Program Verification and Synthesis using Templates over Predicate Abstraction

Saurabh Srivastava #
Sumit Gulwani *
Jeffrey S. Foster #
# University of Maryland, College Park
* Microsoft Research, Redmond
Selection Sort:

```
find min

Sorted
```
Selection Sort:

for i = 0..n
    { for j = i..n
        { find min index
        } if (min != i)
            swap A[i] with A[min]
    }

Sorted
Selection Sort:

```java
for i = 0..n {
    for j = i..n {
        find min index
    }
    if (min != i)
        swap A[i] with A[min]
}
```

Prove the following:
- Sortedness
  - Output array is non-decreasing
- Permutation
  - Output array contains all elements of the input
  - Output array contains only elements from the input
  - Number of copies of each element is the same

∀k₁,k₂ : 0≤k₁<k₂<n ⇒ A[k₁] ≤ A[k₂]
∀k∃j : 0≤k<n ⇒ A₀[k] = A[j] & 0≤j<n
∀k∃j : 0≤k<n ⇒ A[k] = A₀[j] & 0≤j<n
∀n : n∈A ⇒ count(A,n) = count(A₀,n)
Selection Sort:

\[
\begin{align*}
i &:= 0 \\
i &< n \\
j &:= i \\
j &< n \\
\text{Output array} \\
\text{Find min index} \\
\text{if } (\text{min } \neq i) \\
\text{swap } A[i], A[\text{min}] \\
\end{align*}
\]
We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about *simple paths*.

**Verification: Proving Programs Correct**

**Selection Sort:**

- **Output state**
  - Sortedness Permutation

- **Input state**
  - true

**Output array**

- **Find min index**
  - if (min != i)
    - swap A[i], A[min]

**i := 0**

**I_{outer}**

**I_{inner}**
We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.
Verification: Proving Programs Correct

We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.

Selection Sort:

\[ i := 0 \]

\[ j := i \]

\[ i < n \]

\[ j < n \]

\[ \text{if } (\text{min } \neq i) \]

\[ \text{swap } A[i], A[\text{min}] \]

Output state

Sortedness

Permutation

Input state

true state

\[ I_{\text{outer}} \]

\[ I_{\text{inner}} \]
Verifying Programs Correct: Selection Sort

We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.

Output state
Sortedness Permutation

Input state
true

Selection Sort:
- i := 0
- j := i
- j < n
- if (min != i)
  - swap A[i], A[min]

Output array
Find min index

The difficult task is program state (invariant) inference:
- I_inner, I_outer, Input state, Output state

Verification: Proving Programs Correct

Sortedness
Permutation

Output state
Input state

I_outer
I_inner

We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.
Verification: Proving Programs Correct

We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.

Output state

Selection Sort:

Input state

true state

Output array

Find min index

if (min != i)

swap A[i], A[min]

∀k₁,k₂ : 0 ≤ k₁ < k₂ < n ∧ k₁ < i ⇒ A[k₁] ≤ A[k₂]

∀k₁,k₂ : 0 ≤ k₁ < k₂ < n ∧ k₁ < i ⇒ A[k₁] ≤ A[k₂]

∀k : 0 ≤ k ∧ i ≤ k < j ⇒ A[k] ≤ A[i]

The difficult task is program state (invariant) inference: I_{inner}, I_{outer}, Input state, Output state.
Verification: Proving Programs Correct

We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.

Selection Sort:

Output state

Input state

Sortedness Permutation

Output array

Find min index

if (min != i) swap A[i], A[min]

∀k₁,k₂ : 0 ≤ k₁,k₂ < n ∧ k₁ < i ⇒ A[k₁] ≤ A[k₂]

∀k₁,k₂ : 0 ≤ k₁,k₂ < n ∧ k₁ < i ⇒ A[k₁] ≤ A[k₂]

∀k₁,k₂ : 0 ≤ k₁,k₂ < n ∧ k₁ < i ⇒ A[k₁] ≤ A[k₂]

∀k₁,k₂ : 0 ≤ k₁,k₂ < n ∧ k₁ < i ⇒ A[k₁] ≤ A[k₂]

The difficult task is program state (invariant) inference: I_inner, I_outer, Input state, Output state
We know from work in the '70s (Hoare, Floyd, Dijkstra) that it is easy to reason logically about simple paths.

The difficult task is program state (invariant) inference: $I_{\text{inner}}, I_{\text{outer}}, \text{Input state, Output state}$
Inferring program state (invariants)

- Full functional correctness
- Program properties that we can verify/prove

User Input

- No input
- Fully specified invariants

Graph:
- Good
- Tolerable
- Bad
Inferring program state (invariants)

- Full functional correctness
- Program properties that we can verify/prove
- No input
- Fully specified invariants

Mind-blowing

Live with current technology

Undesirable

User Input
Inferring program state (invariants)

Full functional correctness

Program properties that we can verify/prove

No input

User Input

Fully specified invariants

Arbitrary quantification and arbitrary abstraction

Alternating quantification ($\forall \exists$)

Quantified facts ($\forall$)

Quantifier-free facts

User Input
Inferring program state (invariants)

Program properties that we can verify/prove

- Full functional correctness
- Undecidable

User Input
- No input
- Fully specified invariants

- Quantifier-free facts
- Quantified facts (∀)
- Alternating quantification (∀ ∃)
- Arbitrary quantification and arbitrary abstraction
Inferring program state (invariants)

Program properties that we can verify/prove

- Full functional correctness
- No input
- User Input

- Quantifier-free facts
- Quantified facts ($\forall \exists$)

- Abstract interpretation
- Quantifier-free domain
- Cousot etc

- Constraint-based
- Quantifier-free domain
- Gulwani, Beyer etc

- Best-effort inference for alternating quantification ($\forall \exists$)
- Kovacs etc

- Quantifier-free domain lifted to $\forall$
- Gulwani etc

- Shape analysis
- SLAyer etc

- Validation using interactive theorem proving
- Jahob

- Validating program paths using non-interactive theorem provers/SMT solvers
- HAVOC, Spec#, etc

- Arbitrary quantification and arbitrary abstraction
- Alternating quantification ($\forall \exists$)

- Fully specified invariants
Inferring program state (invariants)

- Program properties that we can verify/prove
  - Full functional correctness
  - No input
  - User Input

- Magical technique !!

- Fully specified invariants
  - Arbitrary quantification and arbitrary abstraction
  - Alternating quantification ($\forall \exists$)
  - Quantified facts ($\forall$)
  - Quantifier-free facts

- Inferring program state (invariants)
Inferring program state (invariants)

Program properties that we can verify/prove

Full functional correctness

User Input

- No input
- Fully specified invariants

Template-based invariant inference

- Arbitrary quantification and arbitrary abstraction
- Alternating quantification (\(\exists \forall\))
- Quantified facts (\(\forall\))
- Quantifier-free facts

Fully specified invariants

Inferring program state (invariants)
Inferring program state (invariants)

My tasks in this talk:

Convince you that these small hints go a long way in what we can achieve with verification

SMT/SAT technology greatly facilitates fixed-point inference

Example:
If you wanted to prove sortedness, i.e. each element is smaller than some others:
\[ \forall k_1, k_2 : (-) \Rightarrow (-) \]

If you wanted to prove permutation, i.e. for each element there exists another:
\[ \forall k_1 \exists k_2 : (-) \Rightarrow (-) \]
Key facilitators: (1) Templates

- Templates
  \[ \forall k_1, k_2 : (-) \Rightarrow (-) \]
  \[ \forall k_1 \exists k_2 : (-) \Rightarrow (-) \]

  - Intuitive
    - Depending on the property being proved; form straight-forward

  - Simple
    - Limited to the quantification and disjunction
    - No complicated error reporting, iterative refining required
    - Very different from full invariants used by some techniques

  - Helps: We don't have to define insane joins etc.
    - Previous approaches: Domain $\mathbb{D}$ for the holes, construct $\mathbb{D}_\forall$
    - This approach: Work with only $\mathbb{D}$
    - Catch: Holes have different status: $\forall k_1, k_2 : (-) \Rightarrow (-)$
Key facilitators: (2) Predicates

- **Predicate Abstraction**
  - Simple
    - Small fact hypothesis
      - Nested deep under, the facts for the holes are simple
    - E.g., \( \forall k_1, k_2: 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2] \)
      - Prefs \( 0 \leq k_1, k_1 < k_2, k_2 < n, A[k_1] \leq A[k_2] \) are individually simple (but possibly many)
  - Intuitive
    - Only program variables, array elements and relational operators
  - Example: Enumerate predicates of the form:
    \[ x \; op_r \; (y \; op_a \; c) \]
    - E.g., \( \{i < j, i > j, i \leq j, i \geq j, i < j-1, i > j+1 \ldots \} \)
• **VS³:** Verification and Synthesis using SMT Solvers

• **User input**
  – Templates
  – Predicates

• **Set of algorithms for fixed-point computation**

• **Tool implementing the algorithms**

• **Infers**
  – Loop invariants
  – Weakest preconditions
  – Strongest postconditions
Outline

• Three fixed-point inference algorithms
  – Iterative fixed-point
    • Least Fixed-Point (LFP)
    • Greatest Fixed-Point (GFP)
  – Constraint-based (CFP)

• Weakest Precondition Generation

• Program Synthesis
Outline

• Three fixed-point inference algorithms
  – Iterative fixed-point
    • Least Fixed-Point (LFP)
    • Greatest Fixed-Point (GFP)
  – Constraint-based (CFP)

• Weakest Precondition Generation

• Program Synthesis
Analysis Setup

Loop headers (with invariants) split program into simple paths. Simple paths induce program constraints (verification conditions).

E.g. Selection Sort:

Output sorted array

if (min != i)
   swap A[i], A[min]

true ∧ i=0 ⇒ I₁

i:=0

i≠n

j:=i

j≠n

vc(I₂,I₂)

vc(I₂,I₁)

vc(I₁,I₂)

vc(I₁,I₁)

vc(pre,I₁)

pre

I₁

I₂

vc(I₁,post)

post

I₁

I₂

vc(I₂,I₂)

vc(I₂,I₁)
Analysis Setup

Loop headers (with invariants) split program into simple paths. Simple paths induce program constraints (verification conditions).

E.g. Selection Sort:

Output sorted array

if (min != i)
swap A[i], A[min]

true ∧ i=0 ⇒ I_1

I_1 ∧ i<n ∧ j=i ⇒ I_2

I_1 ∧ i<n ∧ j=i ⇒ I_2

I_1 ∧ i<n ∧ j=i ⇒ I_2

I_1 ∧ i<n ∧ j=i ⇒ I_2
Loop headers (with invariants) split program into simple paths. Simple paths induce program constraints (verification conditions).

E.g. Selection Sort:

\[ i := 0 \]

true \land i = 0 \Rightarrow I_1

I_1 \land i < n \land j = i \Rightarrow I_2

i < n

j := i

\[ j < n \]

Check min index

if (min \neq i)

swap A[i], A[\text{min}]

I_1 \land i \geq n \Rightarrow \text{sorted array}
Loop headers (with invariants) split program into simple paths. Simple paths induce program constraints (verification conditions).

\[
\text{true} \land i=0 \Rightarrow I_1 \\
I_1 \land i<n \land j=i \Rightarrow I_2 \\
I_1 \land i\geq n \Rightarrow \text{sorted array}
\]
Analysis Setup

Loop headers (with invariants) split program into simple paths. Simple paths induce program constraints (verification conditions).

true ∧ i=0 => I_1

I_1 ∧ i≤n ∧ j=i => I_2

∀ k_1, k_2 : 0 ≤ k_1 < k_2 < n => A[k_1] ≤ A[k_2]

Simple FOL formulae over I_1, I_2! We will exploit this.
Iterative Fixed-point: Overview

- Approx. for invariants:
  - $\langle \Phi_1, \Phi_2 \rangle \rightarrow \{ \text{vc}(\text{pre},I_1), \text{vc}(I_1,I_2) \}$
  - Formulae over program variables, e.g. $(i < j \land \text{min} < n)$

- VCs that do not hold:
  - $\text{vc}(I_1,\text{post})$
  - $\text{vc}(I_1,I_2)$
  - $\text{vc}(I_2,I_2)$
  - $\text{vc}(I_2,I_1)$

- Formulae over program variables and symbols $I_1,I_2$, e.g. true $\land i = 0 \Rightarrow I_1$
Iterative Fixed-point: Overview

\[ \text{vc}(\text{pre}, I_1) \quad \text{vc}(I_1, I_2) \quad \text{vc}(I_2, I_2) \]

\[ \text{vc}(I_2, I_1) \]

Pre

Post

\[ \Phi_1, \Phi_2 \rightarrow \{ \text{vc}(\text{pre}, I_1), \text{vc}(I_1, I_2) \} \]

Approx. for invariants

VCs that do not hold

\[ \checkmark \]

\[ \times \]
Iterative Fixed-point: Overview

- Improve candidate
  - Pick unsat VC
  - Use theorem prover to compute optimal change
  - Results in new candidates
Forward Iterative (LFP)

Forward iteration:
• Pick the destination invariant of unsat constraint to improve
• Start with \(<\bot,\ldots,\bot>\)
Forward Iterative (LFP)

Forward iteration:
• Pick the destination invariant of unsat constraint to improve
• Start with $\langle \bot, ..., \bot \rangle$
Forward Iterative (LFP)

Forward iteration:
• Pick the destination invariant of unsat constraint to improve
• Start with $\langle \bot, \ldots, \bot \rangle$

How do we compute these improvements?

`Optimal change` procedure
Positive and Negative unknowns

• Given formula $\Phi$

• Unknowns in have polarities: Positive/Negative
  - Value of positive unknown stronger $\Rightarrow \Phi$ stronger
  - Value of negative unknown stronger $\Rightarrow \Phi$ weaker

  $\Phi = u_1$

• Optimal Soln to $\Phi$
  - Maximally strong positives, maximally weak negatives
Positive and Negative unknowns

• Given formula $\Phi$

• Unknowns in have polarities: Positive/Negative
  – Value of positive unknown stronger $\Rightarrow \Phi$ stronger
  – Value of negative unknown stronger $\Rightarrow \Phi$ weaker

\[ \Phi = \text{negative} - u_1 \]

• Optimal Soln to $\Phi$
  – Maximally strong positives, maximally weak negatives
Positive and Negative unknowns

• Given formula $\Phi$

• Unknowns in have polarities: Positive/Negative
  - Value of positive unknown stronger $\Rightarrow \Phi$ stronger
  - Value of negative unknown stronger $\Rightarrow \Phi$ weaker

• Optimal Soln to $\Phi$
  - Maximally strong positives, maximally weak negatives
Positive and Negative unknowns

- Given formula $\Phi$
- Unknowns in have polarities: Positive/Negative
  - Value of positive unknown stronger $\Rightarrow$ $\Phi$ stronger
  - Value of negative unknown stronger $\Rightarrow$ $\Phi$ weaker

• Optimal Soln to $\Phi$
  - Maximally strong positives, maximally weak negatives
Optimal Change

• One approach (old way):

\[ \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land k_1 \leq k_2 \Rightarrow A[k_1] \leq A[k_2] \]

Domain \( D \): quantifier-free facts

Under-approximate

Lifted domain \( D_\forall \)

Over-approximate

\[ \forall k_1, k_2 : ?? \Rightarrow ?? \]

Very difficult tx, join, widen functions required for the defn of domain \( D_\forall \)
Optimal Change

• Our approach (new way):

\[ \forall k_1, k_2 : 0 \leq k_1, k_2 < n \land k_1 < i \Rightarrow A[k_1] \leq A[k_2] \]

\[ x_1 := e_1 \]

\[ \forall k_1, k_2 : ?? \Rightarrow ?? \]
Optimal Change

- Our approach (new way):

\[ \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land k_1 < i \implies A[k_1] \leq A[k_2] \]

Suppose \( e_i \) does not refer to \( x_i \)

\[ x_1 := e_1 \]

\[ x_2 := e_2 \]

\[ x_3 := e_3 \]

\[ \forall k_1, k_2 : ??_1 \implies ??_2 \]

\[ \forall k_1, k_2 : ??_3 \implies ??_4 \]
Optimal Change

• **Our approach (new way):**

\[ \forall k_1, k_2 : 0 \leq k_1, k_2 < n \land k_1 < i \Rightarrow A[k_1] \leq A[k_2] \]

\[ \land x_1 = e_1 \]

\[ \land x_2 = e_2 \]

\[ \land x_3 = e_3 \Rightarrow \]

Suppose \( e_i \) does not refer to \( x_i \)

\[ \forall k_1, k_2 : ??_1 \Rightarrow ??_2 \]

\[ \forall k_1, k_2 : ??_3 \Rightarrow ??_4 \]
Optimal Change

- Our approach (new way):

\[
\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land k_1 < i \Rightarrow A[k_1] \leq A[k_2] \\
\land x_1 = e_1 \land x_2 = e_2 \land x_3 = e_3
\]

Find optimal solutions to unknowns \( ??_1, ??_2, ??_3, ??_4 \)

\[
\forall k_1, k_2 : ??_1 => ??_2 \\
\forall k_1, k_2 : ??_3 => ??_4
\]

Algorithmic Novelty:
This information can be aggregated efficiently to compute optimal soln

Key idea: Instantiate positive unknowns and query SMT solver polynomial number of times

Unknowns have polarities therefore obey a monotonicity property

Trivial search will involve \( k \times 2^{|\text{Predicates}|} \) queries to the SMT solver

Soln consists of subsets of the set `Predicates`

Two issues: (1) How much time does it take (2) What is the size of the information returned
SMT Solver Query Time

Number of Queries

Time in Milliseconds (log scale)

- 1 query: 5000 milliseconds
- 10 queries: 30000 milliseconds
- 100 queries: 20000 milliseconds
- 1000 queries: 15000 milliseconds
- More than 1000 queries: 30000 milliseconds

413 queries

30 queries
Number of orthogonal solutions generated

Number of orthogonal solutions

Number of calls

0 50 100 150 200 250 300

1 2 3 4 5 6 More

Number of orthogonal solutions
Optimal Change

• Our approach (new way):

\[ \forall k_1, k_2 : 0 \leq k_1, k_2 < n \land k_i \Rightarrow A[k_1] \leq A[k_2] \land x_1 = e_1 \land x_2 = e_2 \land x_3 = e_3 \]

\[ \forall k_1, k_2 : ??_1 \Rightarrow ??_2 \]

\[ \forall k_1, k_2 : ??_3 \Rightarrow ??_4 \]

Find optimal solutions to unknowns ??_1, ??_2, ??_3, ??_4

× Trivial search will involve \( k \times 2^{|\text{Predicates}|} \) queries to the SMT solver

Efficient in practice

Key idea: Instantiate positive unknowns and query SMT solver polynomial number of times

Algorithmic Novelty: This information can be aggregated efficiently to compute optimal soln

Unknows have polarities therefore obey a monotonicity property
Backward Iterative
Greatest Fixed-Point (GFP)

Backward: Same as LFP except

• Pick the source invariant of unsat constraint to improve

• Start with \(< \top, \ldots, \top>\)
Constraint-based over Predicate Abstraction

pre

\[ \text{vc}(\text{pre}, I_1) \]

I_1

\[ \text{vc}(I_1, \text{post}) \]

I_2

\[ \text{vc}(I_1, I_2) \]

\[ \text{vc}(I_2, I_2) \]

\[ \text{vc}(I_2, I_1) \]

post
Constraint-based over Predicate Abstraction

\[ \text{vc}(\text{pre}, I_1) \]

\[ \text{vc}(I_1, \text{post}) \quad \text{vc}(I_1, I_2) \]

\[ \text{vc}(I_2, I_2) \quad \text{vc}(I_2, I_1) \]
Constraint-based over Predicate Abstraction

\[ \text{vc}(\text{pre}, I_1) \]
\[ \text{vc}(I_1, \text{post}) \]
\[ \text{vc}(I_1, I_2) \]
\[ \text{vc}(I_2, I_1) \]
\[ \text{vc}(I_2, I_2) \]
Constraint-based over Predicate Abstraction

\[ \text{vc}(\text{pre}, I_1) \]
\[ \text{vc}(I_1, \text{post}) \]
\[ \text{vc}(I_1, I_2) \]
\[ \text{vc}(I_2, I_1) \]
\[ \text{vc}(I_2, I_2) \]

\[ \forall k : (-) \Rightarrow (-) \]

Remember: VCs are FOL formulae over \( I_1, I_2 \)

If we find assignments \( b^i_j = T/F \) Then we are done!

\( \text{pred}(I_1) \)

\( b_1^1, b_2^1 \ldots b_r^1 \)
\( p_1, p_2 \ldots p_r \)

\( b_1^2, b_2^2 \ldots b_r^2 \)
\( p_1, p_2 \ldots p_r \)

boolean indicators

predicates

template

Remember: VCs are FOL formulae over \( I_1, I_2 \)
Constraint-based over Predicate Abstraction

\[ \text{pred}(A) : A \text{ to unknown predicate} \]

\[ \text{vc}(\text{pre}, I_1) \]
\[ \text{vc}(I_1, \text{post}) \]
\[ \text{vc}(I_1, I_2) \]
\[ \text{vc}(I_2, I_1) \]
\[ \text{vc}(I_2, I_2) \]

Remember:
VCs are FOL formulae over \( I_1, I_2 \)
Constraint-based over Predicate Abstraction

Program constraint to boolean constraint

SAT formulae over predicate indicators

boolc(pred(I))

pred(A) : A to unknown predicate indicator variables

Invariant sol^n

Boolean constraint to satisfying sol^n (SAT Solver)

Local reasoning

Fixed-Point Computation

Remember:
VCs are FOL formulae over I_1, I_2

vc(pre,I_1) \rightarrow boolc(pred(I_1))
vc(I_1,post) \rightarrow boolc(pred(I_1))
vc(I_1,I_2) \rightarrow boolc(pred(I_1), pred(I_2))
vc(I_2,I_1) \rightarrow boolc(pred(I_2), pred(I_1))
vc(I_2,I_2) \rightarrow boolc(pred(I_2))
\[(I_1 \land i \geq n) \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2])\]

Output sorted array

Find min index

if (min \neq i)
swaip A[i], A[min]
VC to Boolean Constraint

\[ (I_1 \land i \geq n) \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]) \]

Template:
\[ \forall k_1, k_2 : (-) \Rightarrow (-) \]

\[ \Phi : (\forall k_1, k_2 : ??_1 \Rightarrow ??_2 \land i \geq n) \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]) \]

Predicates:
- \( p_1 \)
- \( p_2 \)
- ...
VC to Boolean Constraint

\[(I_1 \land i \geq n) \Rightarrow (\forall k_1,k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2])\]

Template:
\[\forall k_1,k_2 : (-) \Rightarrow (-)\]

\[\Phi : (\forall k_1,k_2 : ??_1 \Rightarrow ??_2) \land i \geq n \Rightarrow (\forall k_1,k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2])\]

Positive:
\[\Phi[p_1]\]

Negative:
\[p_2, \ldots\]

\[\Phi[p_1]\] contains only negative unknowns

Query the SMT solver for the maximally weakest value of negative unknown ??_2

There might be multiple maximally weakest

\[N_1, N_1', N_1'' \ldots\]
VC to Boolean Constraint

\[( I_1 \land i \geq n ) \Rightarrow ( \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2] )\]

\[\forall k_1, k_2 : (-) \Rightarrow (-)\]

\[\Phi : (\forall k_1, k_2 : ??_1 \Rightarrow ??_2) \land i \geq n \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2] )\]

positive

\[\Phi[ p_1 ] \rightarrow N_1, N_1', N_1''...\]

\[\Phi[ p_2 ] \rightarrow N_2, N_2', N_2''...\]

\[\vdots\]
VC to Boolean Constraint

\[(I_1 \land i \geq n) \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2])\]

**Template**

\[\forall k_1, k_2 : (-) \Rightarrow (-)\]

**\(\Phi\):**

\[(\forall k_1, k_2 : ???_1 \Rightarrow ???_2 \land i \geq n) \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2])\]

**Predicates**

- \(p_1\):
  - \(N_1, N'_1, N''_1, \ldots\)
  - \(b_{p1} \Rightarrow b_{N1} \lor b_{N'_1} \lor b_{N''_1} \lor \ldots\)

- \(p_2\):
  - \(N_2, N'_2, N''_2, \ldots\)
  - \(\vdots\)

- \(\vdots\)
VC to Boolean Constraint

\[( I_1 \land i \geq n ) \Rightarrow ( \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2] ) \]

**template**

\[\forall k_1, k_2 : (-) \Rightarrow (-)\]

**\( \Phi \):**

\[\forall k_1, k_2 : ??_1 \Rightarrow ??_2 \land i \geq n \Rightarrow ( \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2] )\]

**positive**

\[p_1 \quad N_1, N_1', N_1"...\]
\[b_{p1} \Rightarrow b_{N1} \lor b_{N1'} \lor b_{N1"} \lor ...\]

**negative**

\[p_2 \quad N_2, N_2', N_2"...\]
\[b_{p2} \Rightarrow b_{N2} \lor b_{N2'} \lor b_{N2"} \lor ...\]
VC to Boolean Constraint

\[ (I_1 \land i \geq n) \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]) \]

**template**

\[ \forall k_1, k_2 : (-) \Rightarrow (-) \]

**\( \Phi \) :**

\[ (\forall k_1, k_2 : ?1 \Rightarrow ?2) \land i \geq n \Rightarrow (\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]) \]

**Predicates**

- \( p_1 \)
  - \( N_1, N_1', N_1'' \ldots \)
- \( p_2 \)
  - \( N_2, N_2', N_2'' \ldots \)
- \( \vdots \)

**Boolean SAT formula**

\[ b_{p1} \Rightarrow b_{N1} \lor b_{N1'} \lor b_{N1''} \lor \ldots \]

\[ b_{p2} \Rightarrow b_{N2} \lor b_{N2'} \lor b_{N2''} \lor \ldots \]

\[ \vdots \]
Verification part of the talk

• Three fixed-point inference algorithms
  – Iterative fixed-point
    • Greatest Fixed-Point (GFP)
    • Least Fixed-Point (LFP)
  – Constraint-based (CFP)

• Weakest Precondition Generation

• Experimental evaluation
  – Why experiment?
  – Problem space is undecidable—if not the worst case, is the average, `difficult' case efficiently solvable?
  – We use SAT/SMT solvers—hoping that our instances will not incur their worst case exponential solving time
Implementation

- C Program
- Templates
- Predicate Set
- Microsoft’s Phoenix Compiler
- CFG
- Verification Conditions
  - Iterative Fixed-point GFP/LFP
  - Constraint-based Fixed-Point
  - Z3 SMT Solver
- Candidate Solutions
- Invariants
- Boolean Constraint

Diagram Description:

- The diagram illustrates the process of implementing a C program using Microsoft's Phoenix Compiler.
- The flow starts with the C program and templates, leading to the creation of a Control Flow Graph (CFG).
- The CFG is then used to verify conditions, which can be approached iteratively or through constraint-based fixed-point methods.
- Invariants are used to ensure the correctness of the solutions.
- The Z3 SMT Solver is utilized to handle constraints in the verification process.
Verifying Sorting Algorithms

- **Benchmarks**
  - Considered difficult to verify
  - Require invariants with quantifiers
  - Sorting, ideal benchmark programs

- **5 major sorting algorithms**
  - Insertion Sort, Bubble Sort (\(n^2\) version and termination checking version), Selection Sort, Quick Sort and Merge Sort

- **Properties:**
  - Sort elements
    - \(\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1]\) --- output array is non-decreasing
  - Maintain permutation, when elements distinct
    - \(\forall k \exists j : 0 \leq k < n \Rightarrow A[k] = A_{old}[j] \land 0 \leq j < n\) --- no elements gained
    - \(\forall k \exists j : 0 \leq k < n \Rightarrow A_{old}[k] = A[j] \land 0 \leq j < n\) --- no elements lost
Runtimes: Sortedness

Tool can prove sortedness for all sorting algorithms!
Runtimes: Permutation

... no loss/no gain
part of permutation too!
Inferring program properties

- Full functional correctness
  - Invariant Templates
  - Predicates
  - Template-based invariant inference

No input

Small; intuitive input

User Input

Incrementally increase sophistication

Enumerate all possible predicate sets

Loss of efficiency

More burden on theorem prover
Recap: What VS³ will let you do!

A. Discover invariants with arbitrary quantification and boolean structure
   - ∀: E.g. Sortedness
     \[ \forall k_1, k_2 \quad k_1 \leq k_2 \quad \ldots \]

   - ∀ ∃: E.g. Permutation
     \[ \forall k \exists j \quad k \leq j \quad \ldots \]
Outline

• Three fixed-point inference algorithms
  – Iterative fixed-point
    • Greatest Fixed-Point (GFP)
    • Least Fixed-Point (LFP)
  – Constraint-based (CFP)

• Weakest Precondition Generation

• Program Synthesis
What VS$^3$ will let you do!

B. Discover weakest preconditions

Output array

\[ i := 0 \]

\[ i < n \]

\[ j := i \]

\[ j < n \]

Find min index

if (min != i)

\[ \text{swap } A[i], A[\text{min}] \]

No Swap!

Worst case behavior: swap every time it can

Worse conditions on input?

\[ \leq \]

\[ < \]
Extending to maximally weak preconditions

• **Iterative:**
  – Backwards algorithm:
    • Computes maximally-weak invariants in each step
  – Precondition generation using GFP:
    • Output disjuncts of entry fixed-point in all candidates

• **Constraint-based:**
  – Generate one solution for precondition
  – **Assert constraint** that ensures strictly weaker pre
  – Iterate till unsat—last precondition generated is maximally weakest
Precondition inference experiments

- **Worst case input** (precondition) for sorting:
  - Selection Sort
    - Sorted array except last element is the smallest
  - Insertion, Quick Sort, Bubble Sort (flag)
    - Reverse sorted array

- **Inputs** (precondition) for **functional correctness**:
  - Binary search requires sorted input array
  - Merge in Merge sort requires sorted inputs
  - Missing initializers required in various other programs
Runtimes (GFP): Inferring worst case inputs for sorting

Tool infers worst case inputs for all sorting algorithms!

Nothing to infer as all inputs yield the same behavior
Verification: Summary

Weakest Precondition Inference (GFP)

Strongest Postcondition Inference (LFP)

Input state

Verification

Output state

\( I_{\text{outer}} \)

\( I_{\text{inner}} \)
Synthesis: Problem Statement

Input state

Output state

??
Looping structure; Resources

Resource Limits:
- Stack space
- Computational limits

Input state

Output state

I_{outer}

I_{inner}
Constraint Setup

Verification:
\[ \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land k_1 < i \Rightarrow A[k_1] \leq A[k_2] \]
\[ \land x_1 = e_1 \land x_2 = e_2 \land x_3 = e_3 \]

Synthesis:
\[ \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land k_1 < i \Rightarrow A[k_1] \leq A[k_2] \]
\[ \land x_1 = ??_5 \land x_2 = ??_6 \land x_3 = ??_7 \]

Find optimal solutions to unknowns ??_1, ??_2, ??_3, ??_4

Stack space

Computation limits

Synthesis:
\[ \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \land k_1 < i \Rightarrow A[k_1] \leq A[k_2] \]
\[ \land x_1 = ??_5 \land x_2 = ??_6 \land x_3 = ??_7 \]

and sometimes ??_5, ??_6, ??_7, ??_8

Find optimal solutions to unknowns ??_1, ??_2, ??_3, ??_4

and sometimes ??_9, ??_10

Find optimal solutions to unknowns ??_1, ??_2, ??_3, ??_4

and ??_5, ??_6, ??_7, ??_8
Constraint Solving

Synthesis:

\[ \forall k_1, k_2 : ?? \Rightarrow ?? \]
\[ \land x_1 = ?? \land x_2 = ?? \land x_3 = ?? \]

Find solutions to unknowns

\[ ??, ??, ??, ?? \]
\[ ??, ??, ??, ?? \]
\[ ??, ?? \]

Constraint-based

Precondition

Ideal as it does piecewise reduction

Postcondition

SAT solving fixed-point uses information from both bkwd and fwd
Example: Strassen’s Matrix Mult.

- Recursive Block matrix multiplication
  - Trivial solution is $O(n^3)$
  - Needs 8 multiplication; hence the $O(n^{\ln 8})$
- There exists a way to compute the same eight
  - In 7 multiplications! Will give $O(n^{\ln 7}) = O(n^{2.81})$

Strassen’s sol:

\[
\begin{align*}
    v_1 &= (a_1+a_4)(b_1+b_4) \\
    v_2 &= (a_3+a_4)b_1 \\
    v_3 &= a_1(b_2-b_4) \\
    v_4 &= a_4(b_3-b_1) \\
    v_5 &= (a_1+a_2)b_4 \\
    v_6 &= (a_3-a_1)(b_1+b_2) \\
    v_7 &= (a_2-a_4)(b_3+b_4)
\end{align*}
\]

\[
\begin{align*}
    c_1 &= v_1+v_4-v_5+v_7 \\
    c_2 &= v_3+v_5 \\
    c_3 &= v_2+v_4 \\
    c_4 &= v_1+v_3-v_2+v_6
\end{align*}
\]

Looping structure:
- Acyclic

Resources:
- 7 multiplications
- 7 vars to hold the intermediate results

Our tool synthesizes strassen’s sol along with thousands others:
- 7 multiplications decides the exponent
- Constant factor decided by number of adds/subs
- Add another resource limitations: <19 adds/subs
- We get around 5-10 sol

Our tool can synthesize sorting algorithms too!!!
Synthesis: Result

• If you can give me a verification tool that can handle multiple positive and negative unknowns
• And additionally, generates maximally-weak solutions to negative unknowns

Then

• It can be converted into a synthesis tool!
Conclusion

Weakest Precondition Inference (GFP)

Strongest Postcondition Inference (LFP)

Output state

Input state

Verification

Program Synthesis

I_{outer}

I_{inner}

VS^3: http://www.cs.umd.edu/~saurabhs/pacs

Discuss: saurabhs@cs.umd.edu
Or, catch me after the talk!