Mechanized Verification of
Graph-manipulating Programs

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

Surrey Concurrency Workshop and S-REPLS 12
July 24, 2019
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 → and quantifiers).
- We have developed an expressive machine-checked framework for mathematical graphs that is powerful enough to verify real code.
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
Motivation

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

```c
struct Node {
    unsigned int rank;
    struct Node *parent;
};
```
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;
};
```
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;
}
```
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;
};
```
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;
};
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;
}
```
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
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
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
Graph Library: A General Definition of Graph

A general definition of graph should have

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

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

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

\[
\text{PreGraph} \overset{\text{def}}{=} \{ V, E, v\text{valid}, e\text{valid}, \text{src}, \text{dst} \}
\]

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

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

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

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

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

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

GeneralGraph $\overset{\text{def}}{=} \{ \text{LabeledGraph}, \text{sound\_gg} \}$

For Example: Acyclic
Graph Library: Definition of Path

- Path is used in defining reachability.
Graph Library: Definition of Path

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

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

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

\[ s_{\text{valid}}(\gamma, e) \overset{\text{def}}{=} \text{valid}(\gamma, e) \land \]
\[ \text{vvalid}(\gamma, \text{src}(\gamma, e)) \land \text{vvalid}(\gamma, \text{dst}(\gamma, e)) \]
The Mathematical Graph Library
Core Definitions

Other Derived Definitions: A Peek

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

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

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

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

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

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

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

The Mathematical Graph Library

Architecture

- **PreGraph**
  - **Label**
  - **Dependence**
  - **PreGraph Lemmas**

- **LabeledGraph**
  - **Soundness**
  - **Condition**
  - **Inheritance**
  - **LabeledGraph Lemmas**

- **GeneralGraph**
  - **Property**
  - **Instantialize**
  - **GeneralGraph Lemmas**
  - **Property Lemmas**

Wang, Cao, Mohan, Hobor (NUS)
Various Properties: MathGraph, LstGraph and FiniteGraph
Various Properties: MathGraph, LstGraph and FiniteGraph

MathGraph(\gamma) \text{ def } \begin{cases} 
\text{null} : V \\
\text{weak\_valid}(v) \text{ def } v = \text{null} \lor \text{vvalid}(\gamma, v) \\
\text{valid\_graph} : \forall e. \text{evalid}(\gamma, e) \Rightarrow \\
\text{vvalid}(\gamma, \text{src}(\gamma, e)) \land \\
\text{weak\_valid}(\text{dst}(\gamma, e)) \\
\text{valid\_not\_null} : \forall v. \text{vvalid}(\gamma, v) \Rightarrow \\
v \neq \text{null}
\end{cases}
Various Properties: MathGraph, LstGraph and FiniteGraph

\[
\text{LstGraph}(\gamma) \overset{\text{def}}{=} \begin{cases} \\
\text{out} : V \rightarrow E \\
\text{only_one_edge} : \forall v, e. \text{vvalid}(\gamma, v) \Rightarrow \\
\left( \text{src}(\gamma, e) = v \wedge \text{evalid}(\gamma, e) \right) \Leftrightarrow \\
\quad e = \text{out}(v) \\
\text{acyclic_path} : \forall v, p. \gamma \models v \xrightarrow{p} v \Rightarrow \\
\quad p = (v, []) \end{cases}
\]
Various Properties: MathGraph, LstGraph and FiniteGraph

FiniteGraph($\gamma$) $\overset{\text{def}}{=} \begin{cases} 
\text{finite}_v : \exists S_v, M_v \text{ s.t. } |S_v| \leq M_v \land \\
\forall v. \text{valid}(\gamma, v) \Rightarrow v \in S_v \\
\text{finite}_e : \exists S_e, M_e \text{ s.t. } |S_e| \leq M_e \land \\
\forall e. \text{valid}(\gamma, e) \Rightarrow e \in S_e 
\end{cases}
Various Properties
Various Properties
Various Properties

FiniteGraph

MathGraph

LstGraph
Various Properties
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
  - $C \rightarrow \text{Coq (Clight)} \rightarrow \text{Machine}$
  - Full-Scale C Specification

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

- CompCert
  - C → Coq (Clight) → Machine
  - Full-Scale C Specification
- Verified Software Toolchain

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

- **CompCert**
  - $C \rightarrow$ Coq (Clight) $\rightarrow$ 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**
  - $C \rightarrow \text{Coq (Clight)} \rightarrow \text{Machine}$
  - 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\}

(C. A. R. Hoare)
Recap: Separation Logic

\[ P \star Q \]

(Reynolds et al.)
Recap: Separation Logic

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

(Reynolds et al.)
Recap: Separation Logic

\[ P \rightarrow^* Q \]

(Reynolds et al.)
Recap: Separation Logic

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

(Reynolds et al.)
Recap: Separation Logic

\[ \forall P, Q . \ P \star (P \Rightarrow Q) \vdash Q \]

(Reynolds et al.)
Recap: Separation Logic

\texttt{emp} (Reynolds et al.)
Recap: Separation Logic

\[ a \leftrightarrow v \]  

(Reynolds et al.)
Recap: Separation Logic

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

(Reynolds et al.)
Spatial Representation of Graphs

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

![Diagram of a graph with nodes and edges]
Spatial Representation of Graphs

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

```latex
\textbf{graph}_{rep}(\gamma) \textbf{def} = \langle v_{valid}(\gamma, v) \rangle
\textbf{v}_{rep}(\gamma, v) \textbf{def} = v \textbf{label}(\gamma, v) \rangle (v + 4) \textbf{prt}(\gamma, v) \textbf{def} = \# \textbf{dst}(\gamma, \textbf{out}(v)) \textbf{null} \\
\text{otherwise}
```

Wang, Cao, Mohan, Hobor (NUS)
The Spatial Inference of Graphs

Spatial Representation of Graphs

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

graph_rep(\gamma) \overset{\text{def}}{=} \bigstar_{v \in V} v_{\text{rep}}(\gamma, v)
\quad \text{valid}(\gamma, v)

\text{prt}(\gamma, v) \overset{\text{def}}{=} \#	ext{dst}(\gamma, \text{out}(v)) = \text{null} \quad \text{otherwise}
struct Node {
    unsigned int rank;
    struct Node *parent;
};

\[
\text{graph\_rep}(\gamma) \overset{\text{def}}{=} \bigstar \text{v\_rep}(\gamma, v) \\
\text{vvalid}(\gamma, v) \\
\bigstar P \overset{\text{def}}{=} P(v_1) \ast P(v_2) \ast \cdots \ast P(v_n) \\
\{v_1,v_2,\ldots,v_n\}
\]
struct Node {
    unsigned int rank;
    struct Node *parent;
};

graph_rep(γ) \overset{\text{def}}{=} \star \quad v_{\text{rep}}(γ, v)_{v_{\text{valid}}(γ, v)}
\begin{align*}
\star & \quad \overset{\text{def}}{=} P(v_1) \star P(v_2) \star \cdots \star P(v_n) \\
\{v_1, v_2, \ldots, v_n\} & \\
\end{align*}

\begin{align*}
v_{\text{rep}}(γ, v) & \overset{\text{def}}{=} v \mapsto v_{\text{label}}(γ, v) \star \\
(v + 4) & \mapsto \text{prt}(γ, v)
\end{align*}
struct Node {
    unsigned int rank;
    struct Node *parent;
};

graph_rep(γ) \equiv \star_{vvalid(γ,v)} v_rep(γ,v)

\begin{align*}
\star \quad P & \equiv P(v_1) \ast P(v_2) \ast \cdots \ast P(v_n) \\
\{v_1,v_2,\ldots,v_n\} & \quad v_{\text{rep}}(γ,v) \equiv v \mapsto v_{\text{label}}(γ,v) \ast (v + 4) \mapsto p_{\text{rt}}(γ,v)
\end{align*}

prt(γ,v) \equiv \begin{cases} 
\text{dst}(γ,\text{out}(v)) & \neq \text{null} \\
\text{null} & \text{otherwise}
\end{cases}
The Spatial Inference of Graph

Localize Rule

Ramify Rule

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

(Hobor and Villard)
Ramify Rule

\[
\{L_1\} C\{L_2\} \quad \underbrace{\quad \left\{ G_1 \right\} C\left\{ G_2 \right\}}_{(Hobor and Villard)}
\]
The Spatial Inference of Graph Localize Rule

**Ramify Rule**

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

(Hobor and Villard)
Ramify Rule

\[
\begin{array}{c}
\{L_1\} C \{L_2\} \\
\{G_1\} C \{G_2\}
\end{array}
\]

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

(Hobor and Villard)
The Spatial Inference of Graph

### Ramify Rule

\[
\{ L_1 \} \ C \{ L_2 \} \\
\{ G_1 \} \ C \{ G_2 \} \\
\text{Hint: } \forall P, Q . \ P \star (P \rightarrow Q) \vdash Q
\]

(Hobor and Villard)
Ramify Rule

\[
\begin{array}{c}
\{L_1\} \quad C \{L_2\} \\
\{G_1\} \quad C \{G_2\}
\end{array}
\quad G_1 \vdash L_1 \star (L_2 \star G_2)
\quad (\text{mod}(C) \cap \text{fv}(L_2 \star G_2) = \emptyset)
\]

(Hobor and Villard)
Localize Rule

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

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

\[ \frac{\{L_1\} \ G\{\exists x. \ L_2\} \quad G_1 \vdash L_1 \ast R \quad R \vdash \forall x. (L_2 \rightarrow G_2)}{\{G_1\} \ G\{\exists x. \ G_2\}} \quad (\dagger) \]

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

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

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

Comparing to Ramify rule:

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

\[(\ddagger) \quad \text{mod}(C) \cap \text{fv}(L_2 \ast G_2) = \emptyset\]
• 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
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:** \( \exists \gamma', t \, s.t. \, \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma, \gamma') \land \text{root}(\gamma', x, t) \)
The Specification of Find

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

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

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

\[
\text{root}(\gamma, x, t) \overset{\text{def}}{=} \gamma \models x \leadsto t \land \forall y. \gamma \models t \leadsto y \Rightarrow y = t
\]

\[
\text{uf\_eq}(\gamma_1, \gamma_2) \overset{\text{def}}{=} (\forall x. \text{vvalid}(\gamma_1, x) \Leftrightarrow \text{vvalid}(\gamma_2, x)) \land \\
\forall x, r_1, r_2. \text{root}(\gamma_1, x, r_1) \Rightarrow \\
\text{root}(\gamma_2, x, r_2) \Rightarrow r_1 = r_2
\]
Proof Skeleton of Find

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

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

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

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

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

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

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

\{\text{graph}\_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
p = x \rightarrow \text{parent}; \\
\{\text{graph}\_\text{rep}(\gamma) \land \text{vvalid}(\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 \text{uf}\_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p0)\}
Proof Skeleton of Find

\[
\{\text{graph}\_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
p = x \rightarrow \text{parent}; \\
\{\text{graph}\_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
p_0 = \text{find}(p); \\
\{\text{graph}\_\text{rep}(\gamma_1) \land \text{uf}\_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p_0) \land p = \text{prt}(\gamma, x)\} \\
x \rightarrow \text{parent} = p_0
\]
Proof Skeleton of Find

```plaintext
{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x)}
\begin{align*}
p &= x \rightarrow \text{parent}; \\
{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p &= \text{prt}(\gamma, x)}
\end{align*}
\begin{align*}
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)}
\end{align*}
\begin{align*}
x \rightarrow \text{parent} &= p0
\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)\}

\begin{align*}
p &= x \rightarrow \text{parent}; \\
\{\text{graph\_rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\}
\end{align*}

\begin{align*}
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)\}
\end{align*}

\begin{align*}
x \rightarrow \text{parent} &= p0
\end{align*}

\{\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\_rep}(\gamma) \land \text{vvalid}(\gamma, x)\}
\begin{align*}
p &= x \rightarrow \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) \\
\{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \\
x \rightarrow \text{parent} &= p0 \\
\{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

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

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

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

\[
p_0 = \text{find}(p);
\]

\[
\{\text{graph\_rep}(\gamma_1) \land \text{uf\_eq}(\gamma,\gamma_1) \land \text{root}(\gamma_1,p,p_0) \land p = \text{prt}(\gamma,x)\}
\]

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

\[
x \rightarrow \text{parent} = p_0
\]

\[
\checkmark \{x \mapsto \text{vlabel}(\gamma_1,x),p_0\}
\]

\[
\{\text{graph\_rep}(\gamma_2) \land \gamma_2 = \text{redirect\_parent}(\gamma_1,x,p_0) \land \ldots\}
\]

\[
\{\exists \gamma'. \text{graph\_rep}(\gamma') \land \text{uf\_eq}(\gamma,\gamma') \land \text{root}(\gamma',x,p_0)\}
\]
Proof Skeleton of Find

\[
\{\text{graph}_\text{rep}(\gamma) \land vvalid(\gamma, x)\} \\
p = x \rightarrow \text{parent}; \\
\{\text{graph}_\text{rep}(\gamma) \land vvalid(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
p_0 = \text{find}(p); \\
\{\text{graph}_\text{rep}(\gamma_1) \land \text{uf}_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p_0) \land p = \text{prt}(\gamma, x)\} \\
\{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \\
x \rightarrow \text{parent} = p_0 \\
\{x \mapsto \text{vlabel}(\gamma_1, x), p_0\} \\
\{\text{graph}_\text{rep}(\gamma_2) \land \gamma_2 = \text{redirect}_\text{parent}(\gamma_1, x, p_0) \land \ldots\} \\
\exists \gamma'. \{\text{graph}_\text{rep}(\gamma') \land \text{uf}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p_0)\}\]
Proof Skeleton of Find

\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x)\} \\
\hspace{1cm} p = x \rightarrow \text{parent}; \hspace{1cm} \\
\{\text{graph}_\text{rep}(\gamma) \land \text{vvalid}(\gamma, x) \land p = \text{prt}(\gamma, x)\} \\
\hspace{1cm} p0 = \text{find}(p); \hspace{1cm} \\
\{\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)\} \\
\hspace{1cm} \downarrow \{x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)\} \hspace{1cm} \\
\hspace{2cm} x \rightarrow \text{parent} = p0 \hspace{1cm} \\
\hspace{1cm} \uparrow \{x \mapsto \text{vlabel}(\gamma_1, x), p0\} \hspace{1cm} \\
\{\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 Skeleton of Find

\[
\begin{align*}
\{ & \text{graph}_\text{rep}(\gamma) \land v\text{valid}(\gamma, x) \} \\
& p = x \to \text{parent}; \\
\{ & \text{graph}_\text{rep}(\gamma) \land v\text{valid}(\gamma, x) \land p = \text{prt}(\gamma, x) \} \\
& p_0 = \text{find}(p); \\
\{ & \text{graph}_\text{rep}(\gamma_1) \land u\text{f}_\text{eq}(\gamma, \gamma_1) \land \text{root}(\gamma_1, p, p_0) \land p = \text{prt}(\gamma, x) \}
\end{align*}
\]

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

\[x \to \text{parent} = p_0\]

\[\checkmark \{ x \mapsto v\text{label}(\gamma_1, x), p_0 \}\]

\[
\begin{align*}
\{ & \text{graph}_\text{rep}(\gamma_2) \land \gamma_2 = \text{redirect}_\text{parent}(\gamma_1, x, p_0) \land \ldots \} \\
\{ & \text{graph}_\text{rep}(\gamma_2) \land u\text{f}_\text{eq}(\gamma, \gamma_2) \land \text{root}(\gamma_2, x, p_0) \} \\
\{ & \exists \gamma'. \text{graph}_\text{rep}(\gamma') \land u\text{f}_\text{eq}(\gamma, \gamma') \land \text{root}(\gamma', x, p_0) \}
\end{align*}
\]
Proof Obligation of Find

\[
\text{graph\_rep}(\gamma_1) \vdash (x \mapsto \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)) \star
\]
\[
\left( (x \mapsto \text{vlabel}(\gamma_1, x), p_0) \rightarrow
\right.
\]
\[
\text{graph\_rep}(\text{redirect\_parent}(\gamma_1, x, p_0))
\]
Proof Obligation of Find

```latex
\text{graph\_rep}(\gamma_1) \vdash (x \leftrightarrow \text{vlabel}(\gamma_1, x), \text{prt}(\gamma_1, x)) * \\
\quad ((x \leftrightarrow \text{vlabel}(\gamma_1, x), p_0) \rightarrow \\
\text{graph\_rep}(\text{redirect\_parent}(\gamma_1, x, p_0)))
```

```latex
\text{uf\_eq}(\gamma, \gamma_1) \Rightarrow \text{root}(\gamma_1, p, p_0) \Rightarrow \text{dst}(\gamma, \text{out}(x)) = p \\
\gamma_2 = \text{redirect\_parent}(\gamma_1, x, p_0) \Rightarrow \\
\text{uf\_eq}(\gamma, \gamma_2) \land \text{root}(\gamma_2, x, p_0)
```
Modularity: The Array Version of Find

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

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

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

\[
\text{graph\_rep}(g, s) \overset{\text{def}}{=} \exists n. \left( \forall v. 0 \leq v < n \Leftrightarrow \text{vvalid}(\gamma, v) \land (n \leq \text{MaxInt}/8) \land s \mapsto \text{map}(\lambda v. \text{v\_rep}(\gamma, v)) [0, 1, 2, \ldots, n] \right)
\]
• 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

<table>
<thead>
<tr>
<th>roots</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1)</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(2,2)</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>(3,1)</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g</td>
<td></td>
</tr>
</tbody>
</table>
Overview of forward and do_scan

- Roots:
  - (2,3)
  - (2,2)
  - (3,1)

Diagram showing the forward and do_scan operations.
Overview of forward and do_scan

- roots
  - (2,3)
  - (2,2)
  - (3,1)

Diagram showing the forward and do_scan operations with nodes a, b, c, d, e, f, and g.
Overview of forward and do_scan

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

Garbage Collector
Overview of forward and do_scan

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

```
roots    (2,3)    (2,2)    (3,1)
c
  |   |
  |   |
  |   |
  |   |
d
  |   |
  |   |
  |   |
  |   |

e
  |   |
  |   |
  |   |
  |   |
f
  |   |
  |   |
  |   |
  |   |
g
  |   |
  |   |
  |   |
  |   |
```
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”.
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”.

- A classic OCaml trick:
  ```c
  int test_int_or_ptr (value x) {
    return (int)(((intnat)x)&1); 
  }
  ```
  Discussing char alignment issues with CompCert.
## 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>
Separation between pure and spatial reasoning