Program Verification using Templates over Predicate Abstraction

Saurabh Srivastava and Sumit Gulwani

```
ArrayInit(Array A, int n)
    i := 0;
    while (i < n)
        A[i] := 0;
        i := i + 1;
    Assert(∀j: 0≤j<n ⇒ sel(A,j)=0);
```

Invariant Templates

Instead of supplying concrete invariants, supply invariant templates with *holes* for predicates.

```
ArrayInit(Array A, int n)
    true
    i := 0;
    ∀j: ?? ⇒ sel(A,j)=0
    while (i < n)
        A[i] := 0;
        i := i + 1;
    ∀j: 0≤j<n ⇒ sel(A,j)=0
```

```
ArrayInit(Array A, int n)
      true
      i := 0;
      ∀j: ?? ⇒ sel(A,j)=0
      while (i < n)
            A[i] := 0;
            i := i + 1;
      ∀j: 0≤j<n ⇒ sel(A,j)=0
```

?? = subset of $\{j\ op\ x \mid x \in \{0,i,n\}, op \in \{=,<,\leq,>,\geq\}\}$

Invariant Templates

Instead of supplying concrete invariants, supply invariant templates with *holes* for predicates.

Example predicates:

`i≤0`, `sel(A,j)=x`, `j>k`

Template Language:

$$\tau \quad ::= \quad v \quad | \quad \neg\tau \quad | \quad \tau_1 \vee \tau_2 \quad | \quad \tau_1 \wedge \tau_2 \quad | \quad \exists x : \tau \quad | \quad \forall x : \tau$$

Holes are *conjunctions* of predicates, usually notated as a set

1. $true$

   $i := 0;$

   $\forall j:\ v \Rightarrow sel(A,j)=0$

2. $\forall j:\ v \Rightarrow sel(A,j)=0$

   $Assume(i \geq n);$

   $\forall j:\ 0 \leq j < n \Rightarrow sel(A,j)=0$

3. $\forall j:\ v \Rightarrow sel(A,j)=0$

   $Assume(i < n);$

   $A' := upd(A,i,0);$

   $i' := i + 1;$

   $\forall j:\ v \Rightarrow sel(A,j)=0$

Algorithm for Least (Greatest) Fixed Point

- Start with maximally strong (weak) guess for assignment of holes to predicates.
- Until you have a valid candidate assignment, repeat:
  - Choose one of the guesses and one path (a *path* is a precondition, post-condition, and section of straight-line code)
  - Replace the guess with new guesses that have weaker postconditions (stronger preconditions) for that path.

1. *true*
   `i := 0;`
   ∀j: v ⇒ sel(A,j)=0

2. ∀j: v ⇒ sel(A,j)=0
   `Assume(i ≥ n);`
   ∀j: 0≤j<n ⇒ sel(A,j)=0

3. ∀j: v ⇒ sel(A,j)=0
   `Assume(i < n);`
   `A' := upd(A,i,0);`
   `i' := i + 1;`
   ∀j: v ⇒ sel(A,j)=0

Initial: Guess: v={}

1. $true$
   $i := 0;$
   $\forall j:\ true \Rightarrow sel(A,j)=0$

2. $\forall j:\ true \Rightarrow sel(A,j)=0$
   $Assume(i \geq n);$
   $\forall j:\ 0 \leq j < n \Rightarrow sel(A,j)=0$

3. $\forall j:\ true \Rightarrow sel(A,j)=0$
   $Assume(i < n);$
   $A' := upd(A,i,0);$
   $i' := i + 1;$
   $\forall j:\ true \Rightarrow sel(A,j)=0$

Initial: Guess: v={}

1. $true$
   $i := 0;$
   $\forall j:\ true \Rightarrow sel(A,j)=0$

2. $\forall j:\ true \Rightarrow sel(A,j)=0$
   $Assume(i \geq n);$
   $\forall j:\ 0 \leq j < n \Rightarrow sel(A,j)=0$

3. $\forall j:\ true \Rightarrow sel(A,j)=0$
   $Assume(i < n);$
   $A' := upd(A,i,0);$
   $i' := i + 1;$
   $\forall j:\ true \Rightarrow sel(A,j)=0$

Initial: Guess: v={}

Select initial guess and inconsistent path (1).

1. *true*
   $i := 0;$
   $\forall j:\ true \Rightarrow sel(A,j)=0$

2. $\forall j:\ true \Rightarrow sel(A,j)=0$
   $Assume(i \geq n);$
   $\forall j:\ 0 \leq j < n \Rightarrow sel(A,j)=0$

3. $\forall j:\ true \Rightarrow sel(A,j)=0$
   $Assume(i < n);$
   $A' := upd(A,i,0);$
   $i' := i + 1;$
   $\forall j:\ true \Rightarrow sel(A,j)=0$

Initial: Guess: v={}

Select initial guess and inconsistent path (1).

Update with 4 new optimal guesses:

v={0 < j, j ≤ i}; v={0 ≤ j, j < i}; v={i < j, j ≤ 0}, v={i ≤ j, j < 0}

1. $true$

   $i := 0;$

   $\forall j: 0 \le j < i \Rightarrow sel(A,j)=0$

2. $\forall j: 0 \le j < i \Rightarrow sel(A,j)=0$

   $Assume(i \ge n);$

   $\forall j: 0 \le j < n \Rightarrow sel(A,j)=0$

3. $\forall j: 0 \le j < i \Rightarrow sel(A,j)=0$

   $Assume(i < n);$

   $A' := upd(A,i,0);$

   $i' := i + 1;$

   $\forall j: 0 \le j < i \Rightarrow sel(A,j)=0$

Initial: Guess: v={}

Select initial guess and inconsistent path (1).

Update with 4 new optimal guesses:

v={0 < j, j ≤ i}; v={0 ≤ j, j < i}; v={i < j, j ≤ 0}, v={i ≤ j, j < 0}

One of these is valid (v={0 ≤ j, j < i}), so we're done.

Optimal Guesses

A valid solution (assignment of holes to sets of predicates) for a particular path is *optimal* if removing (adding) predicates to any negative (positive) hole results in an invalid solution.

- Naïve search extremely expensive
- They use a complex algorithm (including an SMT solver) to find these in practice.

```
LeastFixedPoint(Prog, Q)
1   Let σ0 be s.t. σ0(v) ↦ ∅, if v is negative
                   σ0(v) ↦ Q(v), if v is positive
2   S := {σ0};
3   while S ≠ ∅ ∧ ∀σ ∈ S : ¬Valid(VC(Prog, σ))
4      Choose σ ∈ S, (δ, τ1, τ2, σt) ∈ Paths(Prog) s.t.
                          ¬Valid(VC(⟨τ1σ, δ, τ2σσt⟩))
5      S := S − {σ};
6      Let σp = σ | U(Prog)−U(τ2) and θ := τ2σ ⇒ τ2.
7      S := S ∪ {σ'σt^−1 ∪ σp |
             σ' ∈ OptimalSolutions{VC(⟨τ1σ, δ, τ2⟩) ∧ θ, Qσt)}
8   if S = ∅ return ''No solution''
9   else return σ ∈ S s.t. Valid(VC(Prog, σ))
```

(a) Least Fixed Point Computation

Maximally strong initial guess

Weaken precondition

```
GreatestFixedPoint(Prog)
1   Let σ0 be s.t. σ0(v) ↦ Q(v), if v is negative
                   σ0(v) ↦ ∅, if v is positive
2   S := {σ0};
3   while S ≠ ∅ ∧ ∀σ ∈ S : ¬Valid(VC(Prog, σ))
4      Choose σ ∈ S, (δ, τ1, τ2, σt) ∈ Paths(Prog) s.t.
                          ¬Valid(VC(⟨τ1σ, δ, τ2σσt⟩))
5      S := S − {σ};
6      Let σp = σ | U(Prog)−U(τ1) and θ := τ1 ⇒ τ1σ.
7      S := S ∪ {σ' ∪ σp |
             σ' ∈ OptimalSolutions{VC(⟨τ1, δ, τ2σσt⟩) ∧ θ, Q)}
8   if S = ∅ return ''No solution''
9   else return σ ∈ S s.t. Valid(VC(Prog, σ))
```

(b) Greatest Fixed Point Computation

Maximally weak initial guess

Strengthen Postcondition

3rd Algorithm: Encode as SMT

- Boolean variables for every hole-predicate pair.
- Formulae for whether a particular path is valid for each guess
  - Need some tricks to avoid explosion of clauses

1. $true$

   $i := 0;$

   $\forall j:\ v \Rightarrow sel(A,j)=0$

2. $\forall j:\ v \Rightarrow sel(A,j)=0$

   $Assume(i \geq n);$

   $\forall j:\ 0 \leq j < n \Rightarrow sel(A,j)=0$

3. $\forall j:\ v \Rightarrow sel(A,j)=0$

   $Assume(i < n);$

   $A' := upd(A,i,0);$

   $i' := i + 1;$

   $\forall j:\ v \Rightarrow sel(A,j)=0$

Benchmark	LFP	GFP	CFP	Previous
Consumer Producer	0.45	2.27	4.54	45.00 [17]
Partition Array	2.28	0.15	0.76	7.96 [17], 2.4 [2]
List Init	0.15	0.06	0.15	24.5 [12]
List Delete	0.10	0.03	0.19	20.5 [12]
List Insert	0.12	0.30	0.25	23.9 [12]

Table 4. Time (secs) for verification of data-sensitive array/list programs.

Benchmark	GFP
Partial Init	0.50
Init Synthesis	0.72
Binary Search	13.48
Merge	3.37

Table 5. Time (secs) for preconditions for functional correctness.

	Sortedness				∀∃			Upper
Benchmark	LFP	GFP	CFP	Previous	LFP	GFP	CFP	Bound
Selection Sort	1.32	6.79	12.66	na[4]	22.69	17.02	timeout	16.62
Insertion Sort	14.16	2.90	6.82	5.38 [15][4]	2.62	94.42	19.66	39.59
Bubble Sort (n^2)	0.47	0.78	1.21	na	5.49	1.10	13.74	0.00
Bubble Sort (flag)	0.22	0.16	0.55	na	1.98	1.56	10.44	9.04
Quick Sort (inner)	0.43	4.28	1.10	42.2 [12]	1.89	4.36	1.83	1.68
Merge Sort (inner)	2.91	2.19	4.92	334.1 [12]	timeout	7.00	23.75	0.00

Table 6. Time (secs) for sorting programs. We verify sortedness, preservation (∀∃) and infer preconditions for the worst case upper bounds.

Contributions

- Can handle ∀∃ invariants
- Multiple algorithms each with separate strengths
- Technique to infer maximally weak preconditions

Discussion

- Scalability to larger programs
- Difficulty of creating templates
- Difficulty of producing specification

Daikon

dynamic detection of likely invariants

What invariants are detected?

- `x.field > abs(y)`
- `y = 2*x+3`
- `array a is sorted`
- `for all list objects lst, lst.next.prev = lst`
- `for all treenode objects n, n.left.value < n.right.value`
- `p != null => p.content in myArray'`

Variables

- pre-state values
- results of side-effect-free methods
- derived variables
  - introduced in subsequent pass after inference is performed on existing variables
  - unary (e.g., length, min, max, sum)
  - binary (e.g., subscript, concat, intersection)
  - ternary (only arbitrary subsequence)

Where are they detected?

- procedure boundaries
- object invariants
- what limitations does this place on the analysis? Do we care?

How are they detected?

- checks potential invariants against runtime values
  - computes probability observation would occur by chance
  - appears to not always do this in practice
- influenced by provided traces
  - when is this acceptable (even preferred)?
  - when is this insufficient?

How does this scale?

- Reduce redundancy (improves performance)
  - avoid introducing redundant derived variables
  - suppress weaker invariants
    - no theorem proving, just templates
- Prune output (user experience)
  - some redundant invariants are caught afterwards
  - abstract data types

Who wants this?

- Wide array of uses
- Highly extensible
- What was convincing?

Comparative Discussion

- What are the relative strengths of each approach?
- Who wants to use each approach?