(A,IE),E)),S),`	*source configuration* `cf`
`    cf(cmd(L1,C),S1) ) :-`	*target configuration* `cf`
`      eval(IE,S,I),`	*evaluate index expr* `IE`
`      eval(E,S,V),`	*evaluate expression* `E`
`      lookup(S,array(A),FA),`	*get array* `FA` *from store*
`      write(FA,I,V,FA1),`	*update array* `FA`, *getting* `FA1`
`      update(S,array(A),FA1,S1),`	*update store* `S`, *getting* `S1`
`      nextlab(L,L1),`	*next label* `L1`
`      at(L1,C).`	*command* `C` *at next label*

Running Example: Ascending Array Initialization

Given the **program** *SeqInit* and the **partial correctness triple**

```
i=1;
while(i<n) {
 a[i]=a[i-1]+1;
 i=i+1;
}
```

$\{i \geq 0 \wedge n \geq 1 \wedge n = dim(\mathtt{a})\}$

SeqInit

$\{\forall j \ (0 \leq j \wedge j+1 < n \ \rightarrow \ \mathtt{a}[j] < \mathtt{a}[j+1])\}$

CLP encoding of program *SeqInit*

- A set of `at(label, command)` facts.
- `while` = `ite` + `goto`.
- `elem(a,i)` stands for `a[i]`.

$\mathtt{at}(\ell_0, \mathtt{asgn}(\mathtt{i}, 1)).$

$\mathtt{at}(\ell_1, \mathtt{ite}(\mathtt{less}(\mathtt{i}, \mathtt{n}), \ell_2, \ell_{\mathtt{h}})).$

$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{elem}(\mathtt{a}, \mathtt{i}), \mathtt{plus}(\mathtt{elem}(\mathtt{a}, \mathtt{minus}(\mathtt{i}, 1)), 1))).$

$\mathtt{at}(\ell_3, \mathtt{asgn}(\mathtt{i}, \mathtt{plus}(\mathtt{i}, 1))).$

$\mathtt{at}(\ell_4, \mathtt{goto}(\ell_1)).$

$\mathtt{at}(\ell_{\mathtt{h}}, \mathtt{halt}).$

CLP encoding of φ_{init} and φ_{error}

$\mathtt{initConf}(\ell_0, \mathtt{I}, \mathtt{N}, \mathtt{A})$:-
$\mathtt{I} \geq 0, \ \mathtt{N} \geq 1.$

$\mathtt{errorConf}(\ell_{\mathtt{h}}, \mathtt{N}, \mathtt{A})$:-
$\mathtt{W} \geq 0, \ \mathtt{W}+1 < \mathtt{N}, \ \boxed{\mathtt{Z} = \mathtt{W}+1, \ \mathtt{U} \geq \mathtt{V},}$
$\mathtt{read}(\mathtt{A}, \boxed{\mathtt{W}, \mathtt{U}}), \mathtt{read}(\mathtt{A}, \boxed{\mathtt{Z}, \mathtt{V}}).$

Program Transformation of CLP is used to

(A) generate the VC's

(B) prove the satisfiability of the VC's

Interpreter: *Int*

Specialize *Int* w.r.t. *prog* (generate the VC's)

Verification Conditions: VC's

Propagate φ_{init} or φ_{error}

(prove the satisfiability of the VC's)

?

prog correct

(no constrained facts in the VC's)

prog incorrect

(the fact `incorrect` is in the VC's)

The Strategy for Generation

Transform(P)

$\mathit{TransfP} = \emptyset$;

Defs $= \{$`incorrect :- errorConf(X), reach(X)`$\}$;

while $\exists q \in$ Defs **do**

```
% execute a symbolic evaluation step (i.e., resolution)
```

Cls $=$ Unfolding(q);

```
% remove unsatisfiable and subsumed clauses
```

Cls $=$ ClauseRemoval(Cls);

```
% introduce new predicates (i.e., a loop invariant)
```

Defs $=$ (Defs $- \{q\}$) $\cup$ Definition(Cls);

```
% match a predicate definition
```

$\mathit{TransfP} = \mathit{TransfP} \cup$ Folding(Cls, Defs);

od

Verification Conditions Generation

The specialization of *Int* w.r.t. *prog* removes all references to:

- `tr` and
- `at`

VC's: the Specialized Interpreter for *SeqInit*

```
incorrect :- Z=W+1, W≥0, W+1<N, U≥V, N≤I,
             read(A,W,U), read(A,Z,V), new1(I,N,A).
new1(I1,N,B) :- 1≤I, I<N, D=I-1, I1=I+1, V=U+1,
             read(A,D,U), write(A,I,V,B), new1(I,N,A).
new1(I,N,A) :- I=1, N≥1.
```

- A constrained fact is present:
  we cannot conclude that the program is correct.
- The fact `incorrect` is not present:
  we cannot conclude that the program is incorrect either.

The Strategy for Satisfiability

Transform(P)

$TransfP = \emptyset$;

Defs $= \{$`incorrect :- errorConf(X), reach(X)`$\}$;

while $\exists q \in$ Defs **do**

Cls $=$ Unfolding(q);

Cls $=$ ConstraintReplacement(Cls);

Cls $=$ ClauseRemoval(Cls);

Defs $= ($Defs $- \{q\}) \cup$ $\text{Definition}_{\text{array}}$(Cls);

$TransfP = TransfP \cup$ Folding(Cls, Defs);

od

Constraint Replacement Rule

If $\mathcal{A} \models \forall\,(\mathtt{c_0} \leftrightarrow (\mathtt{c_1} \vee \ldots \vee \mathtt{c_n}))$, where $\mathcal{A}$ is the Theory of Arrays

Then replace $\quad$ `H :- c0, d, G`

by $\quad$ `H :- c1, d, G,` $\quad \ldots, \quad$ `H :- cn, d, G`

Constraint Handling Rules for Constraint Replacement:

AC1. Array-Congruence-1: if i=j then a[i]=a[j]

$\mathtt{read(A, I, X)} \setminus \mathtt{read(A1, J, Y)} \Leftrightarrow \mathtt{A == A1, I = J} \mid \mathtt{X = Y}.$

AC2. Array-Congruence-2: if a[i]≠a[j] then i≠j

$\mathtt{read(A, I, X), read(A1, J, Y)} \Rightarrow \mathtt{A == A1, X <> Y} \mid \mathtt{I <> J}.$

ROW. Read-Over-Write: {a[i]=x; y=a[j]} if i=j then x=y

$\mathtt{write(A, I, X, A1)} \setminus \mathtt{read(A2, J, Y)} \Leftrightarrow \mathtt{A1 == A2} \mid$

$\mathtt{(I = J, X = Y)}\ ;\ \mathtt{(I <> J, read(A,J,Y))}.$

Ascending Array Initialization

```
new3(A,B,C) :-  A=2+H, B-H≤3, E-H≤1, E≥1, B-H≥2,... ,
  read(N,H,M), read(C,D,F), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW rule:

```
new3(A,B,C) :- J=1+D,  A=2+D,  K=1+I,  I<F,  ...,
  read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
new3(A,B,C) :- J=1+D,  A=2+D,  K=1+I,  I<F,  ...,
  read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW, AC1, and AC2 rules:

```
new3(A,B,C) :- A=1+H, E=1+D, J=-1+H, K=1+L, D-H≤-2, H<B,...
  read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C),
  reach(J,B,M).
```

Ascending Array Initialization

```
new3(A,B,C) :-  A=2+H, B-H≤3, E-H≤1, E≥1, B-H≥2,... ,
  read(N,H,M), read(C,D,F), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW rule:

```
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ..., J=E, K=G,
  read(C,D,F), read(N,D,I), write(N,J,K,C), ~~read(C,E,G)~~,
  reach(J,B,N).
```

```
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ...,
  read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW, AC1, and AC2 rules:

```
new3(A,B,C) :- A=1+H, E=1+D, J=-1+H, K=1+L, D-H≤-2, H<B,...
  read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C),
  reach(J,B,M).
```

Ascending Array Initialization

```
new3(A,B,C) :-  A=2+H, B-H≤3, E-H≤1, E≥1, B-H≥2,... ,
  read(N,H,M), read(C,D,F), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW rule:

```
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ..., J=E, K=G,
  read(C,D,F), read(N,D,I), write(N,J,K,C), ~~read(C,E,G)~~,
  reach(J,B,N).
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F,  ..., J<>E,
  read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW, AC1, and AC2 rules:

```
new3(A,B,C) :- A=1+H, E=1+D, J=-1+H, K=1+L, D-H≤-2, H<B,...
  read(N,E,G), read(N,D,F), read(N,J,L),write(N,H,K,C),
  reach(J,B,M).
```

Ascending Array Initialization

```
new3(A,B,C) :-  A=2+H, B-H≤3, E-H≤1, E≥1, B-H≥2,... ,
  read(N,H,M), read(C,D,F), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW rule:

```
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ..., J=E, K=G,
  read(C,D,F), read(N,D,I), write(N,J,K,C), ~~read(C,E,G)~~,
  reach(J,B,N).
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F,  ..., J<>E,
  read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
```

- by applying the ROW, AC1, and AC2 rules:

```
new3(A,B,C) :- A=1+H, E=1+D, J=-1+H, K=1+L, D-H≤-2, H<B,...
  read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C),
  reach(J,B,M).
```

Definition Introduction

Introduction of suitable new predicate **definitions** (they correspond to **program invariants**).

Difficulty: Introduction of an unbounded number of new predicate definitions.

Solution: Use of **generalization** operators:

- to ensure the termination of the transformation,
- to generate program invariants.

Note. They are two somewhat conflicting requirements:

- (efficiency) introduction of as few definitions as possible, and
- (precision) proof of as many satisfiability properties as possible.

Constraint Generalizations

Definitions are arranged as a tree:

```
incorrect :- err, A

...        newp :- c, B     : ancestor definition

           newq :- d, B     : candidate definition

           newr :- g, B     : generalized definition
```

d → g

Generalization operators based on widening and convex-hull.

Array Constraint Generalizations

```
incorrect :- err, A

...            newp :- c, read, B                    : ancestor definition

            newq :- d, read, read, B                 : candidate definition

newr :- g, read, B        ??        newr :- g, read, B
```

- We decorate CLP variables with the variable identifiers of the imperative program.

VC's: the Specialized Interpreter for *SeqInit*

$\mathtt{incorrect} \text{ :- } Z=W+1,\ W\geq 0,\ W+1<N,\ U\geq V,\ N\leq I,$
$\qquad \mathtt{read}(A, W^{j}, U^{a[j]}),\ \mathtt{read}(A, Z^{j1}, V^{a[j1]}),\ \mathtt{new1}(I,N,A).$

$\mathtt{new1}(I1,N,B) \text{ :- } 1\leq I,\ I<N,\ D=I-1,\ I1=I+1,\ V=U+1,$
$\qquad \mathtt{read}(A, D^{i}, U^{a[i]}),\ \mathtt{write}(A,I,V,B),\ \mathtt{new1}(I,N,A).$

$\mathtt{new1}(I,N,A) \text{ :- } I=1,\ N\geq 1.$

Array Constraint Generalizations

```
incorrect :- err, A

...        newp :- c, read, B          : ancestor definition

        newq :- d, read, read, B       : candidate definition

newr :- g, read, B     ??     newr :- g, read, B
```

- We decorate CLP variables with the variable identifiers of the imperative program.

VC's: the Specialized Interpreter for *SeqInit*

incorrect :- $Z = W+1$, $W \geq 0$, $W+1 < N$, $U \geq V$, $N \leq I$,
  read($A, W^{j}, U^{a[j]}$), read($A, Z^{j1}, V^{a[j1]}$), new1(I, N, A).
new1($I1, N, B$) :- $1 \leq I$, $I < N$, $D = I-1$, $I1 = I+1$, $V = U+1$,
  read($A, D^{i}, U^{a[i]}$), write(A, I, V, B), new1(I, N, A).
new1(I, N, A) :- $I = 1$, $N \geq 1$.

Ascending Array Initialization

: ancestor definition

new3(I,N,A) :- E+1=F, E≥0, I>F, G≥H, N>F, N≤I+1,
read(A,E^{j},$G^{a[j]}$), read(A,F^{j1},$H^{a[j1]}$), reach(I,N,A).

: candidate definition

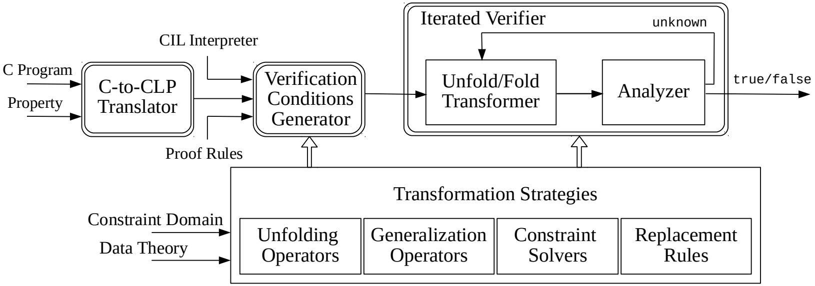
new4(I,N,A) :- E+1=F, E≥0, I>F, G≥H, I=1+I1, I1+2≤C, N≤I1+3,
read(A,E^{j},$G^{a[j]}$), read(A,F^{j1},$H^{a[j1]}$), read(A,P^{i},$Q^{a[i]}$),
reach(I,N,A).

: **generalized** definition

new5(I,N,A) :- E+1=F, E≥0, I>F, G≥H, N>F,
read(A,E^{j},$G^{a[j]}$), read(A,F^{j1},$H^{a[j1]}$), reach(I,N,A).

In the paper: a variable of the form G^{v} is encoded by `val(v,G)`.

- The VeriMAP tool `http://map.uniroma2.it/VeriMAP`



Experimental evaluation

Program	$Gen_{W,\mathcal{I},\Cap}$	$Gen_{H,\mathcal{V},\subseteq}$	$Gen_{H,\mathcal{V},\Cap}$	$Gen_{H,\mathcal{I},\subseteq}$	$Gen_{H,\mathcal{I},\Cap}$
bubblesort-inner	0.9	*unknown*	*unknown*	*unknown*	1.52
copy-partial	*unknown*	*unknown*	3.52	3.51	3.54
copy-reverse	*unknown*	*unknown*	5.25	*unknown*	5.23
copy	*unknown*	*unknown*	5.00	4.88	4.90
find-first-non-null	0.14	0.66	0.64	0.28	0.27
find	1.04	6.53	2.35	2.33	2.29
first-not-null	0.11	0.22	0.22	0.22	0.22
init-backward	*unknown*	1.04	1.04	1.03	1.04
init-non-constant	*unknown*	2.51	2.51	2.47	2.47
init-partial	*unknown*	0.9	0.89	0.9	0.89
init-sequence	*unknown*	4.38	4.33	4.41	4.29
init	*unknown*	1.00	0.97	0.98	0.98
insertionsort-inner	0.58	2.41	2.4	2.38	2.37
max	*unknown*	*unknown*	0.8	0.81	0.82
partition	0.84	1.77	1.78	1.76	1.76
rearrange-in-situ	*unknown*	*unknown*	3.06	3.01	3.03
selectionsort-inner	*unknown*	*time-out*	*unknown*	2.84	2.83
precision	6	10	15	15	17
total time	3.61	21.42	34.76	31.81	38.45
average time	0.60	2.14	2.31	2.12	2.26

Conclusions and Future Work

- Parametric verification framework (semantics, logics, constraint domains)
  - CLP as a metalanguage
  - agile way of synthesizing software verifiers [Rybalchenko et al.]
- Semantics preserving transformations
  - iterative verification
  - use Horn clauses for passing information between verifiers [McMillan]
- **Future work**
  - more experiments (including programs with nested loops)
  - more theories (lists, heaps, etc.)
  - Other programming languages and properties.