A specialization calculus for Program Verification

Wei-Ngan Chin, Cristian Gherghina, Razvan Voicu, Quang Loc Le, Florin Craciun, Shengchao Qin
Setting

• Program verification requires a constant balancing act between
  • Having to reason in rich logics in order to verify richer properties
  • Doing this reasoning in a tractable fashion

• Aim:
  • Diminish the performance penalty incurred by some richer logics
    • logics with inductive predicates
Focus

- Logics with inductive predicates commonly exhibit:
  - **Folding**
    - Tasked with abstracting the current state into a predicate instance
  
  - **Unfolding**
    - Concretizing the predicate by unfolding its definition
      - Is required in order to refine the available information
      - Big source of performance loss
        - Reasoning about concrete states is costlier than reasoning about abstracted states
Proposal

- We introduce a lightweight, sound and complete calculus
  - With the aim of alleviating the unfolding penalty through the application of predicate specialization
The Framework

- A fragment of Separation Logic with inductive predicates

\[
\begin{align*}
\text{pred} &::= p(v^*) \equiv \Phi \\
\Phi &::= \bigvee (\exists w^* \cdot \sigma)^* \\
\sigma &::= \kappa \land \pi \\
\kappa &::= \text{emp} \mid v \mapsto c(v^*) \mid p(v^*) \mid \kappa_1 \ast \kappa_2
\end{align*}
\]
Motivation

```
data node { int val; node prev; node next; }

dll(root, p, n, S) \equiv

root = null \land n = 0 \land S = \{

\lor \exists v, q, S_1 \cdot root \rightarrow node(v, p, q) \land dll(q, root, n-1, S_1)
\land S = S_1 \cup \{v\} \land \forall a \in S_1 \cdot v \leq a
```
Motivation

Given formula: \( \text{dll}(x, p_1, n, S_1) \ast \text{dll}(y, p_2, n, S_2) \land x \neq \text{null} \)

Can I prove: \( S_2 \neq \{\} \) ?

Unfolds to:

\[
\begin{align*}
\text{dll}(\text{root}, p, n, S) & \equiv \text{root} = \text{null} \land n = 0 \land S = \{\} \\
\lor \exists v, q, S_1 \cdot \text{root} \rightarrow \text{node}(v, p, q) \ast \text{dll}(q, \text{root}, n - 1, S_1) \\
& \land S = S_1 \cup \{v\} \land \forall a \in S_1 \cdot v \leq a
\end{align*}
\]
Motivation

- Tweaking the predicate definition:

\[
\text{dll}(\text{root}, p, n, S) \equiv \begin{cases} 
1: (\text{root}=\text{null} \land n=0 \land S=\{\}) \\
\lor 2: (\text{root}=\text{node}(v, p, q) \land \text{dll}(q, \text{root}, n-1, S_1) \land S = S_1 \cup \{v\} \land \forall a \in S_1 \cdot v \leq a)
\end{cases}
\]

- Annotated predicate instances:

\[
\text{dll}(x, p, n, S) \Rightarrow \text{dll}(x, p, n, S)@\{1, 2\}
\]

- Formula with annotated predicates:

\[
\text{dll}(x, p_1, n, S_1)@\{1, 2\} \land \text{dll}(y, p_2, n, S_2)@\{1, 2\} \land x \neq \text{null}
\]

- Specialized formula:

\[
\text{dll}(x, p_1, n, S_1)@\{2\} \land \text{dll}(y, p_2, n, S_2)@\{2\}
\]
Observations

- Predicate annotations simplify considerably the unfolding operation

- Predicate annotations coupled with invariant enrichment can alleviate the need for unfolding
Specification language with annotations

Where:

- \( \mathcal{I} \) denotes a family of predicate invariants
  - One invariant for each set of predicate branches

- \( \mathcal{R} = \{ c | c = \alpha \leftarrow L_0 \} \) denotes a set of pruning conditions

- \( L \subseteq \mathcal{P}(\mathcal{N}) \) denotes a set of viable predicate branches

- \( C \) is a pure formula used in the specialization process, such that \( C \rightarrow \pi \)
**Dll - annotated**

- **dll***(root, p, n, S) ≡
  1: (root = null ∧ n = 0 ∧ S = ∅) ∨
  2: (node(v, p, q) * dll(q, root, n - 1, S₁) ∧
      S = S₁ ∪ \{v\} ∧ ∀a ∈ S₁. v ≤ a
  ); \(\mathcal{I} ; \mathcal{R}\)

- **Where:**

  - **Invariant family** \(\mathcal{I}\) = \(\{(1 \rightarrow S = \emptyset \land n = 0 \land root = \text{null})\),
    \(2 \rightarrow S ≠ \emptyset \land n > 0 \land root ≠ \text{null}\}\)

  - **Pruning conditions**
    \(\mathcal{R} = \{(S = \emptyset \rightarrow 1), (root = \text{null} \rightarrow 1), (n = 0 \rightarrow 1)\),
    \(S ≠ \emptyset \rightarrow 2), (root ≠ \text{null} \rightarrow 2), (n ≠ 0 \rightarrow 2)\),
    (0 ≤ n → 1,2)\)
Specialization transformations

- Predicate specialization

\[ p(v^*)@L_1 \# R_1 \mid C_1 \xrightarrow{sp} p(v^*)@L_2 \# R_2 \mid C_2 \]

- Aims for
  - less viable branches \( \rightarrow L_2 \subseteq L_1 \)
  - less possible pruning conditions \( \rightarrow R_2 \subseteq R_1 \)
  - Strengthening the context : \( C_2 \models C_1 \)
Given \( p(v^*)@L\#R|C \)

1. Pick a pruning condition \((\alpha \rightarrow L_1) \in R\) that contradicts \( C \)
2. Compute the remaining branches \((L_2)\) by dropping the infeasible branches \((L_1)\) from \( L \)
3. Add the stronger invariant, corresponding to \( L_2 \), to \( C \)
   - Where \((L_2 \rightarrow C_1) \in \mathcal{I} \)
4. Drop irrelevant pruning conditions

\[
\begin{align*}
C \land \alpha &\implies false \\
(\alpha \leftarrow L_0) &\in R \\
L \cap L_0 &\neq \emptyset \\
L_2 &= L - L_0 \end{align*}
\]

\[
C_1 = \mathcal{I}nv(p(v^*), L_2) \\
C \land C_1, L_2 &\vdash R \\text{ -- filter } \rightarrow R_1
\]

\[
p(v^*)@L\#R \mid C \xrightarrow{sp} p(v^*)@L_2\#R_1 \mid C \land C_1
\]
dll(x, p, n, S)@\{1,2\}\#R \land n = 0 \mid n = 0

L=\{1,2\} \ ; \ C : n = 0

From \, R \, \text{pick: (} n \neq 0 \to 2\)\

- Contradicts with C : n = 0 \rightarrow \text{such checks can be syntactic}

- \{1,2\} \cap \{2\} = \{2\} \neq \emptyset \Rightarrow L_2 = \{1\}

- C_1 : S = \emptyset \land n = 0 \land x = \text{null}

- S = \emptyset \land n = 0 \land x = \text{null}, \{1\} \vdash R - \text{filter} \rightarrow \emptyset
Predicate specialization

- Given:
  - C : \( n = 0 \land S = \emptyset \land x = \text{null} \)
  - L : \{1\}
- Then:
  - \( R = \{ (x = \text{null} \to 1), (S \neq 0 \to 2), (0 \leq n \to 1, 2) \} \)
  - \( R_f = R \Rightarrow R - R_f = \emptyset \)
- Result:
  - \( \text{dll}(x, p, n, S)@\{1\}\#\emptyset \land n = 0 \mid n = 0 \land S = \emptyset \land x = \text{null} \)
Predicate specialization gains

- Simple implication checks (mostly syntactic)
- Considerable drop in formula size after an unfold
- Increase in formula precision without an unfold
Specialization inference

- We need a mechanism for computing
- Invariant family
  \[ p(v^*) \equiv \bigvee_{i=1}^{n} D_i; \mathcal{I}; \mathcal{R} \]
- Pruning conditions
  \[ p(v^*) \equiv \bigvee_{i=1}^{n} D_i; \mathcal{I}; \mathcal{R} \]
Inferring the invariant family

- Given a predicate definition
  - Compute fixpoint for the predicate definition, the invariant of the entire predicate
  - For each possible set of branches compute a conjunctive invariant
    - Construct the disjunction of all the branches in the current set
    - Approximate the disjunction with one conjunctive invariant

\[
\text{spred}_{old} = (p(v^*) \equiv \bigvee_{i=1}^{n} (\exists u_i^* \cdot \hat{\sigma}_i | C_i)) \quad \rho = [inv_p(v^*) \rightarrow \text{fix}(\bigvee_{i=1}^{n} \exists u_i^* \cdot C_i)]
\]

\[
\mathcal{I} = \{(L \rightarrow \text{hull}(\bigvee_{i \in L} \exists u_i^* \cdot \rho C_i) | \emptyset \subset L \subset \{1..n\}) \cup \{(1..n) \rightarrow \rho(inv_p(v^*))\}
\]

\[
\text{spred}_{new} = (p(v^*) \equiv \bigvee_{i=1}^{n} (\exists u_i^* \cdot \hat{\sigma}_i | \rho C_i); \mathcal{I})
\]
Inferring the pruning conditions

- Given a predicate definition and the invariant families

- Compute an approximation of the transitive closure of each branch invariant

- For each atomic constraint in all closures construct the list of branches in which it appears (by which it is implied)

\[
\text{spred}_{\text{old}} = (p(v^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_i \mid C_i); \ I) \quad G = \bigcup_{i=1}^{n} \text{closure}(I(\{i\}))
\]

\[
\mathcal{R} = \bigcup_{\alpha \in G} \left\{ \alpha \leftarrow \{i \mid 1 \leq i \leq n \land I(\{i\}) \Rightarrow \alpha \} \quad \forall i \in \{1, \ldots, n\} \cdot \hat{\sigma}_i \mid C_i \rightarrow^{*} \hat{\sigma}_{i,2} \mid C_{i,2}
\]

\[
\text{spred}_{\text{new}} = (p(v^*) \equiv \bigvee_{i=1}^{n}(\exists u_i^* \cdot \hat{\sigma}_{i,2} \mid C_{i,2}); \ I; \ \mathcal{R})
\]
Correctness

- We have proven the correctness of both
  - The specialization calculus
  - The invariant and pruning conditions inference algorithm
Experiments

- Verified full functional correctness for varied sized tests

  - A benchmark of 17 small programs (7% faster)
    Singly, doubly, sorted and circular linked lists, selection-sort, insertion-sort, methods for handling heaps an perfect trees

- Complex shapes and invariants (12-90% faster)
  - Red black trees, balanced binary trees, quick sort, merge sort
Experiments

- HIP %
- 17 small progs
- Bubble sort
- Quick sort
- Merge sort
- Complete
- AVL (h, s,b)
- Heap Trees
- AVL (h, s)
- AVL (h, s, s)
- Red Black

HIP + Spec:
- Avg. Disjuncts
- Avg. Size
- Time
## Experiments

<table>
<thead>
<tr>
<th>Progs (specified props)</th>
<th>LOC</th>
<th>HIP</th>
<th>HIP+Spec</th>
<th>HIP</th>
<th>HIP+Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time(s)</td>
<td>Time(s)</td>
<td>Count</td>
<td>Disj</td>
</tr>
<tr>
<td>17 small progs (size)</td>
<td>87</td>
<td>0.86</td>
<td>0.80</td>
<td>229</td>
<td>1.63</td>
</tr>
<tr>
<td>Bubble sort (size, sets)</td>
<td>80</td>
<td>2.20</td>
<td>2.23</td>
<td>296</td>
<td>2.13</td>
</tr>
<tr>
<td>Quick sort (size, sets)</td>
<td>115</td>
<td>2.43</td>
<td>2.13</td>
<td>255</td>
<td>3.29</td>
</tr>
<tr>
<td>Merge sort (size, sets)</td>
<td>128</td>
<td>3.10</td>
<td>2.15</td>
<td>286</td>
<td>2.04</td>
</tr>
<tr>
<td>Complete (size, minh)</td>
<td>137</td>
<td>5.01</td>
<td>2.94</td>
<td>463</td>
<td>3.52</td>
</tr>
<tr>
<td>AVL (height, size, bal)</td>
<td>160</td>
<td>64.1</td>
<td>16.4</td>
<td>764</td>
<td>2.90</td>
</tr>
<tr>
<td>Heap Trees (size, maxel)</td>
<td>208</td>
<td>14.9</td>
<td>4.62</td>
<td>649</td>
<td>2.10</td>
</tr>
<tr>
<td>AVL (height, size)</td>
<td>340</td>
<td>27.5</td>
<td>13.1</td>
<td>704</td>
<td>2.98</td>
</tr>
<tr>
<td>AVL (height, size, sets)</td>
<td>500</td>
<td>657</td>
<td>60.7</td>
<td>1758</td>
<td>8.00</td>
</tr>
<tr>
<td>Red Black (size, height)</td>
<td>630</td>
<td>25.2</td>
<td>15.6</td>
<td>2225</td>
<td>3.84</td>
</tr>
</tbody>
</table>
Conclusions

- We have
  - Presented a simple, flexible calculus for specialization
  - Proven it sound
  - We have exposed considerable performance gains in a general logic framework
Questions?

Thank you!