Verifying Array Programs by Transforming Verification Conditions

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

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

VMCAI 2014
San Diego, Ca, USA, January 19-21, 2014

Proving Partial Correctness

Given the program *prog*:

$x = 0;\ \ y = 0;$
$\texttt{while}\ (x < n)\ \{x = x+1;\ \ y = y+2\}$

and the specification:

$\{n \geq 1\}\ \mathit{prog}\ \{y > x\}$

Generate the verification conditions (VCs):

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

and prove they are satisfiable, i.e., we can find an interpretation for P that makes the VCs true.

Proving Partial Correctness

Given the program *prog*:

$x = 0;\ \ y = 0;$
$\texttt{while}\ (x < n)\ \{x = x + 1;\ \ y = y + 2\}$

and the specification:

$\{n \geq 1\}\ prog\ \{y > x\}$

Generate the verification conditions (VCs):

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

and prove they are satisfiable, i.e., we can find an interpretation for P that makes the VCs true.

The interpretation

$$P(x, y, n) \;\equiv\; (x = 0 \wedge y = 0 \wedge n \geq 1) \vee y > x$$

makes the VCs true

1'. $x = 0 \wedge y = 0 \wedge n \geq 1 \rightarrow (x = 0 \wedge y = 0 \wedge n \geq 1) \vee y > x$

2'. $((x = 0 \wedge y = 0 \wedge n \geq 1) \vee y > x) \wedge x < n$
$\rightarrow (x + 1 = 0 \wedge y + 2 = 0 \wedge n \geq 1) \vee y + 2 > x + 1$

3'. $((x = 0 \wedge y = 0 \wedge n \geq 1) \vee y > x) \wedge x \geq n \rightarrow y > x$

and hence the specification $\{n \geq 1\}\ prog\ \{y > x\}$ is valid.

Problem: How to find the interpretation for P automatically?

The interpretation

$$P(x,y,n) \;\equiv\; (x=0 \wedge y=0 \wedge n\geq 1) \vee y>x$$

makes the VCs true

1'. $x=0 \wedge y=0 \wedge n\geq 1 \rightarrow (x=0 \wedge y=0 \wedge n\geq 1) \vee y>x$

2'. $((x=0 \wedge y=0 \wedge n\geq 1) \vee y>x) \wedge x<n$
$\qquad\qquad \rightarrow (x+1=0 \wedge y+2=0 \wedge n\geq 1) \vee y+2>x+1$

3'. $((x=0 \wedge y=0 \wedge n\geq 1) \vee y>x) \wedge x\geq n \rightarrow y>x$

and hence the specification $\{n\geq 1\}\, prog\, \{y>x\}$ is valid.

Problem: How to find the interpretation for P automatically?

Proving Satisfiability of Verification Conditions

- The VCs are a set of Horn clauses with constraints
- or, equivalently, a constraint logic program V:

1.	$x=0 \wedge y=0 \wedge n \geq 1 \;\rightarrow\; P(x, y, n)$	Constrained fact
2.	$P(x, y, n) \wedge x < n \;\rightarrow\; P(x+1, y+2, n)$	Rule
4.	$P(x, y, n) \wedge x \geq n \wedge y \leq x \rightarrow \; \mathit{false}$	Query

  The VCs are satisfiable iff *false* not in the least model of V.

- Methods for proving the satisfiability of VCs within CHC/CLP:
  - CounterExample Guided Abstraction Refinement, Interpolation, Satisfiability Modulo Theories
  - Symbolic execution of CLP
  - Static Analysis and Transformation of CLP

Proving Satisfiability of Verification Conditions

- The VCs are a set of Horn clauses with constraints
- or, equivalently, a constraint logic program V:

1.	$x=0 \wedge y=0 \wedge n \geq 1 \rightarrow P(x, y, n)$	Constrained fact
2.	$P(x, y, n) \wedge x < n \rightarrow P(x+1, y+2, n)$	Rule
4.	$P(x, y, n) \wedge x \geq n \wedge y \leq x \rightarrow \mathit{false}$	Query

  The VCs are satisfiable iff *false* not in the least model of V.

- Methods for proving the satisfiability of VCs within CHC/CLP:
  - CounterExample Guided Abstraction Refinement, Interpolation, Satisfiability Modulo Theories
  - Symbolic execution of CLP
  - Static Analysis and Transformation of CLP

A Transformation-based Method

- Apply transformations that preserve the least model to V:

  1. $x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)$
  2. $P(x, y, n) \land x < n \rightarrow P(x+1, y+2, n)$
  4. $P(x, y, n) \land x \geq n \land y \leq x \rightarrow \mathit{false}$

  and derive the equisatisfiable V':

  5. $Q(x, y, n) \land x < n \land x > y \land y \geq 0 \rightarrow Q(x+1, y+2, n)$
  6. $Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \rightarrow \mathit{false}$

  No constrained facts: V' satisfiable with $Q(x, y, n) \equiv \mathit{false}$.

- Problem: How to transform V into V' automatically?
- Some work done for programs over integers [De Angelis et al. PEPM-13].
- This work: Design automatic transformation strategies of VCs for programs over arrays.

A Transformation-based Method

- Apply transformations that preserve the least model to V:

  1. $x=0 \wedge y=0 \wedge n\geq 1 \rightarrow P(x,y,n)$
  2. $P(x,y,n) \wedge x<n \rightarrow P(x+1,y+2,n)$
  4. $P(x,y,n) \wedge x\geq n \wedge y\leq x \rightarrow \mathit{false}$

  and derive the equisatisfiable V':

  5. $Q(x,y,n) \wedge x<n \wedge x>y \wedge y\geq 0 \rightarrow Q(x+1,y+2,n)$
  6. $Q(x,y,n) \wedge x\geq n \wedge x\geq y \wedge y\geq 0 \wedge n\geq 1 \rightarrow \mathit{false}$

  No constrained facts: V' satisfiable with $Q(x,y,n) \equiv \mathit{false}$.

- Problem: How to transform V into V' automatically?
- Some work done for programs over integers [De Angelis et al. PEPM-13].
- This work: Design automatic transformation strategies of VCs for programs over arrays.

A Transformation-based Method

- Apply transformations that preserve the least model to V:

  1. $x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)$
  2. $P(x, y, n) \land x < n \rightarrow P(x+1, y+2, n)$
  4. $P(x, y, n) \land x \geq n \land y \leq x \rightarrow \mathit{false}$

  and derive the equisatisfiable V':

  5. $Q(x, y, n) \land x < n \land x > y \land y \geq 0 \rightarrow Q(x+1, y+2, n)$
  6. $Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \rightarrow \mathit{false}$

  No constrained facts: V' satisfiable with $Q(x, y, n) \equiv \mathit{false}$.

- Problem: How to transform V into V' automatically?
- Some work done for programs over integers [De Angelis et al. PEPM-13].
- This work: Design automatic transformation strategies of VCs for programs over arrays.

A Transformation-based Method

- Apply transformations that preserve the least model to V:

  1. $x=0 \wedge y=0 \wedge n \geq 1 \rightarrow P(x,y,n)$
  2. $P(x,y,n) \wedge x<n \rightarrow P(x+1,y+2,n)$
  4. $P(x,y,n) \wedge x \geq n \wedge y \leq x \rightarrow \mathit{false}$

  and derive the equisatisfiable V':

  5. $Q(x,y,n) \wedge x<n \wedge x>y \wedge y \geq 0 \rightarrow Q(x+1,y+2,n)$
  6. $Q(x,y,n) \wedge x \geq n \wedge x \geq y \wedge y \geq 0 \wedge n \geq 1 \rightarrow \mathit{false}$

  No constrained facts: V' satisfiable with $Q(x,y,n) \equiv \mathit{false}$.

- Problem: How to transform V into V' automatically?
- Some work done for programs over integers [De Angelis et al. PEPM-13].
- This work: Design automatic transformation strategies of VCs for programs over arrays.

Outline of the Talk

- Constraint Logic Programming as a metalanguage for representing
  - the imperative program (integer and array variables)
  - the semantics of the imperative language (i.e., the interpreter)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving unfold/fold rules and strategies
  - VC generation by specialization of the interpreter
  - VC transformation by propagation of the property to be verified
- The verification method at work: Sequence Array Initialization
- Experimental evaluation

Outline of the Talk

- Constraint Logic Programming as a metalanguage for representing
  - the imperative program (integer and array variables)
  - the semantics of the imperative language (i.e., the interpreter)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving unfold/fold rules and strategies
  - VC generation by specialization of the interpreter
  - VC transformation by propagation of the property to be verified
- The verification method at work: Sequence Array Initialization
- Experimental evaluation

Outline of the Talk

- Constraint Logic Programming as a metalanguage for representing
  - the imperative program (integer and array variables)
  - the semantics of the imperative language (i.e., the interpreter)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving unfold/fold rules and strategies
  - VC generation by specialization of the interpreter
  - VC transformation by propagation of the property to be verified
- The verification method at work: Sequence Array Initialization
- Experimental evaluation

Outline of the Talk

- Constraint Logic Programming as a metalanguage for representing
  - the imperative program (integer and array variables)
  - the semantics of the imperative language (i.e., the interpreter)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving unfold/fold rules and strategies
  - VC generation by specialization of the interpreter
  - VC transformation by propagation of the property to be verified
- The verification method at work: Sequence Array Initialization
- Experimental evaluation

CLP with integer and array constraints

- A CLP clause is an implication $\mathtt{c} \wedge \mathtt{G} \rightarrow \mathtt{H}$, written as:

  ```
  H :- c, G.
  ```

  where H is an atom, c is a constraint, and G is a conjunction of atoms
- A constraint is a conjunction of:
  - equalities/inequalities over integers ($p_1 = p_2$, $p_1 \geq p_2$, $p_1 > p_2$)
  - array constraints:
    - $\mathtt{read}(a, i, v)$ (the i-th element of array a is v)
    - $\mathtt{write}(a, i, v, b)$
      (array b is equal to array a except that its i-th element is v)
    - $\mathtt{dim}(a, n)$ (the dimension of a is n)
- A CLP program is a set of CLP clauses
- Semantics: least model of the program with the fixed interpretation of constraints.

CLP with integer and array constraints

- A CLP clause is an implication $c \land G \rightarrow H$, written as:

  `H :- c, G.`

  where H is an atom, c is a constraint, and G is a conjunction of atoms
- A constraint is a conjunction of:
  - equalities/inequalities over integers ($p_1 = p_2$, $p_1 \geq p_2$, $p_1 > p_2$)
  - array constraints:
    - $\mathtt{read}(a, i, v)$ (the i-th element of array a is v)
    - $\mathtt{write}(a, i, v, b)$
      (array b is equal to array a except that its i-th element is v)
    - $\mathtt{dim}(a, n)$ (the dimension of a is n)
- A CLP program is a set of CLP clauses
- Semantics: least model of the program with the fixed interpretation of constraints.

CLP with integer and array constraints

- A CLP clause is an implication $\mathtt{c} \land \mathtt{G} \rightarrow \mathtt{H}$, written as:

  ```
  H :- c, G.
  ```

  where $\mathtt{H}$ is an atom, $\mathtt{c}$ is a constraint, and $\mathtt{G}$ is a conjunction of atoms
- A constraint is a conjunction of:
  - equalities/inequalities over integers $(\mathtt{p}_1 = \mathtt{p}_2,\ \mathtt{p}_1 \geq \mathtt{p}_2,\ \mathtt{p}_1 > \mathtt{p}_2)$
  - array constraints:
    - $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$ (the $\mathtt{i}$-th element of array $\mathtt{a}$ is $\mathtt{v}$)
    - $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$
      (array $\mathtt{b}$ is equal to array $\mathtt{a}$ except that its $\mathtt{i}$-th element is $\mathtt{v}$)
    - $\mathtt{dim}(\mathtt{a}, \mathtt{n})$ (the dimension of $\mathtt{a}$ is $\mathtt{n}$)
- A CLP program is a set of CLP clauses
- Semantics: least model of the program with the fixed interpretation of constraints.

CLP with integer and array constraints

- A CLP clause is an implication $\mathtt{c} \wedge \mathtt{G} \rightarrow \mathtt{H}$, written as:

  ```
  H :- c, G.
  ```

  where H is an atom, c is a constraint, and G is a conjunction of atoms
- A constraint is a conjunction of:
  - equalities/inequalities over integers ($\mathtt{p}_1 = \mathtt{p}_2$, $\mathtt{p}_1 \geq \mathtt{p}_2$, $\mathtt{p}_1 > \mathtt{p}_2$)
  - array constraints:
    - $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$ (the i-th element of array a is v)
    - $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$
      (array b is equal to array a except that its i-th element is v)
    - $\mathtt{dim}(\mathtt{a}, \mathtt{n})$ (the dimension of a is n)
- A CLP program is a set of CLP clauses
- Semantics: least model of the program with the fixed interpretation of constraints.

Running Example: Array Initialization

Program SeqInit

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

An Execution

$[4, \_, \_, \_] \Longrightarrow [4, 5, \_, \_] \Longrightarrow [4, 5, 6, \_] \Longrightarrow [4, 5, 6, 7]$

Partial Correctness Specification

$\{i \geq 0 \ \wedge \ n = dim(a) \ \wedge \ n \geq 1\}$
SeqInit
$\{\forall j \ (0 \leq j \ \wedge \ j+1 < n \ \rightarrow \ a[j] < a[j+1])\}$

Running Example: Array Initialization

Program SeqInit

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

An Execution

$[4, \_, \_, \_] \Longrightarrow [4, 5, \_, \_] \Longrightarrow [4, 5, 6, \_] \Longrightarrow [4, 5, 6, 7]$

Partial Correctness Specification

$\{i \geq 0 \ \wedge \ n = dim(a) \ \wedge \ n \geq 1\}$
SeqInit
$\{\forall j \ (0 \leq j \ \wedge \ j+1 < n \ \rightarrow \ a[j] < a[j+1])\}$

Running Example: Array Initialization

Program SeqInit

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

An Execution

$[4, \_, \_, \_] \Longrightarrow [4, 5, \_, \_] \Longrightarrow [4, 5, 6, \_] \Longrightarrow [4, 5, 6, 7]$

Partial Correctness Specification

$\{i \geq 0 \ \wedge \ n = dim(a) \ \wedge \ n \geq 1\}$
SeqInit
$\{\forall j \ (0 \leq j \ \wedge \ j+1 < n \ \rightarrow \ a[j] < a[j+1])\}$

CLP encoding of imperative programs

Program SeqInit

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

CLP encoding of program SeqInit

A set of at(label, command) facts. while commands are replaced by ite and goto. elem(a,i) stands for a[i].

$\mathtt{at}(\ell_0, \mathtt{asgn}(\mathtt{i}, 1)).$
$\mathtt{at}(\ell_1, \mathtt{ite}(\mathtt{less}(\mathtt{i}, \mathtt{n}), \ell_2, \ell_h)).$
$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{elem}(\mathtt{a}, \mathtt{i}), \mathtt{plus}(\mathtt{elem}(\mathtt{a}, \mathtt{minus}(\mathtt{i}, 1)), 1))).$
$\mathtt{at}(\ell_3, \mathtt{asgn}(\mathtt{i}, \mathtt{plus}(\mathtt{i}, 1))).$
$\mathtt{at}(\ell_4, \mathtt{goto}(\ell_1)).$
$\mathtt{at}(\ell_h, \mathtt{halt}).$

CLP encoding of imperative programs

Program SeqInit

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

CLP encoding of program SeqInit

A set of at(label, command) facts. `while` commands are replaced by `ite` and `goto`. `elem(a,i)` stands for `a[i]`.

$\mathtt{at}(\ell_0, \mathtt{asgn}(\mathtt{i}, 1)).$
$\mathtt{at}(\ell_1, \mathtt{ite}(\mathtt{less}(\mathtt{i}, \mathtt{n}), \ell_2, \ell_\mathsf{h})).$
$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{elem}(\mathtt{a}, \mathtt{i}), \mathtt{plus}(\mathtt{elem}(\mathtt{a}, \mathtt{minus}(\mathtt{i}, 1)), 1))).$
$\mathtt{at}(\ell_3, \mathtt{asgn}(\mathtt{i}, \mathtt{plus}(\mathtt{i}, 1))).$
$\mathtt{at}(\ell_4, \mathtt{goto}(\ell_1)).$
$\mathtt{at}(\ell_\mathsf{h}, \mathtt{halt}).$

CLP encoding of the operational semantics (1)

A transition semantics for the imperative language is defined by:

- a set of configurations, i.e., a CLP term: $\texttt{cf}(\texttt{C}, \texttt{D})$

  where:

  - C is a command
  - D is an environment,
    i.e., a list of [variable identifier, value] pairs:

    $$[[\texttt{int}(\texttt{i}), 4],\ [\texttt{array}(\texttt{a}), [4, 5, 6, 7]]]$$

- a transition relation: $\texttt{tr}(\texttt{cf}(\texttt{C}, \texttt{D}),\ \texttt{cf}(\texttt{C1}, \texttt{D1}))$

The transition for array assignment

- command: $L : a[ie] = e$
- environment: D
- transition:

```
tr(cf(cmd(L,asgn(elem(A,IE),E)),D),     source configuration
   cf(cmd(L1,C),D1)) :-                 target configuration
     eval(IE,D,I),                      evaluate index expr
     eval(E,D,V),                       evaluate expression
     lookup(D,array(A),FA),             get array from env
     write(FA,I,V,FA1),                 update array
     update(D,array(A),FA1,D1),         update environment
     nextlab(L,L1),                     next label
     at(L1,C).                          next command
```

CLP encoding of (in)correctness

Given the Specification $\{\varphi_{init}\}$ *prog* $\{\psi\}$ define $\varphi_{error} \equiv \neg\psi$

Definition (Partial Correctness)

A program *prog* is correct w.r.t. φ_{init} and φ_{error} if
from any initial configuration satisfying φ_{init}
no final configuration satisfying φ_{error} can be reached.
Otherwise, program P is incorrect.

Definition (CLP encoding of incorrectness)

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

`errorConf(X)` $\equiv$ X is a configuration satisfying φ_{error}
`initConf(Y)` $\equiv$ Y is a configuration satisfying φ_{init}

Theorem (Correctness of Encoding)

prog is correct iff `incorrect` $\notin M(Int)$ *(the least model of Int)*

CLP encoding of (in)correctness

Given the Specification $\{\varphi_{init}\}$ *prog* $\{\psi\}$ define $\varphi_{error} \equiv \neg\psi$

Definition (Partial Correctness)

A program *prog* is correct w.r.t. φ_{init} and φ_{error} if
from any initial configuration satisfying φ_{init}
no final configuration satisfying φ_{error} can be reached.
Otherwise, program P is incorrect.

Definition (CLP encoding of incorrectness)

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

`errorConf(X)` $\equiv$ `X` is a configuration satisfying φ_{error}

`initConf(Y)` $\equiv$ `Y` is a configuration satisfying φ_{init}

Theorem (Correctness of Encoding)

prog is correct iff `incorrect` $\notin M(Int)$ *(the least model of Int)*

CLP encoding of (in)correctness

Given the Specification $\{\varphi_{init}\}$ *prog* $\{\psi\}$ define $\varphi_{error} \equiv \neg\psi$

Definition (Partial Correctness)

A program *prog* is correct w.r.t. φ_{init} and φ_{error} if
from any initial configuration satisfying φ_{init}
no final configuration satisfying φ_{error} can be reached.
Otherwise, program P is incorrect.

Definition (CLP encoding of incorrectness)

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

`errorConf(X)` $\equiv$ `X` is a configuration satisfying φ_{error}

`initConf(Y)` $\equiv$ `Y` is a configuration satisfying φ_{init}

Theorem (Correctness of Encoding)

prog is correct iff `incorrect` $\notin M(Int)$ *(the least model of Int)*

Running Example: Array Initialization (Cont'd)

Partial Correctness Specification

$\{i \geq 0 \ \wedge \ n = dim(a) \ \wedge \ n \geq 1\}$ $\qquad \varphi_{init}$

SeqInit

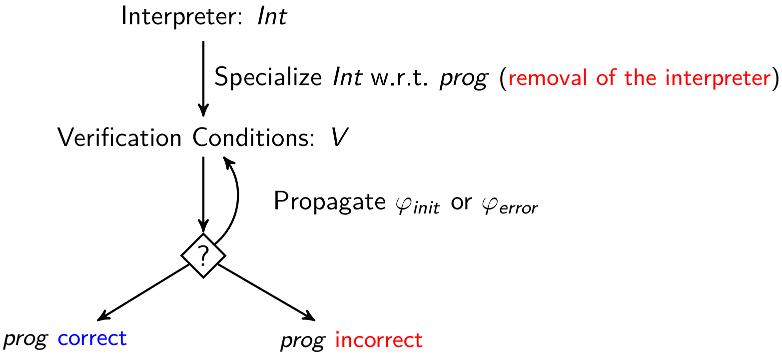
$\{\forall j \ (0 \leq j \ \wedge \ j+1 < n \ \rightarrow \ a[j] < a[j+1])\}$ $\qquad \neg\varphi_{error}$

$\{\exists j \ (0 \leq j \ \wedge \ j+1 < n \ \wedge \ a[j] \geq a[j+1])\}$ $\qquad \varphi_{error}$

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

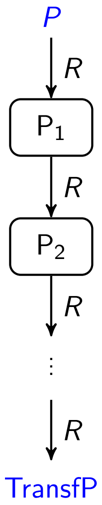
```
phiInit(I,N,A) :- I≥0, dim(A,N), N≥1.
phiError(N,A) :- J≥0, J+1<N, J1=J+1, AJ≥AJ1,
                 read(A,J,AJ), read(A,J1,AJ1).
```

The Transformation-based Verification Method



- *prog* correct if no constrained facts appear in the VCs.
- *prog* incorrect if the fact `incorrect.` appears in the VCs.

Unfold/Fold Program Transformation



- transformation rules:
  $R \in \{$ Definition,
  Unfolding,
  Folding,
  Clause Removal,
  Constraint Replacement$\}$

- the transformation rules preserve the semantics:

  $\boxed{\texttt{incorrect} \in M(\mathrm{P}) \;\; \textit{iff} \;\; \texttt{incorrect} \in M(\mathrm{TransfP})}$

- the rules must be guided by a strategy.

Unfold/Fold Program Transformation

$$\begin{array}{c} P \\ \downarrow R \\ \boxed{P_1} \\ \downarrow R \\ \boxed{P_2} \\ \downarrow R \\ \vdots \\ \downarrow R \\ \text{TransfP} \end{array}$$

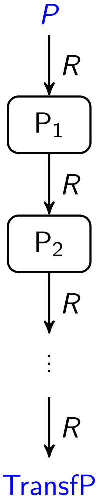
- transformation rules:
  $R \in \{$ Definition,
  Unfolding,
  Folding,
  Clause Removal,
  Constraint Replacement$\}$

- the transformation rules preserve the semantics:

$$\boxed{\texttt{incorrect} \in M(\text{P}) \quad \textit{iff} \quad \texttt{incorrect} \in M(\text{TransfP})}$$

- the rules must be guided by a strategy.

Unfold/Fold Program Transformation



- transformation rules:
  $R \in \{$ Definition,
  Unfolding,
  Folding,
  Clause Removal,
  Constraint Replacement$\}$

- the transformation rules preserve the semantics:

$$\boxed{\texttt{incorrect} \in M(\mathrm{P}) \quad \textit{iff} \quad \texttt{incorrect} \in M(\mathrm{TransfP})}$$

- the rules must be guided by a strategy.

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

$$\mathtt{newp(X)} \ \mathtt{:-}\ \mathtt{c}, \mathtt{A}$$

R2. Unfolding. A symbolic evaluation step (resolution)

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

$\quad\quad \underline{\mathtt{A}}\ \mathtt{:-}\ \mathtt{d_1}, \mathtt{G_1},\ \ldots,\ \underline{\mathtt{A}}\ \mathtt{:-}\ \mathtt{d_m}, \mathtt{G_m}$

derive $\quad \mathtt{H}\ \mathtt{:-}\ \mathtt{c}, \mathtt{d_1}, \mathtt{G_1}, \mathtt{G},\ \ldots,\ \mathtt{H}\ \mathtt{:-}\ \mathtt{c}, \mathtt{d_m}, \mathtt{G_m}, \mathtt{G}$

R3. Folding. Matching a predicate definition (e.g., a loop invariant)

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

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

derive $\quad \mathtt{H}\ \mathtt{:-}\ \mathtt{d}, \mathtt{newp(X)}, \mathtt{G}$

R4. Clause Removal. Removal of clauses with unsatisfiable constraint or subsumed by others

Rules for Transforming CLP Programs

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

$\mathtt{newp(X)} \ \mathtt{:-}\ \mathtt{c}, \mathtt{A}$

R2. Unfolding. A symbolic evaluation step (resolution)

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

$\underline{\mathtt{A}} \ \mathtt{:-}\ \mathtt{d}_1, \mathtt{G}_1,\ \ldots,\ \underline{\mathtt{A}} \ \mathtt{:-}\ \mathtt{d}_\mathtt{m}, \mathtt{G}_\mathtt{m}$

derive $\mathtt{H} \ \mathtt{:-}\ \mathtt{c}, \mathtt{d}_1, \mathtt{G}_1, \mathtt{G},\ \ldots,\ \mathtt{H} \ \mathtt{:-}\ \mathtt{c}, \mathtt{d}_\mathtt{m}, \mathtt{G}_\mathtt{m}, \mathtt{G}$

R3. Folding. Matching a predicate definition (e.g., a loop invariant)

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

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

derive $\mathtt{H} \ \mathtt{:-}\ \mathtt{d}, \mathtt{newp(X)}, \mathtt{G}$

R4. Clause Removal. Removal of clauses with unsatisfiable constraint or subsumed by others

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

$\mathtt{newp(X)\ :\!-\ c, A}$

R2. Unfolding. A symbolic evaluation step (resolution)

given $\mathtt{H\ :\!-\ c,\ \underline{A},\ G}$

$\mathtt{\underline{A}\ :\!-\ d_1, G_1,\ \ \ldots,\ \ \underline{A}\ :\!-\ d_m, G_m}$

derive $\mathtt{H\ :\!-\ c, d_1, G_1, G,\ \ \ldots,\ \ H\ :\!-\ c, d_m, G_m, G}$

R3. Folding. Matching a predicate definition (e.g., a loop invariant)

given $\mathtt{H\ :\!-\ d, \underline{A}, G}$

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

derive $\mathtt{H\ :\!-\ d, newp(X), G}$

R4. Clause Removal. Removal of clauses with unsatisfiable constraint or subsumed by others

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

$\mathtt{newp(X)\ :\!-\ c, A}$

R2. Unfolding. A symbolic evaluation step (resolution)

given $\mathtt{H\ :\!-\ c,\ \underline{A},\ G}$

$\mathtt{\underline{A}\ :\!-\ d_1, G_1,\ \ \ldots,\ \ \underline{A}\ :\!-\ d_m, G_m}$

derive $\mathtt{H\ :\!-\ c, d_1, G_1, G,\ \ \ldots,\ \ H\ :\!-\ c, d_m, G_m, G}$

R3. Folding. Matching a predicate definition (e.g., a loop invariant)

given $\mathtt{H\ :\!-\ d, \underline{A}, G}$

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

derive $\mathtt{H\ :\!-\ d, newp(X), G}$

R4. Clause Removal. Removal of clauses with unsatisfiable constraint or subsumed by others

R5. Constraint Replacement :

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 with dimension

Then replace `H :- c0, d, G`
by `H :- c1, d, G, ..., H :- cn, d, G`

The Unfold/Fold Transformation strategy

Transform(P)

TransfP $= \emptyset$;
Defs $= \{$`incorrect :- errorConf(X), reach(X)`$\}$;
while $\exists q \in$ Defs **do**
  Cls $=$ Unfold(q);
  Cls $=$ ConstraintReplacement(Cls);
  Cls $=$ ClauseRemoval(Cls);
  Defs $=$ (Defs $-$ $\{q\}$) $\cup$ Define(Cls);
  TransfP $=$ TransfP $\cup$ Fold(Cls, Defs);
od

Theorem (Correctness of the Transformation Strategy)

`incorrect` $\in M(\mathrm{P})$ *iff* `incorrect` $\in M(\mathrm{TransfP})$

The Unfold/Fold Transformation strategy

Transform(P)

TransfP $= \emptyset$;
Defs $= \{$`incorrect :- errorConf(X), reach(X)`$\}$;
while $\exists q \in$ Defs **do**
  Cls $=$ Unfold(q);
  Cls $=$ ConstraintReplacement(Cls);
  Cls $=$ ClauseRemoval(Cls);
  Defs $=$ (Defs $- \{q\}) \cup$ Define(Cls);
  TransfP $=$ TransfP $\cup$ Fold(Cls, Defs);
od

Theorem (Correctness of the Transformation Strategy)

`incorrect` $\in M$(P) *iff* `incorrect` $\in M$(TransfP)

Generating Verification Conditions via Specialization

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

- `tr` (i.e., the operational semantics of the imperative language)
- `at` (i.e., the encoding of *prog*)

The Specialized Interpreter for SeqInit (Verification Conditions *V*)

```
incorrect :- J≥0, J+1<N, J1=J+1, AJ≥AJ1, N≤I,
             read(A,J,AJ), read(A,J1,AJ1), p(I,N,A).
p(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), p(I,N,A).
p(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.

Generating Verification Conditions via Specialization

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

- `tr` (i.e., the operational semantics of the imperative language)
- `at` (i.e., the encoding of *prog*)

The Specialized Interpreter for SeqInit (Verification Conditions *V*)

$\texttt{incorrect} \texttt{ :- } J \geq 0,\ J+1 < N,\ J1 = J+1,\ AJ \geq AJ1,\ N \leq I,$
$\quad \texttt{read}(A, J, AJ), \texttt{read}(A, J1, AJ1),\ \texttt{p}(I, N, A).$
$\texttt{p}(I1, N, B) \texttt{ :- } 1 \leq I,\ I < N,\ D = I-1,\ I1 = I+1,\ V = U+1,$
$\quad \texttt{read}(A, D, U),\ \texttt{write}(A, I, V, B),\ \texttt{p}(I, N, A).$
$\texttt{p}(I, N, A) \texttt{ :- } I = 1,\ N \geq 1.$

- 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.

Generating Verification Conditions via Specialization

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

- `tr` (i.e., the operational semantics of the imperative language)
- `at` (i.e., the encoding of *prog*)

The Specialized Interpreter for SeqInit (Verification Conditions *V*)

```
incorrect :- J≥0, J+1<N, J1=J+1, AJ≥AJ1, N≤I,
             read(A,J,AJ), read(A,J1,AJ1), p(I,N,A).
p(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), p(I,N,A).
p(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.

Propagating the Error Property by Unfold/Fold Transformation

The Unfold/Fold transformation strategy propagates the error property with the goal of

- either removing all constrained facts from V
- or deriving the fact `incorrect`.

The Output of the U/F Strategy for *SeqInit*

incorrect :- J1 = J+1, J ≥ 0, J1 < I, AJ ≥ AJ1, D = I−1, N = I+1, Y = X+1,
    read(A, J, AJ), read(A, J1, AJ1), read(A, D, X), write(A, I, Y, B),
    new1(I, N, A).
new1(I1, N, B) :- I1 = I+1, Z = W+1, Y = X+1, D = I−1, N ≤ I+2,
    I ≥ 1, Z < I, Z ≥ 1, N > I, U ≥ V, read(A, W, U), read(A, Z, V),
    read(A, D, X), write(A, I, Y, B), new2(I, N, A).
new2(I1, N, B) :- I1 = I+1, Z = W+1, Y = X+1, D = I−1, I ≥ 1,
    Z < I, Z ≥ 1, N > I, U ≥ V, read(A, W, U), read(A, Z, V),
    read(A, D, X), write(A, I, Y, B), new2(I, N, A).

No constrained facts: the program *SeqInit* is correct.

Propagating the Error Property by Unfold/Fold Transformation

The Unfold/Fold transformation strategy propagates the error property with the goal of

- either removing all constrained facts from V
- or deriving the fact `incorrect`.

The Output of the U/F Strategy for *SeqInit*

$\texttt{incorrect} \texttt{ :- } \mathtt{J1=J+1,\ J\geq 0,\ J1<I,\ AJ\geq AJ1,\ D=I-1,\ N=I+1,\ Y=X+1,}$
$\quad \mathtt{read(A,J,AJ),\ read(A,J1,AJ1),\ read(A,D,X),\ write(A,I,Y,B),}$
$\quad \mathtt{new1(I,N,A).}$

$\mathtt{new1(I1,N,B)} \texttt{ :- } \mathtt{I1=I+1,\ Z=W+1,\ Y=X+1,\ D=I-1,\ N\leq I+2,}$
$\quad \mathtt{I\geq 1,\ Z<I,\ Z\geq 1,\ N>I,\ U\geq V,\ read(A,W,U),\ read(A,Z,V),}$
$\quad \mathtt{read(A,D,X),\ write(A,I,Y,B),\ new2(I,N,A).}$

$\mathtt{new2(I1,N,B)} \texttt{ :- } \mathtt{I1=I+1,\ Z=W+1,\ Y=X+1,\ D=I-1,\ I\geq 1,}$
$\quad \mathtt{Z<I,\ Z\geq 1,\ N>I,\ U\geq V,\ read(A,W,U),\ read(A,Z,V),}$
$\quad \mathtt{read(A,D,X),\ write(A,I,Y,B),\ new2(I,N,A).}$

No constrained facts: the program *SeqInit* is correct.

Rewrite rules for Constraint Replacement based on the Theory of Arrays:

Array congruence

(AC) $\mathtt{I}=\mathtt{J},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}) \ \rightarrow\ \mathtt{U}=\mathtt{V}$

[AC1] replace: $\mathtt{I}=\mathtt{J},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$
by: $\mathtt{I}=\mathtt{J},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{U}=\mathtt{V}$

[AC2] replace: $\mathtt{U}\neq\mathtt{V},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$
by: $\mathtt{U}\neq\mathtt{V},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}),\ \mathtt{I}\neq\mathtt{J}$

Auxiliary Strategies (2)

Read-over-Write

(RoW1) $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V})\ \rightarrow\ \mathtt{U} = \mathtt{V}$

(RoW2) $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V})\ \rightarrow\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V})$

[RoW1] replace: $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V})$
by: $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{U} = \mathtt{V}$

[RoW2] replace: $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V})$
by: $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V})$

[RoW12] replace: $\mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V})$
by: $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{U} = \mathtt{V}$
and $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V})$

- The most critical transformation step within the unfold/fold transformation strategy is the introduction of new predicate definitions to be used for folding.

- Given `p(X) :- c(X,Y), q(Y).`

  Introduce `newp(Y) :- d(Y), q(Y).`

  where $c(X,Y) \to d(Y)$ (`d(Y)` is a generalization of `c(X,Y)`)

  and fold: `p(X) :- c(X,Y), newp(Y).`

- Generalization strategies based on widening and convex-hull of linear constraints.

- The most critical transformation step within the unfold/fold transformation strategy is the introduction of new predicate definitions to be used for folding.
- Given `p(X) :- c(X,Y), q(Y).`

  Introduce `newp(Y) :- d(Y), q(Y).`

  where $\mathtt{c}(\mathtt{X},\mathtt{Y}) \rightarrow \mathtt{d}(\mathtt{Y})$ (`d(Y)` is a generalization of `c(X,Y)`)

  and fold: `p(X) :- c(X,Y), newp(Y).`
- Generalization strategies based on widening and convex-hull of linear constraints.

- The most critical transformation step within the unfold/fold transformation strategy is the introduction of new predicate definitions to be used for folding.

- Given `p(X) :- c(X,Y), q(Y).`

  Introduce `newp(Y) :- d(Y), q(Y).`

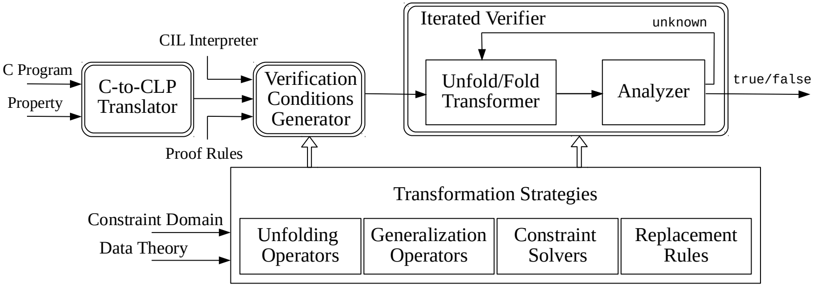
  where $\mathtt{c}(\mathtt{X},\mathtt{Y}) \rightarrow \mathtt{d}(\mathtt{Y})$ (`d(Y)` is a generalization of `c(X,Y)`)

  and fold: `p(X) :- c(X,Y), newp(Y).`

- Generalization strategies based on widening and convex-hull of linear constraints.

VeriMAP

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



Experimental evaluation

Program	Gen_W	Gen_{WD}	Gen_S	Gen_{SD}
init	*unknown*	0.06	0.10	0.08
init-partial	*unknown*	0.06	0.07	0.08
init-non-constant	*unknown*	0.06	0.22	0.22
init-sequence	*unknown*	0.80	*unknown*	1.20
copy	*unknown*	0.27	0.33	0.29
copy-partial	*unknown*	0.29	0.34	0.34
copy-reverse	*unknown*	0.27	0.46	0.45
max	*unknown*	0.31	0.24	0.33
sum	*unknown*	0.68	1.14	1.12
difference	*unknown*	0.66	1.15	1.11
find	0.25	0.43	0.46	0.45
first-not-null	0.38	0.41	0.42	0.42
find-first-non-null	1.24	1.87	1.94	1.93
partition	0.06	0.11	0.14	0.12
insertionsort-inner	0.21	0.26	0.45	0.43
bubblesort-inner	2.46	2.71	2.45	2.75
selectionsort-inner	7.20	6.40	7.23	7.16
precision	7	17	16	17
total time	11.80	15.65	17.14	18.48
average time	1.69	0.92	1.07	1.09

What Can Transformation do for Verification?

- Help build a verification framework which is parametric with respect to:
  - programming language and its operational semantics
  - properties and proof rules
  - theory of data structures
- PT can be used both for generating VCs (in the form of CLP) and for proving their satisfiability
- The input and the output of PT are semantically equivalent CLP programs. This allows:
  - incremental verification
  - iteration for refining verification
  - easy interoperation with other verifiers that use Horn clause format

Future Work

- More experiments (e.g., nested loops)
- Recursive functions
- More theories (lists, heaps, etc.)
- Other programming languages, properties, proof rules

Thanks for your attention!