Mechanized Verification of Graph-manipulating Programs

Shengyi Wang†, Qingxiang Cao‡, Anshuman Mohan†, Aquinas Hobor†

Object-Oriented Programming, Systems, Languages & Applications
October 24, 2019
Our Focus

We would like to verify graph-manipulating programs written in real C with end-to-end machine-checked correctness proofs.

- Hard to reason about
- Occur in critical areas
- C is hard
- Machine-checked proofs are hard
Our Strategy

Use CompCert and Verified Software Toolchain (VST) to certify code against strong specifications expressed with mathematical graphs.

- CompCert + VST = 50+ person-years
- No changes to CompCert
- Add 1% to VST
- Vanilla separation logic (using →* and quantifiers).
- This framework is powerful enough to verify real code.
Our Workflow

Spatial Graph Library

Verification of a Graph-Manipulating Function

{P_0} C_1 \rightarrow \{P_1\} C_2 \rightarrow \{P_2\} C_3 \rightarrow \{P_3\} \ldots

Mathematical Graph Library

Verified Software Toolchain (VST)

C \rightarrow \text{Parser, Simplifier} \rightarrow \text{Coq} \rightarrow \text{Verified Compiler} \rightarrow \text{Asm}

The CompCert Project

Wang, Cao, Mohan, Hobor (NUS)
We have verified half a dozen graph algorithms, including:

- Graph visiting/coloring; ditto for DAG
- Graph reclamation (i.e. spanning tree followed by tree reclamation)
- Graph copy
- Union-find (both for heap- and array-represented nodes)
- Garbage collector for CertiCoq project
  - Generational OCaml-style GC for a purely functional language
  - \( \approx \)400 lines of (rather devilish) C
  - We find two places where C is too weak to define an OCaml-style GC
## Statistics

<table>
<thead>
<tr>
<th>Component</th>
<th>Files</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Utilities</td>
<td>10</td>
<td>2,842</td>
</tr>
<tr>
<td>Math Graph Library</td>
<td>19</td>
<td>12,723</td>
</tr>
<tr>
<td>Memory Model &amp; Logic</td>
<td>13</td>
<td>2,373</td>
</tr>
<tr>
<td>Spatial Graph Library</td>
<td>10</td>
<td>6,458</td>
</tr>
<tr>
<td>Integration into VST</td>
<td>12</td>
<td>1,917</td>
</tr>
<tr>
<td>Examples (excluding GC)</td>
<td>13</td>
<td>3,290</td>
</tr>
<tr>
<td>GC, subdivided into</td>
<td>18</td>
<td>14,170</td>
</tr>
<tr>
<td>• mathematical graph</td>
<td>1</td>
<td>5,764</td>
</tr>
<tr>
<td>• spatial graph</td>
<td>1</td>
<td>1,618</td>
</tr>
<tr>
<td>• function specifications</td>
<td>1</td>
<td>461</td>
</tr>
<tr>
<td>• function Hoare proofs</td>
<td>14</td>
<td>3,062</td>
</tr>
<tr>
<td>• isomorphism proof</td>
<td>1</td>
<td>3,265</td>
</tr>
<tr>
<td><strong>Total Development</strong></td>
<td>95</td>
<td>43,773</td>
</tr>
</tbody>
</table>
Union-Find Algorithm: Problem
Union-Find Algorithm: Find

```c
struct Node {
    unsigned int rank;
    struct Node *parent;
};

struct Node* find(struct Node* x) {
    struct Node *p, *p0;
    p = x -> parent;
    if (p != x) {
        p0 = find(p);
        p = p0;
        x -> parent = p;
    }
    return p;
}
```
Union-Find Algorithm: The Specification of Find

**PRE:** \( \text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \)

**POST:** \( \exists \gamma', ret. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, ret) \)

- How to define \( \gamma \), the mathematical graph?
- How to define \( \text{graph\_rep}(\gamma) \), the spatial representation of the graph in memory?
- How to define other predicates, such as \( \text{uf\_eq}(\gamma, \gamma') \), the graph equivalence and \( \text{root}(\gamma', x, ret) \), the root of \( x \) in \( \gamma' \) is \( ret? \)
Graph Library: Definition of Graph and Path

\[\text{PreGraph} \overset{\text{def}}{=} \{V, E, \text{vvalid}, \text{evalid}, \text{src}, \text{dst}\}\]

\[\text{LabeledGraph} \overset{\text{def}}{=} \{\text{PreGraph}, L_V, L_E, L_G, \text{vlabel}, \text{elabel}, \text{glabel}\}\]

\[\text{GeneralGraph} \overset{\text{def}}{=} \{\text{LabeledGraph}, \text{sound}_{\text{gg}}\}\]

\[\text{Path} \overset{\text{def}}{=} (v_0, [e_0, e_1, \ldots, e_k])\]

\[\gamma \models s \leadsto t \overset{\text{def}}{=} \text{valid\_path}(\gamma, p) \land\]
\[\text{fst}(p) = s \land \text{end}(\gamma, p) = t\]

\[\gamma \models s \leadsto t \overset{\text{def}}{=} \exists p \text{ s.t. } \gamma \models s \leadsto t\]
Spatial Representation of Graphs

```
struct Node {
    unsigned int rank;
    struct Node *parent;
};
```

\[
\text{graph}_\text{rep}(\gamma) \overset{\text{def}}{=} \star \quad \text{v}_\text{rep}(\gamma, v) \\
\quad v_{\text{valid}}(\gamma, v)
\]

\[
\star \quad P \overset{\text{def}}{=} P(v_1) \star P(v_2) \star \cdots \star P(v_n)
\]

\[
\{v_1, v_2, \ldots, v_n\}
\]

\[
\text{v}_\text{rep}(\gamma, v) \overset{\text{def}}{=} v \mapsto v\text{label}(\gamma, v) \star \\
(v + 4) \mapsto \text{prt}(\gamma, v)
\]

\[
\text{prt}(\gamma, v) \overset{\text{def}}{=} \begin{cases} 
     \text{dst}(\gamma, \text{out}(v)) \neq \text{null} \\
     v & \text{otherwise}
\end{cases}
\]
Ramify Rule

\[
\begin{align*}
\{L_1\} C \{L_2\} & \quad G_1 \vdash L_1 \ast (L_2 \ast G_2) \\
\{G_1\} C \{G_2\} & \quad \text{(mod}(C) \cap \text{fv}(L_2 \ast G_2) = \emptyset) \\
\text{Hint: } \forall P, Q. \ P \ast (P \ast Q) \vdash Q \quad \text{(Hobor and Villard)}
\end{align*}
\]
Our Localize Rule

\[
\begin{array}{c}
\{L_1\} C\{\exists x. \; L_2\} \quad G_1 \vdash L_1 \star R \quad R \vdash \forall x. (L_2 \rightarrow \star G_2) \\
\hline
\{G_1\} C\{\exists x. \; G_2\}
\end{array}
\]  

\((\dagger)\) \quad \text{mod}(C) \cap \text{fv}(R) = \emptyset

Comparing to Hobor and Villard’s Ramify rule:

\[
\begin{array}{c}
\{L_1\} C\{L_2\} \quad G_1 \vdash L_1 \star (L_2 \rightarrow \star G_2) \\
\hline
\{G_1\} C\{G_2\}
\end{array}
\]  

\((\ddagger)\) \quad \text{mod}(C) \cap \text{fv}(L_2 \rightarrow \star G_2) = \emptyset
Verification of the Find function

The Specification of Find

**PRE:** \( \text{graph\_rep}(\gamma) \land v\text{valid}(\gamma, x) \)

**POST:** \( \exists \gamma', \text{ret} . \text{graph\_rep}(\gamma') \land u\text{f\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, \text{ret}) \)

\[
\begin{align*}
\text{graph\_rep}(\gamma) & \overset{\text{def}}{=} \quad \star \quad v\_\text{rep}(\gamma, v) \\
v\text{valid}(\gamma, v) & \\
\text{root}(\gamma, x, \text{ret}) & \overset{\text{def}}{=} \gamma \models x \leadsto \text{ret} \land \forall y. \gamma \models \text{ret} \leadsto y \Rightarrow y = \text{ret} \\
u\text{f\_eq}(\gamma_1, \gamma_2) & \overset{\text{def}}{=} (\forall x. v\text{valid}(\gamma_1, x) \iff v\text{valid}(\gamma_2, x)) \land \\
& \forall x, r_1, r_2. \text{root}(\gamma_1, x, r_1) \Rightarrow \\
& \text{root}(\gamma_2, x, r_2) \Rightarrow r_1 = r_2
\end{align*}
\]
Proof Skeleton of Find

\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
  p = x \to \text{parent}; \\
\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
  p0 = \text{find}(p); \\
\{\text{graph}_\text{rep}(\gamma_1) \land \text{uf}\_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x)\} \\
  \{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \\
  x \to \text{parent} = p0 \\
  \{x \mapsto \text{vlabel}(\gamma_1, x), p0\} \\
\{\text{graph}_\text{rep}(\gamma_2) \land \gamma_2 = \text{redirect\_parent}(\gamma_1, x, p0) \land \ldots\} \\
\{\text{graph}_\text{rep}(\gamma_2) \land \text{uf}\_\text{eq}(\gamma, \gamma_2) \land \text{root}(\gamma_2, x, p0)\} \\
\{\exists \gamma'. \text{graph}_\text{rep}(\gamma') \land \text{uf}\_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}
Proof Obligation of Find

\[
\text{graph}\_\text{rep}(\gamma_1) \vdash (x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)) \ast \left( (x \mapsto \text{vlabel}(\gamma_1, x), p0) \Rightarrow \text{graph}\_\text{rep}(\text{redirect}\_\text{parent}(\gamma_1, x, p0)) \right)
\]

\[
\text{uf}\_\text{eq}(\gamma, \gamma_1) \Rightarrow \text{root}(\gamma_1, p, p0) \Rightarrow \text{dst}(\gamma, \text{out}(x)) = p \\
\gamma_2 = \text{redirect}\_\text{parent}(\gamma_1, x, p0) \Rightarrow \text{uf}\_\text{eq}(\gamma, \gamma_2) \land \text{root}(\gamma_2, x, p0)
\]
A Generational Garbage Collector

- 12 generations; mutator allocates only into the first
- Functional mutator, so no backward pointers
- Cheney’s mark-and-copy collects generation to its successor
- Receiving generation may exceed fullness bound, triggering cascade of further pairwise collections
- Most tasks are handled by two key functions: `forward` (to copy individual objects) and `do_scan` (to repair the copied objects)
Separation between pure and spatial reasoning
A Garbage Collector

Undefined behavior in C

- Double-bounded pointer comparisons:

  ```
  int Is_from(value * from_start, 
              value * from_limit, value * v) {
      return (from_start <= v && v < from_limit); }
  ```

  Resolved using CompCert’s “extcall_properties”.

- A classic OCaml trick:

  ```
  int test_int_or_ptr (value x) {
      return (int)(((intnat)x)&1); }
  ```

  Discussing char alignment issues with CompCert.