Mechanized Verification of Graph-manipulating Programs

Shengyi Wang†, Qingxiang Cao‡, Anshuman Mohan†, Aquinas Hobor†
We would like to verify graph-manipulating programs written in real C with end-to-end machine-checked correctness proofs.

- Graph algorithms are hard to reason about but occur in critical areas of real systems
- Real C code has achingly subtle semantics in some places
- Machine-checked proofs are merciless and lengthy: we want to reuse existing codebases
Our Strategy

We will use the **CompCert** certified compiler’s definition of C and the **Verified Software Toolchain**’s (VST) version of **Separation Logic** to certify our code against strong specifications expressed with mathematical graphs.

- Between them, CompCert and VST have 50+ person-years worth of development effort. *It is highly desirous to fit within their frameworks rather than reinventing the wheel.*
- We make no changes to CompCert. We make minimal *(approximately 1% of codebase)* additions to VST (two new tacticals, assorted lemmas).
- Our techniques use vanilla separation logic (albeit with \( \rightarrow \) and quantifiers).
- We have developed an expressive machine-checked framework for mathematical graphs that is **powerful enough to verify real code.**
Our Results

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)
- 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 pinpoint two places where C is too weak to define an OCaml-style GC
- Verify (almost) full graph isomorphism
- $\approx 14,000$ lines of example-specific proof script
Union-Find Algorithm
Motivation

Union-Find Algorithm
Motivation

Union-Find Algorithm
Union-Find Algorithm: Disjoint-Set Data Structure

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

**Union-Find Algorithm: Disjoint-Set Data Structure**

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

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;
}
```
Motivation

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;
};
```
Motivation

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;
};
```
Motivation

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;
};
```
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;
};
Motivation

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;
};
```
Motivation

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;
};
```
Motivation

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;
}
```
Motivation

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;
}
```
Motivation

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;
};
```
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;
}
Motivation

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;
};
```
Motivation

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;
};
```
Motivation

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: 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;
};
```
Verifying Graph-Manipulating Algorithm is Hard
Verifying Graph-Manipulating Algorithm is Hard
Verifying Graph-Manipulating Algorithm is Hard
Verifying Graph-Manipulating Algorithm is Hard
Verifying Graph-Manipulating Algorithm is Hard
Motivation

Verifying Graph-Manipulating Algorithm is Hard
Verifying Graph-Manipulating Algorithm is Hard
• Motivation

• The Mathematical Graph Library
  • Core Definitions
  • Architecture
  • Selection of Properties

• The Spatial Representation of Graphs
  • CompCert and VST
  • Hoare Logic and Separation Logic
  • Spatial Representation of Graphs
  • Localize Rule

• Verification of the Find function
  • Specification
  • Proof Skeleton
  • Modularity

• A Generational Garbage Collector
### 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>
A general definition of graph should have
Graph Library: A General Definition of Graph

A general definition of graph should have

- Vertices
- Pairs of vertices as Edges
A general definition of graph should have

- Vertices
- Pairs of vertices as Edges
A general definition of graph should have

- Vertices
- Edges, sources and destinations
A general definition of graph should have

- Vertices
- Edges, sources and destinations
A general definition of graph should have

- Vertices
- Edges, sources and destinations
- Validity of vertices and edges
Graph Library: A General Definition of Graph

\[
\text{PreGraph} \overset{\text{def}}{=} \{ V, E, \text{vvalid}, \text{evalid}, \text{src}, \text{dst} \}
\]
Graph Library: A General Definition of Graph

\[
\text{PreGraph} \overset{\text{def}}{=} \{ V, E, v\text{valid}, e\text{valid}, \ src, \ dst \}
\]
Graph Library: A General Definition of Graph

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

LabeledGraph \overset{\text{def}}{=} \{ \text{PreGraph}, L_V, L_E, L_G, vlabel, elabel, glabel \}
**Graph Library: A General Definition of Graph**

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

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

\[ \text{GeneralGraph} \overset{\text{def}}{=} \{\text{LabeledGraph}, \text{sound\_gg}\} \]
Graph Library: A General Definition of Graph

PreGraph $\stackrel{\text{def}}{=} \{ V, E, \text{vvalid}, \text{evalid}, \text{src}, \text{dst} \}$

LabeledGraph $\stackrel{\text{def}}{=} \{ \text{PreGraph}, L_V, L_E, L_G, \text{vlabel}, \text{elabel}, \text{glabel} \}$

GeneralGraph $\stackrel{\text{def}}{=} \{ \text{LabeledGraph}, \text{sound}\_\text{gg} \}$

For Example: Acyclic
Graph Library: Definition of Path

- Path is used in defining reachability.
Path is used in defining reachability.

A path is a sequence of edges which connect a sequence of vertices.
Graph Library: Definition of Path

- Path is used in defining reachability.
- A path is a sequence of edges which connect a sequence of vertices.

\[
\text{Path} \overset{\text{def}}{=} \left[ v_0, e_0, v_1, e_1, \ldots, v_{k-1}, e_{k-1}, v_k \right]
\]
Path is used in defining reachability.

A path is a sequence of edges which connect a sequence of vertices.

\[
\text{Path} \overset{\text{def}}{=} [v_0, e_0, v_1, e_1, \ldots, v_{k-1}, e_{k-1}, v_k]
\]

\[
\text{Path} \overset{\text{def}}{=} [e_0, e_1, \ldots, e_k]
\]
Graph Library: Definition of Path

- Path is used in defining reachability.
- A path is a sequence of edges which connect a sequence of vertices.

\[ \text{Path} \overset{\text{def}}{=} [v_0, e_0, v_1, e_1, \ldots, v_{k-1}, e_{k-1}, v_k] \]

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

\[ \text{Path} \overset{\text{def}}{=} (v_0, [e_0, e_1, \ldots, e_k]) \]
Other Derived Definitions: A Peek

\[ s\_valid(\gamma, e) \overset{\text{def}}{=} valid(\gamma, e) \land \]
\[ \quad vvalid(\gamma, src(\gamma, e)) \land vvalid(\gamma, dst(\gamma, e)) \]
**Other Derived Definitions: A Peek**

\[
\text{s\_evalid}(\gamma, e) \overset{\text{def}}{=} \text{evalid}(\gamma, e) \land \\
\quad \text{vvalid}(\gamma, \text{src}(\gamma, e)) \land \text{vvalid}(\gamma, \text{dst}(\gamma, e))
\]

\[
\text{valid\_path}(\gamma, (v, [])) \overset{\text{def}}{=} \text{vvalid}(\gamma, v)
\]

\[
\text{valid\_path}(\gamma, (v, [e_1, e_2, \ldots, e_n])) \overset{\text{def}}{=} v = \text{src}(\gamma, e_1) \land \text{s\_evalid}(\gamma, e_1) \land \\
\quad \text{dst}(\gamma, e_1) = \text{src}(\gamma, e_2) \land \\
\quad \text{s\_evalid}(\gamma, e_2) \land \ldots
\]
Other Derived Definitions: A Peek

\[\text{s\_valid}(\gamma, e) \overset{\text{def}}{=} \text{valid}(\gamma, e) \land \text{vvalid}(\gamma, \text{src}(\gamma, e)) \land \text{vvalid}(\gamma, \text{dst}(\gamma, e))\]

\[\text{valid\_path}(\gamma, (v, [])) \overset{\text{def}}{=} \text{vvalid}(\gamma, v)\]

\[\text{valid\_path}(\gamma, (v, [e_1, e_2, \ldots, e_n])) \overset{\text{def}}{=} v = \text{src}(\gamma, e_1) \land \text{s\_valid}(\gamma, e_1) \land \text{dst}(\gamma, e_1) = \text{src}(\gamma, e_2) \land \text{s\_valid}(\gamma, e_2) \land \ldots\]

\[\text{end}(\gamma, (v, [])) \overset{\text{def}}{=} v\]

\[\text{end}(\gamma, (v, [e_1, e_2, \ldots, e_n])) \overset{\text{def}}{=} \text{dst}(\gamma, e_n)\]

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

\[\gamma \models s \xrightarrow{p} t \overset{\text{def}}{=} \exists p \text{ s.t. } \gamma \models s \xrightarrow{p} t\]
Architecture

The Mathematical Graph Library

Architecture

PreGraph ➔ LabeledGraph ➔ GeneralGraph

Soundness ➔ Condition ➔ Property

Property Lemmas ➔ Dependence ➔ Inheritance ➔ Instantialize

Wang, Cao, Mohan, Hobor (NUS)  Mechanized Verification
INRIA, 29/7/2019  14/38
Various Properties: MathGraph, LstGraph and FiniteGraph
Various Properties: MathGraph, LstGraph and FiniteGraph

MathGraph(\(\gamma\)) \(\text{def} \) \(\{ \)
\[\text{null} : V\]
\[\text{weak}_\text{valid}(v) \text{def} = v = \text{null} \lor \text{vvalid}(\gamma, v)\]
\[\text{valid}_\text{graph} : \forall e. \text{evalid}(\gamma, e) \Rightarrow \text{vvalid}(\gamma, \text{src}(\gamma, e)) \land \text{weak}_\text{valid}(\gamma, \text{dst}(\gamma, e))\]
\[\text{valid}_\text{not}_\text{null} : \forall v. \text{vvalid}(\gamma, v) \Rightarrow v \neq \text{null} \} \)
Various Properties: MathGraph, LstGraph and FiniteGraph

\[
\text{LstGraph}(\gamma) \overset{\text{def}}{=} \begin{cases} \\
\quad \text{out}: V \rightarrow E \\
\quad \text{only\_one\_edge}: \forall v, e. \text{vvalid}(\gamma, v) \Rightarrow \\
\quad \quad \left( \text{src}(\gamma, e) = v \land \\
\quad \quad \quad \text{evalid}(\gamma, e) \right) \iff \\
\quad \quad e = \text{out}(v) \\
\quad \text{acyclic\_path}: \forall v, p. \gamma \models v \xrightarrow{p} v \Rightarrow \\
\quad \quad p = (v, []) \end{cases}
\]
Various Properties: MathGraph, LstGraph and FiniteGraph

FiniteGraph(\(\gamma\)) \text{def} \{\}

- finite\_v: \exists S_v, M_v \text{ s.t. } |S_v| \leq M_v \land \forall v. vvalid(\(\gamma\), v) \Rightarrow v \in S_v
- finite\_e: \exists S_e, M_e \text{ s.t. } |S_e| \leq M_e \land \forall e. evalid(\(\gamma\), e) \Rightarrow e \in S_e\}
Various Properties
Various Properties
Various Properties
Various Properties

FiniteGraph

LstGraph

MathGraph

BiGraph
Various Properties

FiniteGraph  
BiGraph

LstGraph  
MathGraph
• Motivation ✓
• The Mathematical Graph Library ✓
  • Core Definitions ✓
  • Architecture ✓
  • Selection of Properties ✓
• The Spatial Representation of Graphs
  • CompCert and VST
  • Hoare Logic and Separation Logic
  • Spatial Representation of Graphs
  • Localize Rule
• Verification of the Find function
  • Specification
  • Proof Skeleton
  • Modularity
• A Generational Garbage Collector
CompCert and VST

- CompCert

(Leroy et al., Appel et al.)
CompCert and VST

- CompCert
  - C → Coq (Clight) → Machine

(Leroy et al., Appel et al.)
CompCert and VST

- **CompCert**
  - C → Coq (Clight) → Machine
  - Full-Scale C Specification

(Leroy et al., Appel et al.)
CompCert and VST

- CompCert
  - $C \rightarrow \text{Coq (Clight)} \rightarrow$ Machine
  - Full-Scale C Specification
- Verified Software Toolchain

(Leroy et al., Appel et al.)
CompCert and VST

- **CompCert**
  - C → Coq (Clight) → Machine
  - Full-Scale C Specification
- **Verified Software Toolchain**
  - Separation Hoare Logic

(Leroy et al., Appel et al.)
CompCert and VST

- CompCert
  - C → Coq (Clight) → Machine
  - Full-Scale C Specification
- Verified Software Toolchain
  - Separation Hoare Logic
  - Verifiable C

(Leroy et al., Appel et al.)
CompCert and VST

- CompCert
  - C → Coq (Clight) → Machine
  - Full-Scale C Specification
- Verified Software Toolchain
  - Separation Hoare Logic
  - Verifiable C
  - Interactive Symbolic Execution

(Leroy et al., Appel et al.)
Recap: Hoare Logic

\{P\} C \{Q\}
Recap: Separation Logic

\[ P \mathbin{\star} Q \]

(Reynolds et al.)
Recap: Separation Logic

\[ h \models P \star Q \overset{\text{def}}{=} \exists h_1, h_2 \text{s.t. } h_1 \oplus h_2 = h \land h_1 \models P \land h_2 \models Q \]

(Reynolds et al.)
Recap: Separation Logic

\[ P \rightarrow Q \]

(Reynolds et al.)
Recap: Separation Logic

\[ h \models P \star Q \overset{\text{def}}{=} \forall h_1, h_2 . \ h_1 \oplus h = h_2 \Rightarrow h_1 \models P \Rightarrow h_2 \models Q \]

(Reynolds et al.)
Recap: Separation Logic

\[ \forall P, Q. \ P \ast (P \rightarrow Q) \vdash Q \]
Recap: Separation Logic

emp

(Reynolds et al.)
Recap: Separation Logic

\[ a \leftrightarrow v \]
Recap: Separation Logic

\[
\{P\} \; C \{Q\} \frac{\{P \star F\} \; C \{Q \star F\}}{(\text{mod}(C) \cap \text{fv}(F) = \emptyset)}
\]

(Reynolds et al.)
struct Node {
    unsigned int rank;
    struct Node *parent;
};
struct Node {
    unsigned int rank;
    struct Node *parent;
};
Spatial Representation of Graphs

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

$\texttt{graph\_rep}(\gamma) \equiv \bigcirc \vphantom{\null} \begin{array}{c}
\texttt{v\_rep}(\gamma, v) \\
\text{vvalid}(\gamma, v)
\end{array}$
Spatial Representation of Graphs

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

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

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

\[
\{v_1, v_2, \ldots, v_n\}
\]
Spatial Representation of Graphs

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

\[
\text{graph}_\text{rep}(\gamma) \overset{\text{def}}{=} \bigstar \text{v}_\text{rep}(\gamma, v) \quad \text{vvalid}(\gamma,v)
\]

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

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

\[
\text{v}_\text{rep}(\gamma, v) \overset{\text{def}}{=} v \mapsto \text{vlabel}(\gamma, v) \star (v + 4) \mapsto \text{prt}(\gamma, v)
\]
Spatial Representation of Graphs

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

```
graph_rep(γ) \overset{\text{def}}{=} \bigstar_{vvalid(γ,v)} v_rep(γ, v)
```

```
P \overset{\text{def}}{=} P(v_1) \ast P(v_2) \ast \cdots \ast P(v_n)
```

```
v_rep(γ, v) \overset{\text{def}}{=} v \mapsto v\text{label}(γ, v) \ast (v + 4) \mapsto \text{prt}(γ, v)
```

```
\text{prt}(γ, v) \overset{\text{def}}{=} \begin{cases} 
\text{dst}(γ, \text{out}(v)) & \neq \text{null} \\
v & \text{otherwise}
\end{cases}
```

Wang, Cao, Mohan, Hobor (NUS)  Mechanized Verification  INRIA, 29/7/2019  23 / 38
Ramify Rule

\[ \{ G_1 \} \ C \{ G_2 \} \]

(Hobor and Villard)
The Spatial Inference of Graph

Localize Rule

Ramify Rule

\[
\begin{align*}
\{ L_1 \} \ C \{ L_2 \} \\
\{ G_1 \} \ C \{ G_2 \}
\end{align*}
\]

(Hobor and Villard)
Ramify Rule

\[ \{ L_1 \} C \{ L_2 \} \]

\[ \{ G_1 \} C \{ G_2 \} \]

(Hobor and Villard)

Wang, Cao, Mohan, Hobor (NUS)
Ramify Rule

\[
\{ L_1 \} C \{ L_2 \} \\
\{ G_1 \} C \{ G_2 \}
\]

Hint: \( \forall P, Q. P \star (P \rightarrow Q) \vdash Q \)

(Hobor and Villard)
**Ramify Rule**

\[
\begin{align*}
\left\{ L_1 \right\} & \quad C \left\{ L_2 \right\} \\
\left\{ G_1 \right\} & \quad C \left\{ G_2 \right\}
\end{align*}
\]

**Hint:** \( \forall P, Q. \ P \star (P \rightarrow Q) \models Q \)  

(Hobor and Villard)
**Ramify Rule**

\[
\begin{align*}
\{L_1\} & C\{L_2\} & G_1 \vdash L_1 \ast (L_2 \rightarrow G_2) \\
\{G_1\} & C\{G_2\} & (\text{mod}(C) \cap \text{fv}(L_2 \rightarrow G_2) = \emptyset) \\
\end{align*}
\]

(Hobor and Villard)
The Spatial Inference of Graph

Localize Rule

\[
\begin{array}{c}
\{L_1\} \quad C\{ \quad L_2\} \\
G_1 \vdash L_1 \ast R \\
\{G_1\} \quad C\{ \quad G_2\} \\
\hline
R \vdash \\
L_2 \ast G_2
\end{array}
\]
Localize Rule

\[
\begin{array}{c}
\{L_1\} \ C\{\exists x. L_2\} \\
G_1 \vdash L_1 \ast R \\
R \vdash \forall x. (L_2 \ast G_2)
\end{array}
\]

\[
\{G_1\} \ C\{\exists x. G_2\}
\]
Localize Rule

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

\[
\{G_1\} \ C\{\exists x. \ G_2\}
\]

(†) \ mod(C) \cap \text{fv}(R) = \emptyset
The Spatial Inference of Graph

Localize Rule

\[
\begin{align*}
\{L_1\} \quad C\{\exists x. \; L_2\} & \quad G_1 \vdash L_1 \ast R & \quad R \vdash \forall x. (L_2 \ast G_2) \\
\{G_1\} \quad C\{\exists x. \; G_2\} & \hfill \hfill (\dagger)
\end{align*}
\]

\[
(\dagger) \; \text{mod}(C) \cap \text{fv}(R) = \emptyset
\]

Comparing to Ramify rule:

\[
\begin{align*}
\{L_1\} \quad C\{L_2\} & \quad G_1 \vdash L_1 \ast (L_2 \ast G_2) \\
\{G_1\} \quad C\{G_2\} & \hfill \hfill (\ddagger)
\end{align*}
\]

\[
(\ddagger) \; \text{mod}(C) \cap \text{fv}(L_2 \ast G_2) = \emptyset
\]
- Motivation ✓
- The Mathematical Graph Library ✓
  - Core Definitions ✓
  - Architecture ✓
  - Selection of Properties ✓
- The Spatial Representation of Graphs ✓
  - CompCert and VST ✓
  - Hoare Logic and Separation Logic ✓
  - Spatial Representation of Graphs ✓
  - Localize Rule ✓
- Verification of the Find function
  - Specification
  - Proof Skeleton
  - Modularity
- A Generational Garbage Collector
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;
};
```
The Specification of Find

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

**POST:** $\exists \gamma', t \text{s.t. } \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \\
\text{root}(\gamma', x, t)$
The Specification of Find

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

**POST:** \( \exists \gamma', t \text{ s.t. } \text{graph}\_\text{rep}(\gamma') \land \text{uf}\_\text{eq}(\gamma, \gamma') \land \\
\text{root}(\gamma', x, t) \)

\[
\text{graph}\_\text{rep}(\gamma) \overset{\text{def}}{=} \star \text{ v}\_\text{rep}(\gamma, v) \\
\text{vvalid}(\gamma, v) \\
\text{root}(\gamma, x, t) \overset{\text{def}}{=} \gamma \models x \rightsquigarrow t \land \forall y. \gamma \models t \rightsquigarrow y \Rightarrow y = t \\
\text{uf}\_\text{eq}(\gamma_1, \gamma_2) \overset{\text{def}}{=} (\forall x. \text{vvalid}(\gamma_1, x) \leftrightarrow \text{vvalid}(\gamma_2, x)) \land \\
\forall x, r_1, r_2. \text{root}(\gamma_1, x, r_1) \Rightarrow \\
\text{root}(\gamma_2, x, r_2) \Rightarrow r_1 = r_2
\]
Proof Skeleton of Find

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

\quad p = x \rightarrow \text{parent};

\quad p0 = \text{find}(p);

\quad x \rightarrow \text{parent} = p0

\{\exists \gamma'. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}
Proof Skeleton of Find

\[
\{ \text{graph}_\text{rep}(\gamma) \land v\text{valid}(\gamma, x) \} \\
p = x \rightarrow \text{parent}; \\
\{ \text{graph}_\text{rep}(\gamma) \land v\text{valid}(\gamma, x) \land p = \text{prt}(\gamma, x) \} \\
p0 = \text{find}(p);
\]

\[
x \rightarrow \text{parent} = p0
\]

\[
\{ \exists \gamma'. \text{graph}_\text{rep}(\gamma') \land u\text{f}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0) \}
\]
Proof Skeleton of Find

\[
\{ \text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \} \\
\quad \quad \quad \quad p = x \rightarrow \text{parent} ; \\
\{ \text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x) \} \\
\quad \quad \quad \quad p0 = \text{find}(p) ; \\
\quad \quad \quad \quad x \rightarrow \text{parent} = p0
\]

\[
\{ \exists \gamma'. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0) \}
\]
Proof Skeleton of Find

\[\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\}\]

\[p = x \to \text{parent};\]

\[\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\}\]

\[p0 = \text{find}(p);\]

\[\{\text{graph}_\text{rep}(\gamma_1) \land \text{uf}_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x)\}\]

\[x \to \text{parent} = p0\]

\[\{\exists \gamma'. \text{graph}_\text{rep}(\gamma') \land \text{uf}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}\]
Proof Skeleton of Find

\[ \{ \text{graph}_\text{rep}(\gamma) \land vvalid(\gamma, x) \} \]

\[
p = x \to \text{parent};
\]

\[ \{ \text{graph}_\text{rep}(\gamma) \land vvalid(\gamma, x) \land p = \text{prt}(\gamma, x) \} \]

\[
p0 = \text{find}(p);
\]

\[ \{ \text{graph}_\text{rep}(\gamma_1) \land \text{uf}_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x) \} \]

\[
x \to \text{parent} = p0
\]

\[ \{ \exists \gamma'. \text{graph}_\text{rep}(\gamma') \land \text{uf}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0) \} \]
Proof Skeleton of Find

\[
\begin{align*}
\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
p = x \to \text{parent};& \\
\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} & \\
p_0 = \text{find}(p);& \\
\{\text{graph\_rep}(\gamma_1) \land \text{uf\_eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p_0) \land p = \text{prt}(\gamma, x)\} & \\
x \to \text{parent} = p_0 \\
\{\text{graph\_rep}(\gamma_2) \land \gamma_2 = \text{redirect\_parent}(\gamma_1, x, p_0) \land \ldots\}\ \\
\{\exists\gamma'. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p_0)\}
\end{align*}
\]
Proof Skeleton of Find

\[
\begin{align*}
&\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
&\quad p = x \rightarrow \text{parent}; \\
&\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
&\quad p0 = \text{find}(p); \\
&\{\text{graph\_rep}(\gamma_1) \land \text{uf\_eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x)\} \\
&\quad \{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \\
&\quad x \rightarrow \text{parent} = p0 \\
&\quad \{x \mapsto \text{vlabel}(\gamma_1, x), p0\} \\
&\{\text{graph\_rep}(\gamma_2) \land \gamma_2 = \text{redirect\_parent}(\gamma_1, x, p0) \land \ldots\}
\end{align*}
\]

\[
\{\exists \gamma'. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}
\]
Proof Skeleton of Find

\[
\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x)\}
\]

\[
p = x \to \text{parent};
\]

\[
\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\}
\]

\[
p0 = \text{find}(p);
\]

\[
\{\text{graph\_rep}(\gamma_1) \land \text{uf\_eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x)\}
\]

\[
\downarrow \{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\}
\]

\[
x \to \text{parent} = p0
\]

\[
\checkmark \{x \mapsto \text{vlabel}(\gamma_1, x), p0\}
\]

\[
\{\text{graph\_rep}(\gamma_2) \land \gamma_2 = \text{redirect\_parent}(\gamma_1, x, p0) \land \ldots\}
\]

\[
\{\exists \gamma'. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}
\]
Proof Skeleton of Find

\[
\{\text{graph}_\text{rep}(\gamma) \land v\text{valid}(\gamma, x)\} \\
\quad p = x \to \text{parent}; \\
\{\text{graph}_\text{rep}(\gamma) \land v\text{valid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
\quad p0 = \text{find}(p); \\
\{\text{graph}_\text{rep}(\gamma_1) \land \text{uf}_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x)\} \\
\quad \downarrow \{x \mapsto v\text{label}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \\
\quad x \to \text{parent} = p0 \\
\quad \downarrow \{x \mapsto v\text{label}(\gamma_1, x), p0\} \\
\{\text{graph}_\text{rep}(\gamma_2) \land \gamma_2 = \text{redirect}_\text{parent}(\gamma_1, x, p0) \land \ldots\}\]

\[
\exists \gamma'. \text{graph}_\text{rep}(\gamma') \land \text{uf}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\]
Proof Skeleton of Find

\begin{align*}
\{\text{graph}\_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
\text{p} = x \rightarrow \text{parent}; \\
\{\text{graph}\_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
\text{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)\} \\
\downarrow \{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \\
\text{x} \rightarrow \text{parent} = \text{p0} \\
\uparrow \{x \mapsto \text{vlabel}(\gamma_1, x), \text{p0}\} \\
\{\text{graph}\_\text{rep}(\gamma_2) \land \gamma_2 = \text{redirect\_parent}(\gamma_1, x, p0) \land \ldots\}\end{align*}

\begin{align*}
\{\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)\}\end{align*}
Proof Skeleton of Find

\[
\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\}
\]

\[p = x \rightarrow \text{parent};\]

\[
\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\}
\]

\[p0 = \text{find}(p);\]

\[
\{\text{graph}_\text{rep}(\gamma_1) \land \text{uf}_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p0) \land p = \text{prt}(\gamma, x)\}
\]

\[\downarrow \{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\}\]

\[x \rightarrow \text{parent} = p0\]

\[\Rightarrow \{x \mapsto \text{vlabel}(\gamma_1, x), p0\}\]

\[
\{\text{graph}_\text{rep}(\gamma_2) \land \gamma_2 = \text{redirect}_\text{parent}(\gamma_1, x, p0) \land \ldots\}
\]

\[
\{\text{graph}_\text{rep}(\gamma_2) \land \text{uf}_\text{eq}(\gamma, \gamma_2) \land \text{root}(\gamma_2, x, p0)\}
\]

\[
\{\exists \gamma'. \text{graph}_\text{rep}(\gamma') \land \text{uf}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}\]
Proof Obligation of Find

\[
\text{graph}_\text{rep}(\gamma_1) \vdash (x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)) \cdot \\
((x \mapsto \text{vlabel}(\gamma_1, x), p0) \rightarrow \\
\text{graph}_\text{rep(redirect\_parent}(\gamma_1, x, p0)))
\]
Proof Obligation of Find

\[
\text{graph\_rep}(\gamma_1) \vdash (x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)) \star \\
\left( (x \mapsto \text{vlabel}(\gamma_1, x), p0) \rightarrow \\
\text{graph\_rep}(\text{redirect\_parent}(\gamma_1, x, p0)) \right)
\]

\[
\text{uf\_eq}(\gamma, \gamma_1) \Rightarrow \text{root}(\gamma_1, p, p0) \Rightarrow \text{dst}(\gamma, \text{out}(x)) = p \\
\gamma_2 = \text{redirect\_parent}(\gamma_1, x, p0) \Rightarrow \\
\text{uf\_eq}(\gamma, \gamma_2) \land \text{root}(\gamma_2, x, p0)
\]
Modularity: The Array Version of Find

```c
struct subset {
    int parent;
    unsigned int rank;
};

int find(struct subset subs[], int i) {
    int p0 = 0;
    int p = subs[i].parent;
    if (p != i) {
        p0 = find(subs, p);
        p = p0;
        subs[i].parent = p;
    }
    return p;
}
```
The same specification but a different representation

**PRE:** \( \text{graph}_\text{rep}(\gamma, s) \land \text{vvalid}(\gamma, x) \)

**POST:** \( \exists \gamma', \text{ s.t. } \text{graph}_\text{rep}(\gamma', s) \land \text{uf}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, t) \)
The same specification but a different representation

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

**POST:** \( \exists \gamma', t \text{ s.t. } \text{graph\_rep}(\gamma', s) \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, t) \)

\[
\text{graph\_rep}(g, s) \overset{\text{def}}{=} \exists n. \left( \forall v. 0 \leq v < n \iff v\text{valid}(\gamma, v) \land (n \leq \text{MaxInt}/8) \land s \mapsto \text{map}(\lambda v. \text{v\_rep}(\gamma, v)) [0, 1, 2, \ldots, n] \right)
\]
• Motivation ✓
• The Mathematical Graph Library ✓
  • Core Definitions ✓
  • Architecture ✓
  • Selection of Properties ✓
• The Spatial Representation of Graphs ✓
  • CompCert and VST ✓
  • Hoare Logic and Separation Logic ✓
  • Spatial Representation of Graphs ✓
  • Localize Rule ✓
• Verification of the Find function ✓
  • Specification ✓
  • Proof Skeleton ✓
  • Modularity ✓
• A Generational Garbage Collector
A Generational Garbage Collector

- 12 generations; mutator allocates only into the first
- Functional mutator, so no backward pointers
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
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)
Overview of forward and do_scan

roots
(1,1)
(2,2)
(3,1)
Overview of forward and do_scan

roots

(2,3)

(2,2)

(3,1)
Overview of forward and do_scan

roots
(2,3)
(2,2)
(3,1)
Overview of forward and do_scan
Overview of forward and do_scan

<table>
<thead>
<tr>
<th>roots</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3,1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows the garbage collector's state with roots (2,3), (2,2), and (3,1). Nodes a, b, c, d, e, f, and g are connected with arrows indicating the garbage collector's traversal and marking of live objects.
Bugs in the source C code

- Cheney was executed too conservatively, only part of to needs to be scanned.
Bugs in the source C code

- Cheney was executed too conservatively, only part of `to` needs to be scanned.
- Overflow in the following calculation:
  
  ```c
  int space_size =
  h->spaces[i].limit - h->spaces[i].start;
  ```
Undefined behavior in C

- Double-bounded pointer comparisons:

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

Resolved using CompCert’s "extcall_properties".
Unsorted behavior in C

- Double-bounded pointer comparisons:
  ```c
  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:
  ```c
  int test_int_or_ptr (value x) {
    return (int)(((intnat)x)&1); }
  ```
  Discussing char alignment issues with CompCert.
Separation between pure and spatial reasoning