# Mechanized Verification of Graph-manipulating Programs

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

(†) National University of Singapore

(‡) 上海交通大學 Shanghai Jiao Tong University

Object-Oriented Programming, Systems, Languages & Applications
November 8, 2019

## Our Focus

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

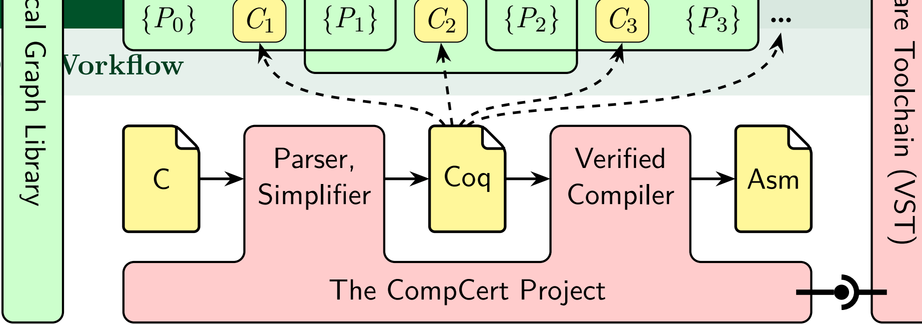
- Hard to reason about
- Occur in critical areas
- C is hard
- Machine-checked proofs are hard

## 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 $-\!\!\ast$ and quantifiers).
- This framework is powerful enough to verify real code.

## O Workflow



cal Graph Library
$\{P_0\}$ $C_1$ $\{P_1\}$ $C_2$ $\{P_2\}$ $C_3$ $\{P_3\}$ ...
C
Parser, Simplifier
Coq
Verified Compiler
Asm
The CompCert Project
are Toolchain (VST)

## 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)
- Graph copy
- Union-find (both for heap- and array-represented nodes)
- Garbage collector for CertiCoq project

## Our Results

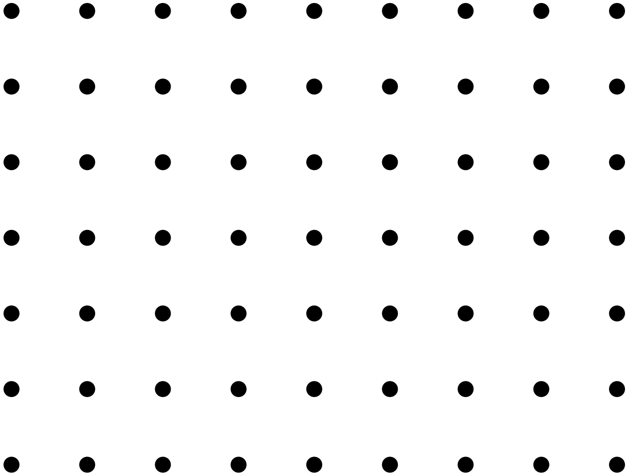
We have verified half a dozen graph algorithms, including:

- Graph visiting/coloring; ditto for DAG
- Graph reclamation (*i.e.* spanning tree followed by tree reclamation)
- 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

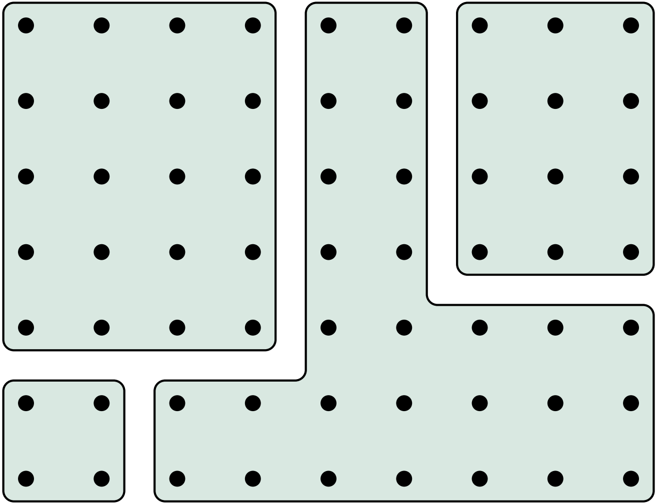
## Statistics

| **Component** | **Files** | **LOC** |
|---|---|---|
| Common Utilities | 10 | 2,842 |
| Math Graph Library | 19 | 12,723 |
| Memory Model & Logic | 13 | 2,373 |
| Spatial Graph Library | 10 | 6,458 |
| Integration into VST | 12 | 1,917 |
| Examples (excluding GC) | 13 | 3,290 |
| GC, subdivided into | 18 | 14,170 |
| • mathematical graph | 1 | 5,764 |
| • spatial graph | 1 | 1,618 |
| • function specifications | 1 | 461 |
| • function Hoare proofs | 14 | 3,062 |
| • isomorphism proof | 1 | 3,265 |
| **Total Development** | 95 | 43,773 |

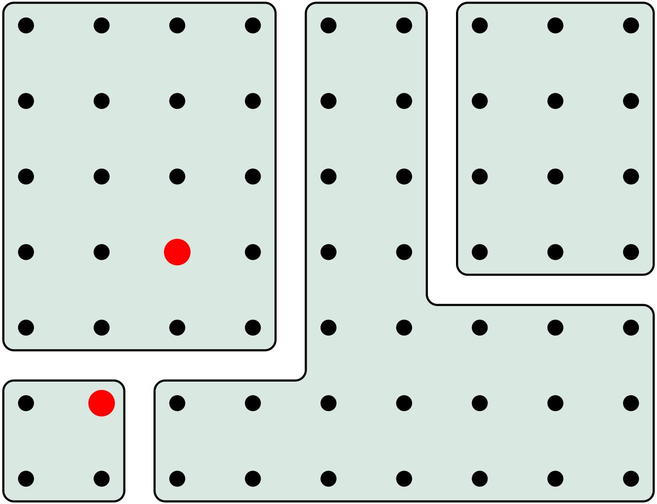
## Union-Find Algorithm: Problem



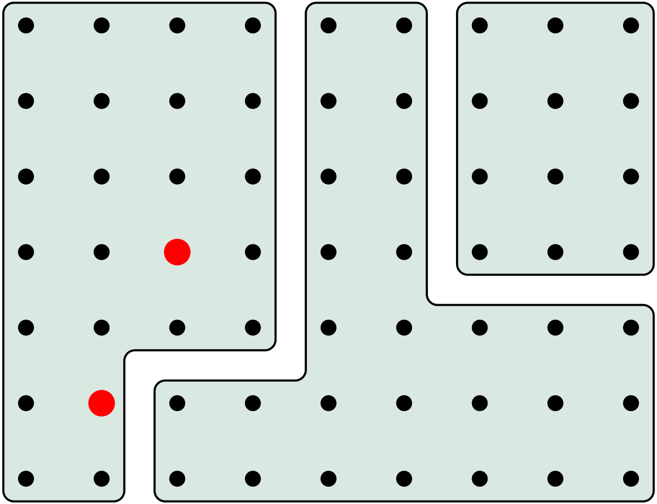
## Union-Find Algorithm: Problem



## Union-Find Algorithm: Problem

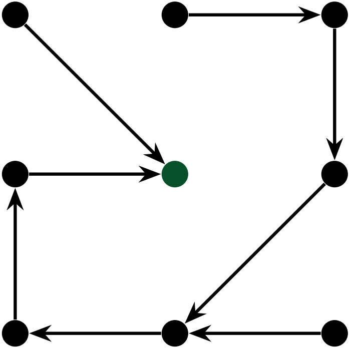


## Union-Find Algorithm: Problem



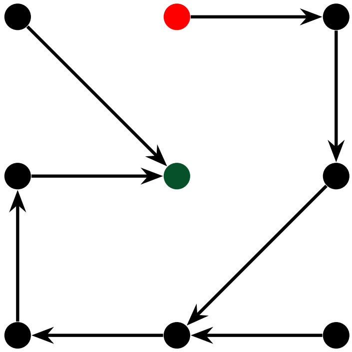
## Union-Find Algorithm: Find

```
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;
};
```



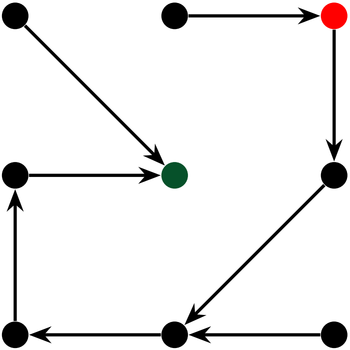
## Union-Find Algorithm: Find

```
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;
};
```



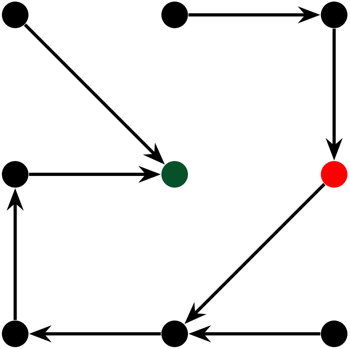
# Union-Find Algorithm: Find

```
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;
};
```



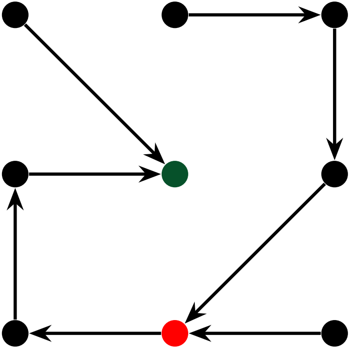
## Union-Find Algorithm: Find

```
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;
};
```



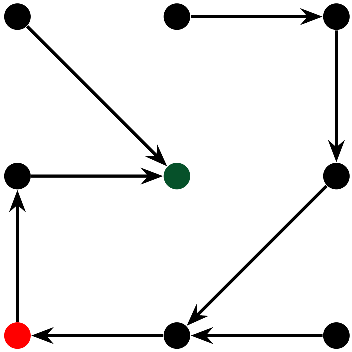
## Union-Find Algorithm: Find

```
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;
};
```



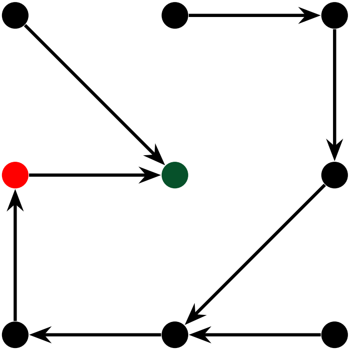
## Union-Find Algorithm: Find

```
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;
};
```



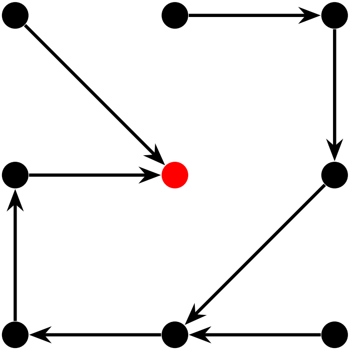
## Union-Find Algorithm: Find

```
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;
};
```



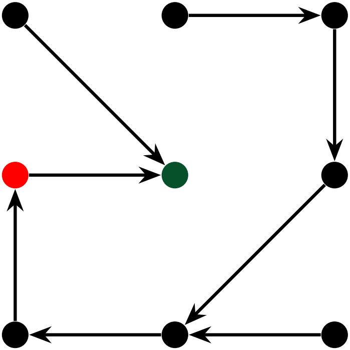
## Union-Find Algorithm: Find

```
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;
};
```



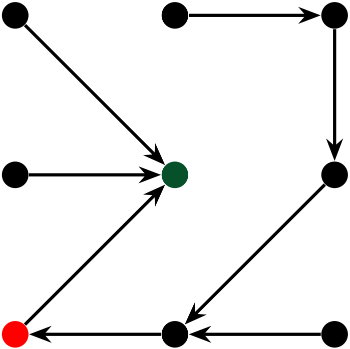
## Union-Find Algorithm: Find

```
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;
};
```



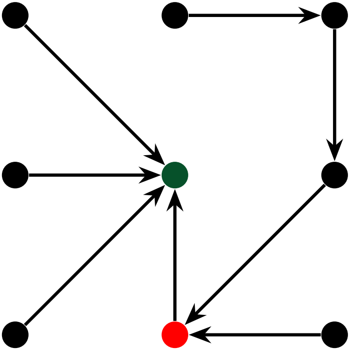
## Union-Find Algorithm: Find

```
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;
};
```



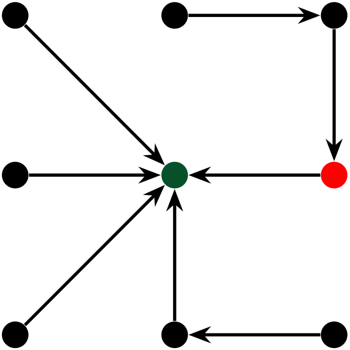
# Union-Find Algorithm: Find

```
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;
};
```



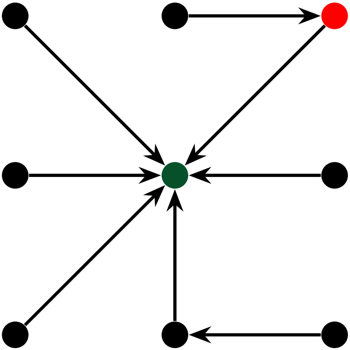
# Union-Find Algorithm: Find

```
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;
};
```



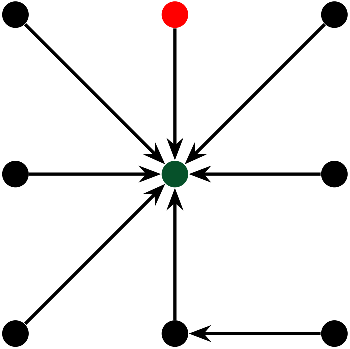
## Union-Find Algorithm: Find

```
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;
};
```



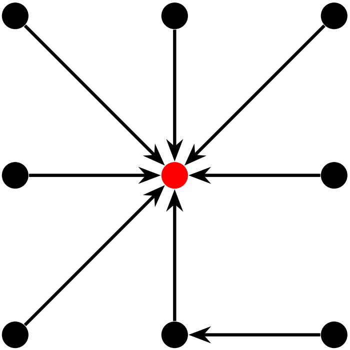
## Union-Find Algorithm: Find

```
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

```
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:** $\texttt{graph\_rep}(\gamma) \land \texttt{vvalid}(\gamma, x)$

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

## Union-Find Algorithm: The Specification of Find

**PRE:** $\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, x)$

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

- How to define $\gamma$, the mathematical graph?

## Union-Find Algorithm: The Specification of Find

**PRE:** $\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, x)$

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

- How to define $\gamma$, the mathematical graph?
- How to define $\texttt{graph\_rep}(\gamma)$, the spatial representation of the graph in memory ?

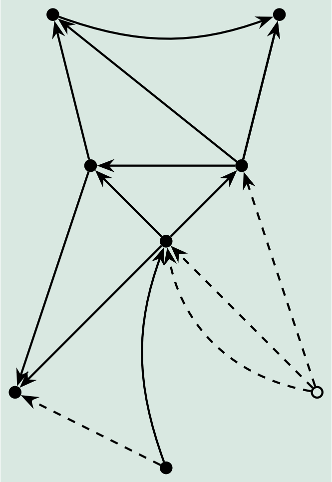
## Union-Find Algorithm: The Specification of Find

**PRE:** $\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, x)$

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

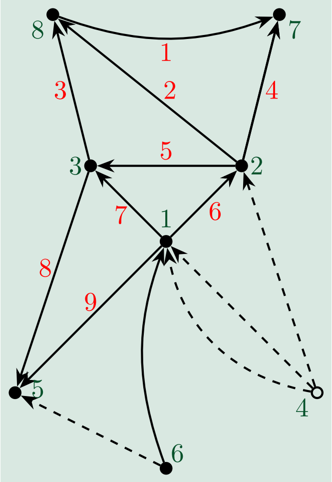
- How to define $\gamma$, the mathematical graph?
- How to define $\texttt{graph\_rep}(\gamma)$, the spatial representation of the graph in memory ?
- How to define other predicates, such as $\texttt{uf\_eq}(\gamma, \gamma')$, the graph equivalence and $\texttt{root}(\gamma', x, ret)$, the root of $x$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

# Graph Library: Definition of Graph and Path



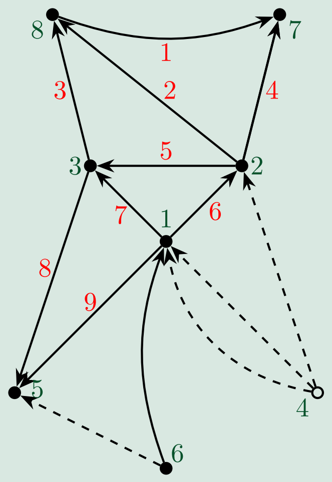
$$\text{PreGraph} \stackrel{\text{def}}{=} \{V, E, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}\}$$

## Graph Library: Definition of Graph and Path




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

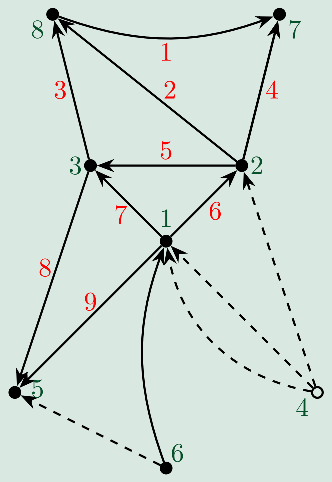
## Graph Library: Definition of Graph and Path




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

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

## Graph Library: Definition of Graph and Path

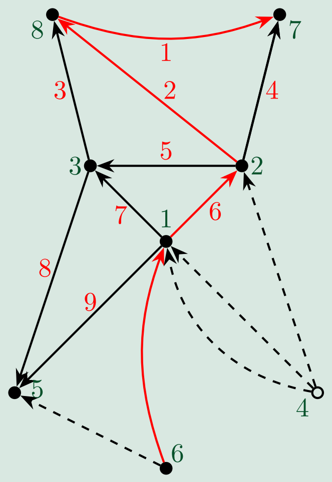



$$\text{PreGraph} \stackrel{\text{def}}{=} \{V, E, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}\}$$

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

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

## Graph Library: Definition of Graph and Path

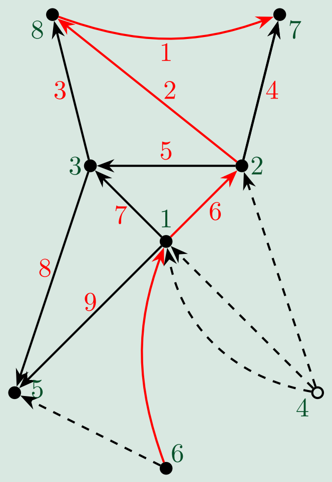



$$\text{PreGraph} \stackrel{\text{def}}{=} \{V, E, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}\}$$

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

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

## Graph Library: Definition of Graph and Path



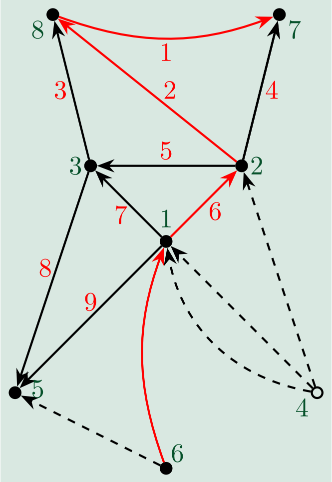

$$\text{PreGraph} \stackrel{\text{def}}{=} \{V, E, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}\}$$

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

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

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

## Graph Library: Definition of Graph and Path




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

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

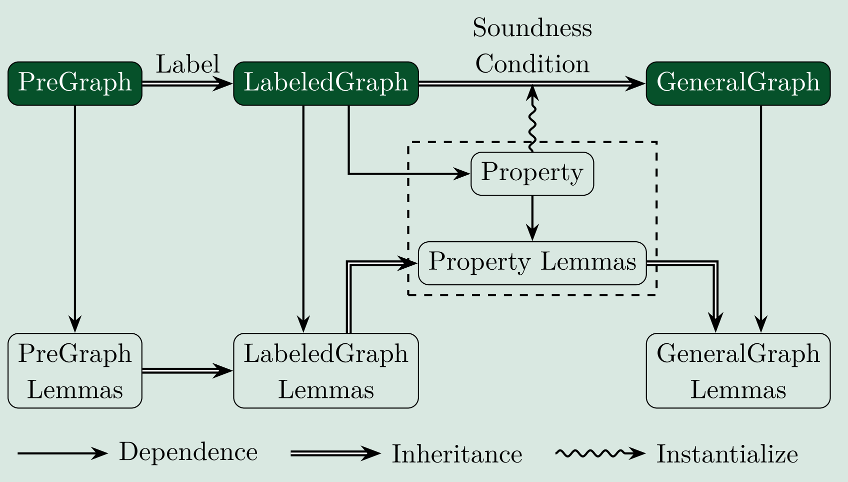
$$\text{GeneralGraph} \stackrel{\text{def}}{=} \{\text{LabeledGraph}, \texttt{sound\_gg}\}$$

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

$$\gamma \models s \stackrel{p}{\rightsquigarrow} t \stackrel{\text{def}}{=} \texttt{valid\_path}(\gamma, p) \wedge \texttt{fst}(p) = s \wedge \texttt{end}(\gamma, p) = t$$

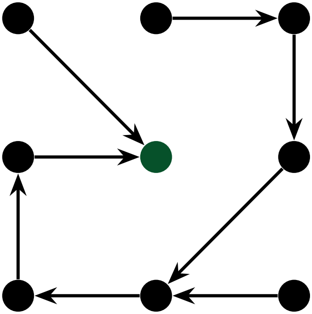
$$\gamma \models s \rightsquigarrow t \stackrel{\text{def}}{=} \exists p \text{ s.t. } \gamma \models s \stackrel{p}{\rightsquigarrow} t$$

## Architecture



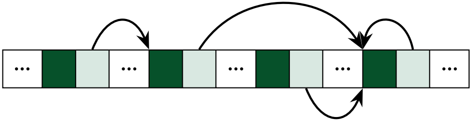
# Spatial Representation of Graphs

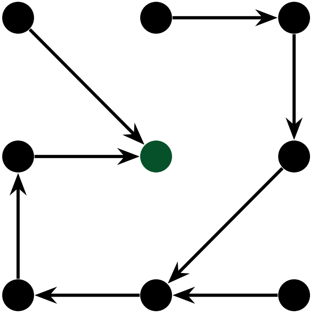
```
struct Node {
    unsigned int rank;
    struct Node *parent;
};
```



## Spatial Representation of Graphs

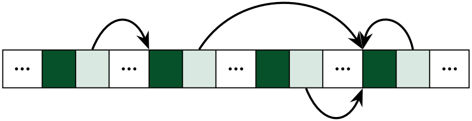
```
struct Node {
    unsigned int rank;
    struct Node *parent;
};
```

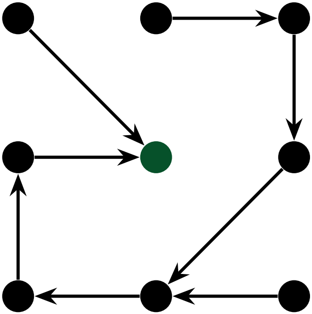


## Spatial Representation of Graphs

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

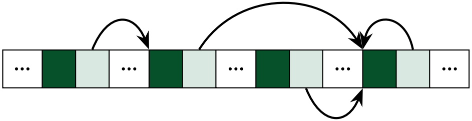


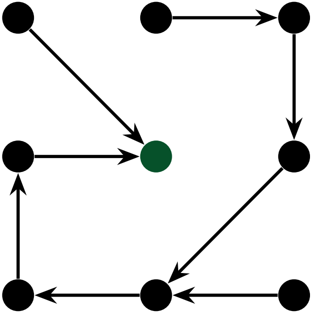



$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma,v)} \texttt{v\_rep}(\gamma, v)$$

## Spatial Representation of Graphs

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



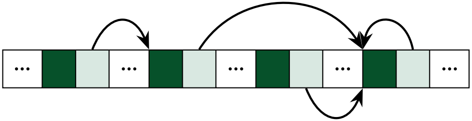



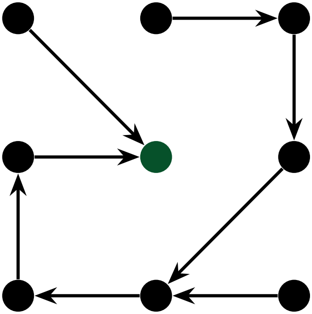
$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma, v)} \texttt{v\_rep}(\gamma, v)$$

$$\mathop{\bigstar}_{\{v_1, v_2, \ldots, v_n\}} P \stackrel{\text{def}}{=} P(v_1) \ast P(v_2) \ast \cdots \ast P(v_n)$$

## Spatial Representation of Graphs

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



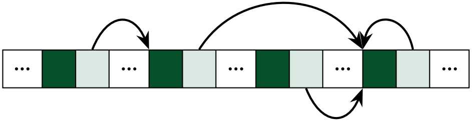
$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma,v)} \texttt{v\_rep}(\gamma, v)$$

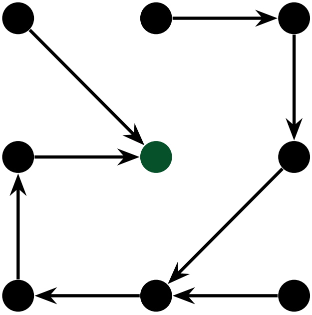
$$\mathop{\bigstar}_{\{v_1,v_2,\ldots,v_n\}} P \stackrel{\text{def}}{=} P(v_1) \star P(v_2) \star \cdots \star P(v_n)$$

$$\texttt{v\_rep}(\gamma, v) \stackrel{\text{def}}{=} v \mapsto \texttt{vlabel}(\gamma, v) \star (v+4) \mapsto \texttt{prt}(\gamma, v)$$

## Spatial Representation of Graphs

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



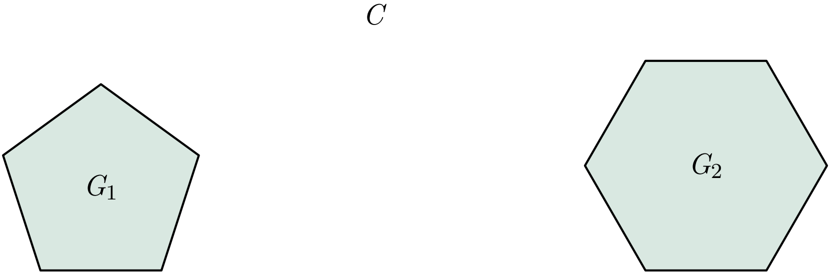
$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma,v)} \texttt{v\_rep}(\gamma, v)$$

$$\mathop{\bigstar}_{\{v_1,v_2,\ldots,v_n\}} P \stackrel{\text{def}}{=} P(v_1) \star P(v_2) \star \cdots \star P(v_n)$$

$$\texttt{v\_rep}(\gamma, v) \stackrel{\text{def}}{=} v \mapsto \texttt{vlabel}(\gamma, v) \star$$
$$(v + 4) \mapsto \texttt{prt}(\gamma, v)$$

$$\texttt{prt}(\gamma, v) \stackrel{\text{def}}{=} \begin{cases} \texttt{dst}(\gamma, \texttt{out}(v)) & \neq \texttt{null} \\ v & \text{otherwise} \end{cases}$$

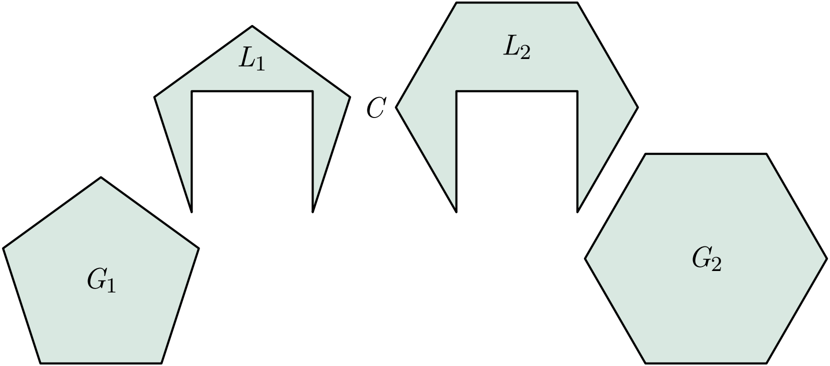
## Ramify Rule




$$\{G_1\}\ C\ \{G_2\}$$

(Hobor and Villard)

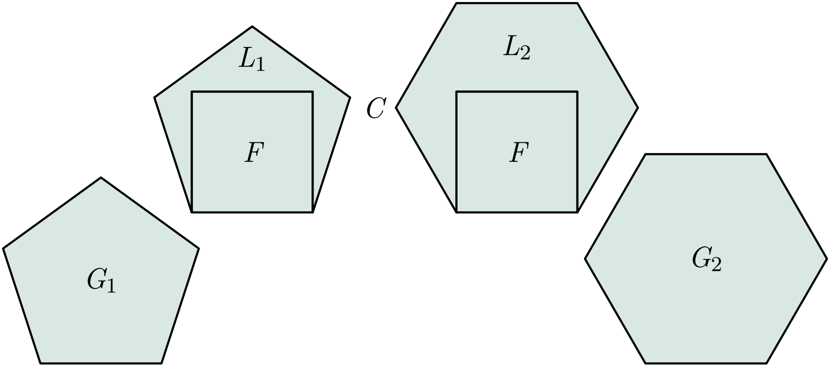
## Ramify Rule




$$\frac{\{L_1\}\ C\ \{L_2\}}{\{G_1\}\ C\ \{G_2\}}$$

(Hobor and Villard)

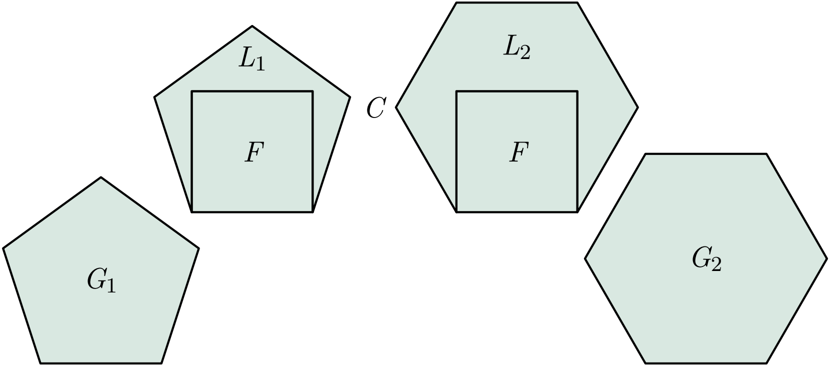
## Ramify Rule




$$\frac{\{L_1\}\ C\ \{L_2\}}{\{G_1\}\ C\ \{G_2\}}$$

(Hobor and Villard)

## Ramify Rule

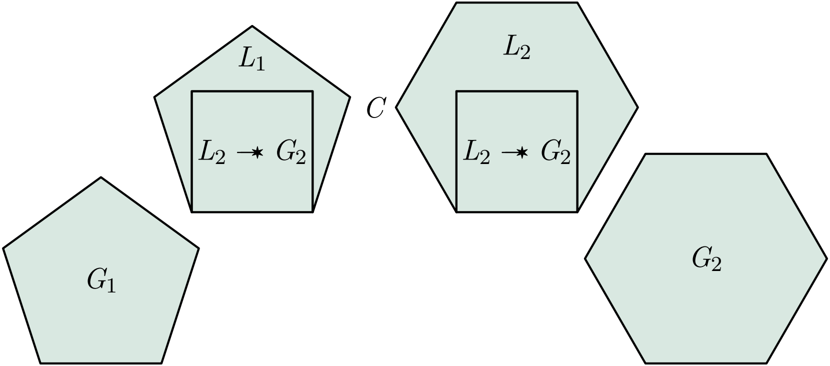



$$\frac{\{L_1\}\ C\ \{L_2\}}{\{G_1\}\ C\ \{G_2\}}$$

Hint: $\forall P, Q.\ P \star (P -\!\!\star\ Q) \vdash Q$

(Hobor and Villard)

## Ramify Rule

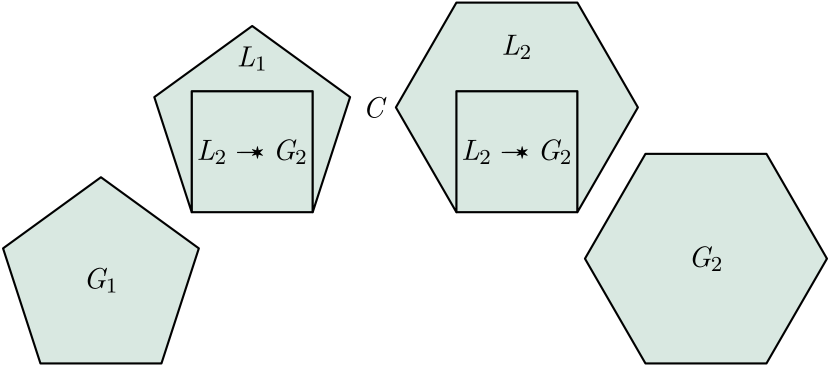



$$\frac{\{L_1\}\ C\ \{L_2\}}{\{G_1\}\ C\ \{G_2\}}$$

Hint: $\forall P, Q \,.\, P \star (P \mathrel{-\!\!\star} Q) \vdash Q$

(Hobor and Villard)

## Ramify Rule




$$\frac{\{L_1\}\ C\ \{L_2\} \qquad G_1 \vdash L_1 \star (L_2 \mathrel{-\!\!\star} G_2)}{\{G_1\}\ C\ \{G_2\}}\ (\mathtt{mod}(C) \cap \mathtt{fv}(L_2 \mathrel{-\!\!\star} G_2) = \emptyset)$$

(Hobor and Villard)

## Our Localize Rule

$$\frac{\{L_1\}\ C\,\{\exists x.\ L_2\} \qquad G_1 \vdash L_1 \star R \qquad R \vdash \forall x.\ (L_2 \mathrel{-\!\!\star} G_2)}{\{G_1\}\ C\,\{\exists x.\ G_2\}}\ (\dagger)$$

$$(\dagger)\ \ \mathtt{mod}(C) \cap \mathtt{fv}(R) = \emptyset$$

Comparing to Hobor and Villard's Ramify rule:

$$\frac{\{L_1\}\ C\,\{L_2\} \qquad G_1 \vdash L_1 \star (L_2 \mathrel{-\!\!\star} G_2)}{\{G_1\}\ C\,\{G_2\}}\ (\ddagger)$$

$$(\ddagger)\ \ \mathtt{mod}(C) \cap \mathtt{fv}(L_2 \mathrel{-\!\!\star} G_2) = \emptyset$$

## The Specification of Find

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

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

## The Specification of Find

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

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

$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma, v)} \texttt{v\_rep}(\gamma, v)$$

$$\texttt{root}(\gamma, x, ret) \stackrel{\text{def}}{=} \gamma \models x \rightsquigarrow ret \land \forall y.\ \gamma \models ret \rightsquigarrow y \Rightarrow y = ret$$

$$\begin{aligned}\texttt{uf\_eq}(\gamma_1, \gamma_2) \stackrel{\text{def}}{=} & \big(\forall x.\ \texttt{vvalid}(\gamma_1, x) \Leftrightarrow \texttt{vvalid}(\gamma_2, x)\big) \ \land \\ & \forall x, r_1, r_2.\ \texttt{root}(\gamma_1, x, r_1) \Rightarrow \\ & \texttt{root}(\gamma_2, x, r_2) \Rightarrow r_1 = r_2\end{aligned}$$

## Proof Skeleton of Find

$\{$`graph_rep`$(\gamma) \wedge$ `vvalid`$(\gamma, $ `x`$)\}$

```
p = x -> parent;
```

```
p0 = find(p);
```

```
x -> parent = p0
```

$\{\exists \gamma'.$ `graph_rep`$(\gamma') \wedge$ `uf_eq`$(\gamma, \gamma') \wedge$ `root`$(\gamma', $ `x`$, $ `p0`$)\}$

## Proof Skeleton of Find

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

```
p = x -> parent;
```

$$\{\texttt{graph\_rep}(\gamma) \land \texttt{vvalid}(\gamma, \texttt{x}) \land \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

```
p0 = find(p);
```

```
x -> parent = p0
```

$$\{\exists \gamma'.\ \texttt{graph\_rep}(\gamma') \land \texttt{uf\_eq}(\gamma, \gamma') \land \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$$

## Proof Skeleton of Find

$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x})\}$

```
p = x -> parent;
```

$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$

```
p0 = find(p);
```

```
x -> parent = p0
```

$\{\exists \gamma'. \texttt{graph\_rep}(\gamma') \wedge \texttt{uf\_eq}(\gamma, \gamma') \wedge \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$

## Proof Skeleton of Find

$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x})\}$

```
p = x -> parent;
```

$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$

```
p0 = find(p);
```

$\{\texttt{graph\_rep}(\gamma_1) \wedge \texttt{uf\_eq}(\gamma, \gamma_1) \wedge \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$

```
x -> parent = p0
```

$\{\exists \gamma'.\ \texttt{graph\_rep}(\gamma') \wedge \texttt{uf\_eq}(\gamma, \gamma') \wedge \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$

## Proof Skeleton of Find

$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x})\}$

```
p = x -> parent;
```

$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$

```
p0 = find(p);
```

$\{\texttt{graph\_rep}(\gamma_1) \wedge \texttt{uf\_eq}(\gamma, \gamma_1) \wedge \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$

```
x -> parent = p0
```

$\{\exists \gamma'. \ \texttt{graph\_rep}(\gamma') \wedge \texttt{uf\_eq}(\gamma, \gamma') \wedge \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$

## Proof Skeleton of Find

$$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x})\}$$

```
p = x -> parent;
```

$$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

```
p0 = find(p);
```

$$\{\texttt{graph\_rep}(\gamma_1) \wedge \texttt{uf\_eq}(\gamma, \gamma_1) \wedge \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

```
x -> parent = p0
```

$$\{\texttt{graph\_rep}(\gamma_2) \wedge \gamma_2 = \texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0}) \wedge \ldots\}$$

$$\{\exists \gamma'.\ \texttt{graph\_rep}(\gamma') \wedge \texttt{uf\_eq}(\gamma, \gamma') \wedge \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$$

# Proof Skeleton of Find

$\{\mathtt{graph\_rep}(\gamma) \wedge \mathtt{vvalid}(\gamma, \mathtt{x})\}$

```
p = x -> parent;
```

$\{\mathtt{graph\_rep}(\gamma) \wedge \mathtt{vvalid}(\gamma, \mathtt{x}) \wedge \mathtt{p} = \mathtt{prt}(\gamma, \mathtt{x})\}$

```
p0 = find(p);
```

$\{\mathtt{graph\_rep}(\gamma_1) \wedge \mathtt{uf\_eq}(\gamma, \gamma_1) \wedge \mathtt{root}(\gamma_1, \mathtt{p}, \mathtt{p0}) \wedge \mathtt{p} = \mathtt{prt}(\gamma, \mathtt{x})\}$

$\{\mathtt{x} \mapsto \mathtt{vlabel}(\gamma_1, \mathtt{x}), \mathtt{prt}(\gamma_1, \mathtt{x})\}$

```
x -> parent = p0
```

$\{\mathtt{x} \mapsto \mathtt{vlabel}(\gamma_1, \mathtt{x}), \mathtt{p0}\}$

$\{\mathtt{graph\_rep}(\gamma_2) \wedge \gamma_2 = \mathtt{redirect\_parent}(\gamma_1, \mathtt{x}, \mathtt{p0}) \wedge \ldots\}$

$\{\exists \gamma'.\ \mathtt{graph\_rep}(\gamma') \wedge \mathtt{uf\_eq}(\gamma, \gamma') \wedge \mathtt{root}(\gamma', \mathtt{x}, \mathtt{p0})\}$

## Proof Skeleton of Find

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

```
p = x -> parent;
```

$$\{\mathtt{graph\_rep}(\gamma) \land \mathtt{vvalid}(\gamma, \mathtt{x}) \land \mathtt{p} = \mathtt{prt}(\gamma, \mathtt{x})\}$$

```
p0 = find(p);
```

$$\{\mathtt{graph\_rep}(\gamma_1) \land \mathtt{uf\_eq}(\gamma, \gamma_1) \land \mathtt{root}(\gamma_1, \mathtt{p}, \mathtt{p0}) \land \mathtt{p} = \mathtt{prt}(\gamma, \mathtt{x})\}$$

$$\searrow \{\mathtt{x} \mapsto \mathtt{vlabel}(\gamma_1, \mathtt{x}), \mathtt{prt}(\gamma_1, \mathtt{x})\}$$

```
x -> parent = p0
```

$$\swarrow \{\mathtt{x} \mapsto \mathtt{vlabel}(\gamma_1, \mathtt{x}), \mathtt{p0}\}$$

$$\{\mathtt{graph\_rep}(\gamma_2) \land \gamma_2 = \mathtt{redirect\_parent}(\gamma_1, \mathtt{x}, \mathtt{p0}) \land \ldots\}$$

$$\{\exists \gamma'.\ \mathtt{graph\_rep}(\gamma') \land \mathtt{uf\_eq}(\gamma, \gamma') \land \mathtt{root}(\gamma', \mathtt{x}, \mathtt{p0})\}$$

## Proof Skeleton of Find

$$\{\mathtt{graph\_rep}(\gamma) \wedge \mathtt{vvalid}(\gamma, \mathtt{x})\}$$

```
p = x -> parent;
```

$$\{\mathtt{graph\_rep}(\gamma) \wedge \mathtt{vvalid}(\gamma, \mathtt{x}) \wedge \mathtt{p} = \mathtt{prt}(\gamma, \mathtt{x})\}$$

```
p0 = find(p);
```

$$\{\mathtt{graph\_rep}(\gamma_1) \wedge \mathtt{uf\_eq}(\gamma, \gamma_1) \wedge \mathtt{root}(\gamma_1, \mathtt{p}, \mathtt{p0}) \wedge \mathtt{p} = \mathtt{prt}(\gamma, \mathtt{x})\}$$

$$\searrow \{\mathtt{x} \mapsto \mathtt{vlabel}(\gamma_1, \mathtt{x}), \mathtt{prt}(\gamma_1, \mathtt{x})\}$$

```
x -> parent = p0
```

$$\swarrow \{\mathtt{x} \mapsto \mathtt{vlabel}(\gamma_1, \mathtt{x}), \mathtt{p0}\}$$

$$\{\mathtt{graph\_rep}(\gamma_2) \wedge \gamma_2 = \mathtt{redirect\_parent}(\gamma_1, \mathtt{x}, \mathtt{p0}) \wedge \ldots\}$$

$$\{\exists \gamma'.\ \mathtt{graph\_rep}(\gamma') \wedge \mathtt{uf\_eq}(\gamma, \gamma') \wedge \mathtt{root}(\gamma', \mathtt{x}, \mathtt{p0})\}$$

## Proof Skeleton of Find

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

```
p = x -> parent;
```

$$\{\texttt{graph\_rep}(\gamma) \land \texttt{vvalid}(\gamma, \texttt{x}) \land \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

```
p0 = find(p);
```

$$\{\texttt{graph\_rep}(\gamma_1) \land \texttt{uf\_eq}(\gamma, \gamma_1) \land \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \land \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

$$\searrow \{\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{prt}(\gamma_1, \texttt{x})\}$$

```
x -> parent = p0
```

$$\swarrow \{\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{p0}\}$$

$$\{\texttt{graph\_rep}(\gamma_2) \land \gamma_2 = \texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0}) \land \ldots\}$$

$$\{\texttt{graph\_rep}(\gamma_2) \land \texttt{uf\_eq}(\gamma, \gamma_2) \land \texttt{root}(\gamma_2, \texttt{x}, \texttt{p0})\}$$

$$\{\exists \gamma'.\ \texttt{graph\_rep}(\gamma') \land \texttt{uf\_eq}(\gamma, \gamma') \land \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$$

## Proof Skeleton of Find

$$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x})\}$$

```
p = x -> parent;
```

$$\{\texttt{graph\_rep}(\gamma) \wedge \texttt{vvalid}(\gamma, \texttt{x}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

```
p0 = find(p);
```

$$\{\texttt{graph\_rep}(\gamma_1) \wedge \texttt{uf\_eq}(\gamma, \gamma_1) \wedge \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \wedge \texttt{p} = \texttt{prt}(\gamma, \texttt{x})\}$$

$$\searrow \{\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{prt}(\gamma_1, \texttt{x})\}$$

```
x -> parent = p0
```

$$\swarrow \{\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{p0}\}$$

$$\{\texttt{graph\_rep}(\gamma_2) \wedge \gamma_2 = \texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0}) \wedge \ldots\}$$

$$\{\texttt{graph\_rep}(\gamma_2) \wedge \texttt{uf\_eq}(\gamma, \gamma_2) \wedge \texttt{root}(\gamma_2, \texttt{x}, \texttt{p0})\}$$

$$\{\exists \gamma'.\ \texttt{graph\_rep}(\gamma') \wedge \texttt{uf\_eq}(\gamma, \gamma') \wedge \texttt{root}(\gamma', \texttt{x}, \texttt{p0})\}$$

## Proof Obligation of Find

$$\texttt{graph\_rep}(\gamma_1) \vdash \big(\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{prt}(\gamma_1, \texttt{x})\big) * \\ \Big(\big(\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{p0}\big) -\!\!* \\ \texttt{graph\_rep}\big(\texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0})\big)\Big)$$

## Proof Obligation of Find

$$\texttt{graph\_rep}(\gamma_1) \vdash \Big(\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{prt}(\gamma_1, \texttt{x})\Big) \ast$$
$$\Bigg(\Big(\texttt{x} \mapsto \texttt{vlabel}(\gamma_1, \texttt{x}), \texttt{p0}\Big) \mathrel{-\!\!\ast}$$
$$\texttt{graph\_rep}\Big(\texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0})\Big)\Bigg)$$

$$\texttt{uf\_eq}(\gamma, \gamma_1) \Rightarrow \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \Rightarrow \texttt{dst}\Big(\gamma, \texttt{out}(\texttt{x})\Big) = \texttt{p}$$
$$\gamma_2 = \texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0}) \Rightarrow$$
$$\texttt{uf\_eq}(\gamma, \gamma_2) \wedge \texttt{root}(\gamma_2, \texttt{x}, \texttt{p0})$$

## A Generational Garbage Collector

- 12 generations; mutator allocates only into the first
- Functional mutator, so no backward pointers

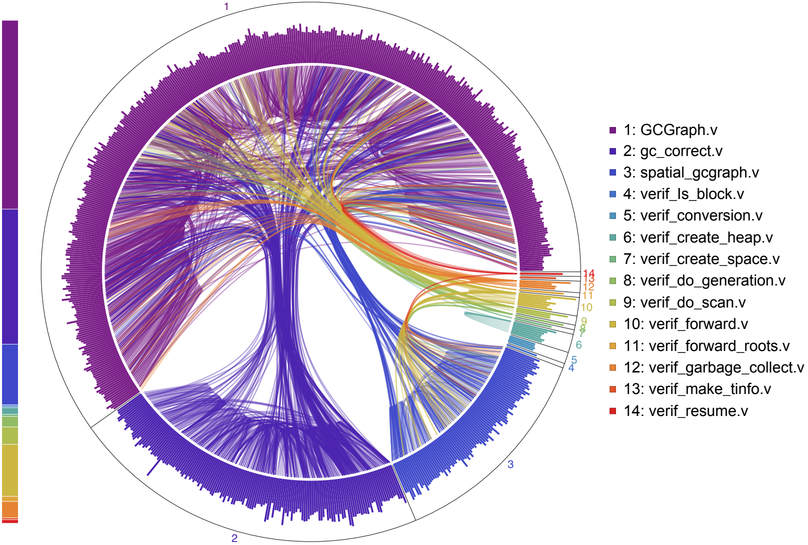
## A Generational Garbage Collector

- 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)

# Separation between pure and spatial reasoning



## 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".

## 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.