Mechanized Verification of Graph-manipulating Programs

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

(†)

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY (‡)

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

Our Focus

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

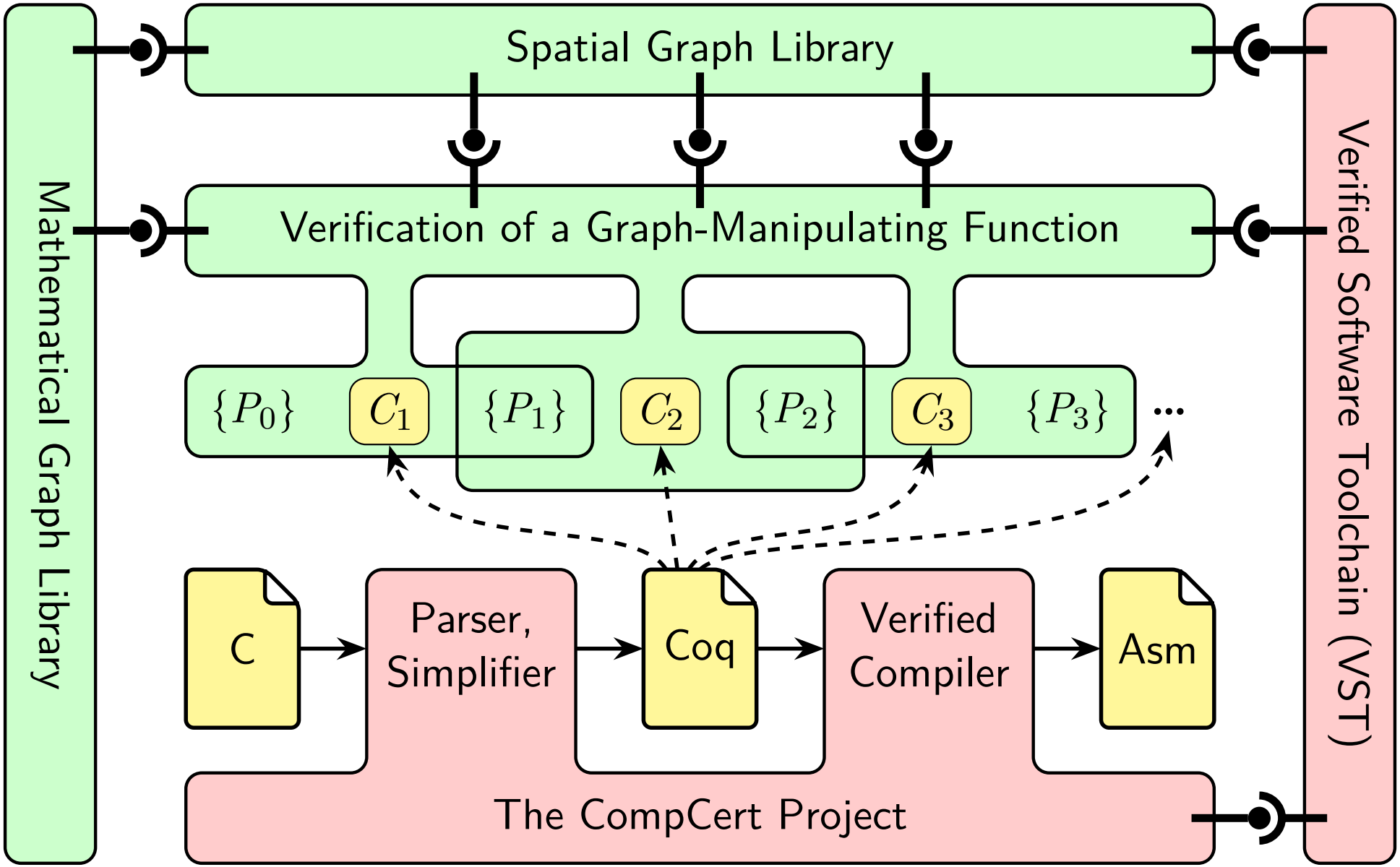
- Hard to reason about
- Occur in critical areas
- C is hard
- Machine-checked proofs are hard

Our Strategy

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

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

Our Workflow



Our Results

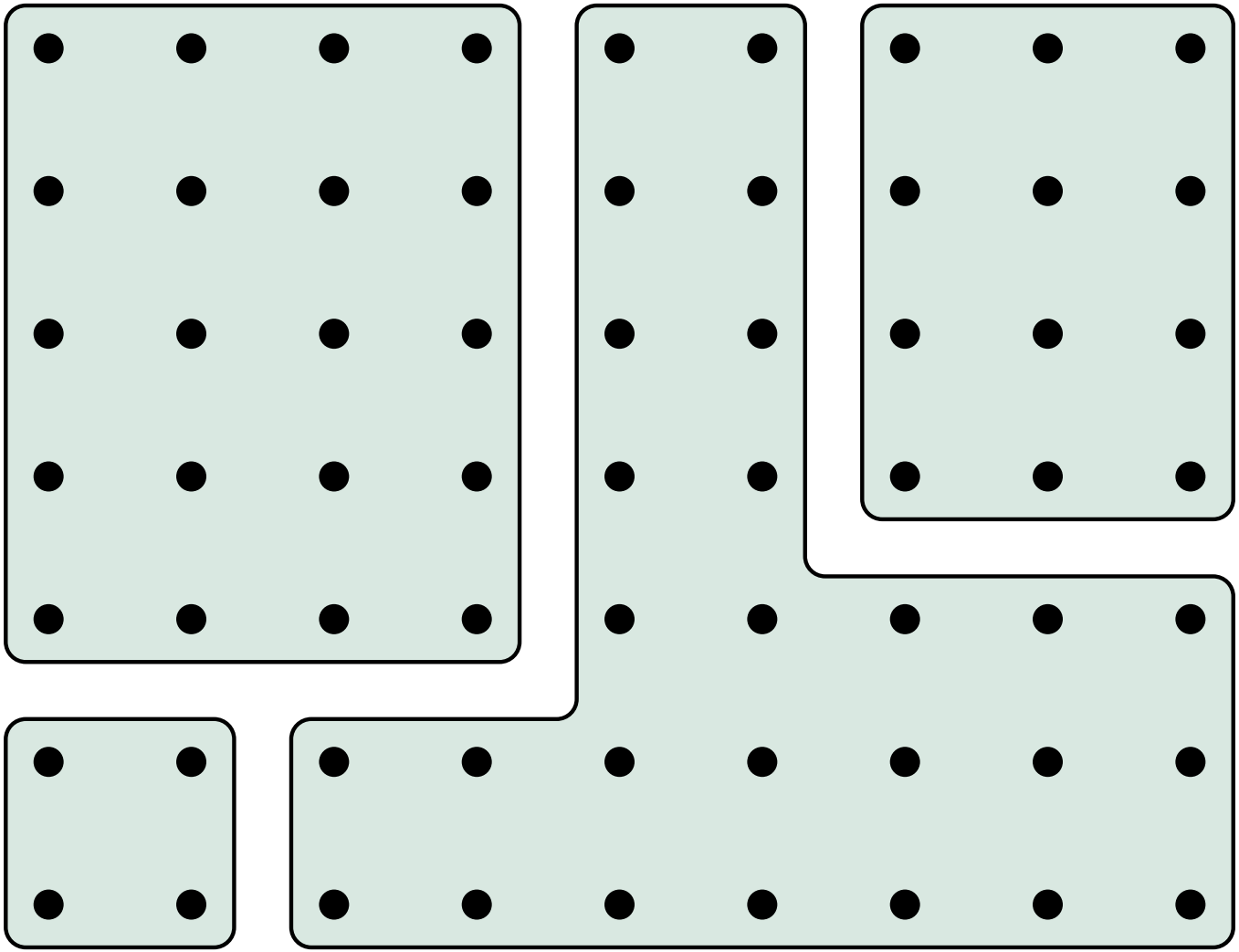
We have verified half a dozen graph algorithms, including:

- 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
  - ≈400 lines of (rather devilish) C
  - We find two places where C is too weak to define an OCaml-style GC

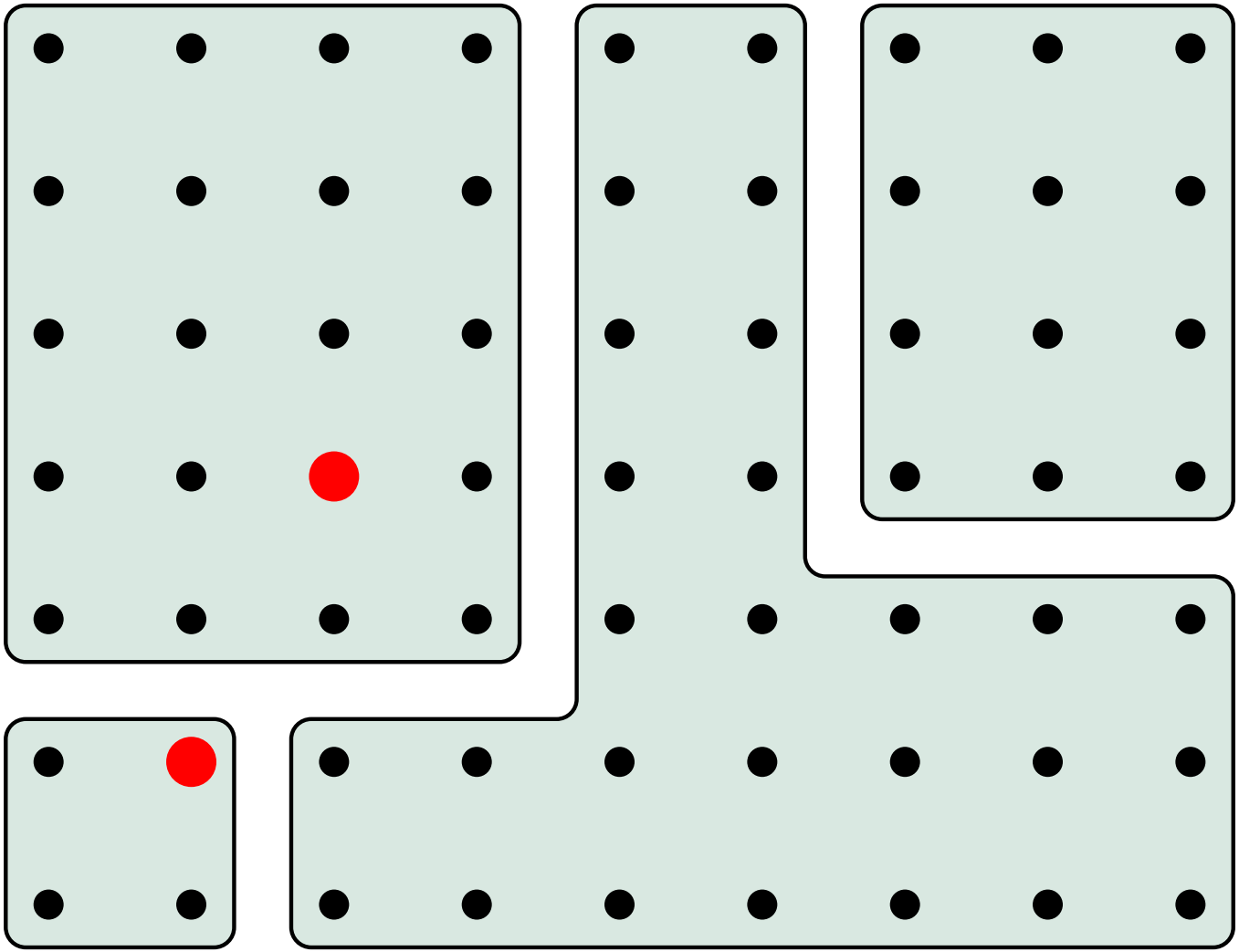
Statistics

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

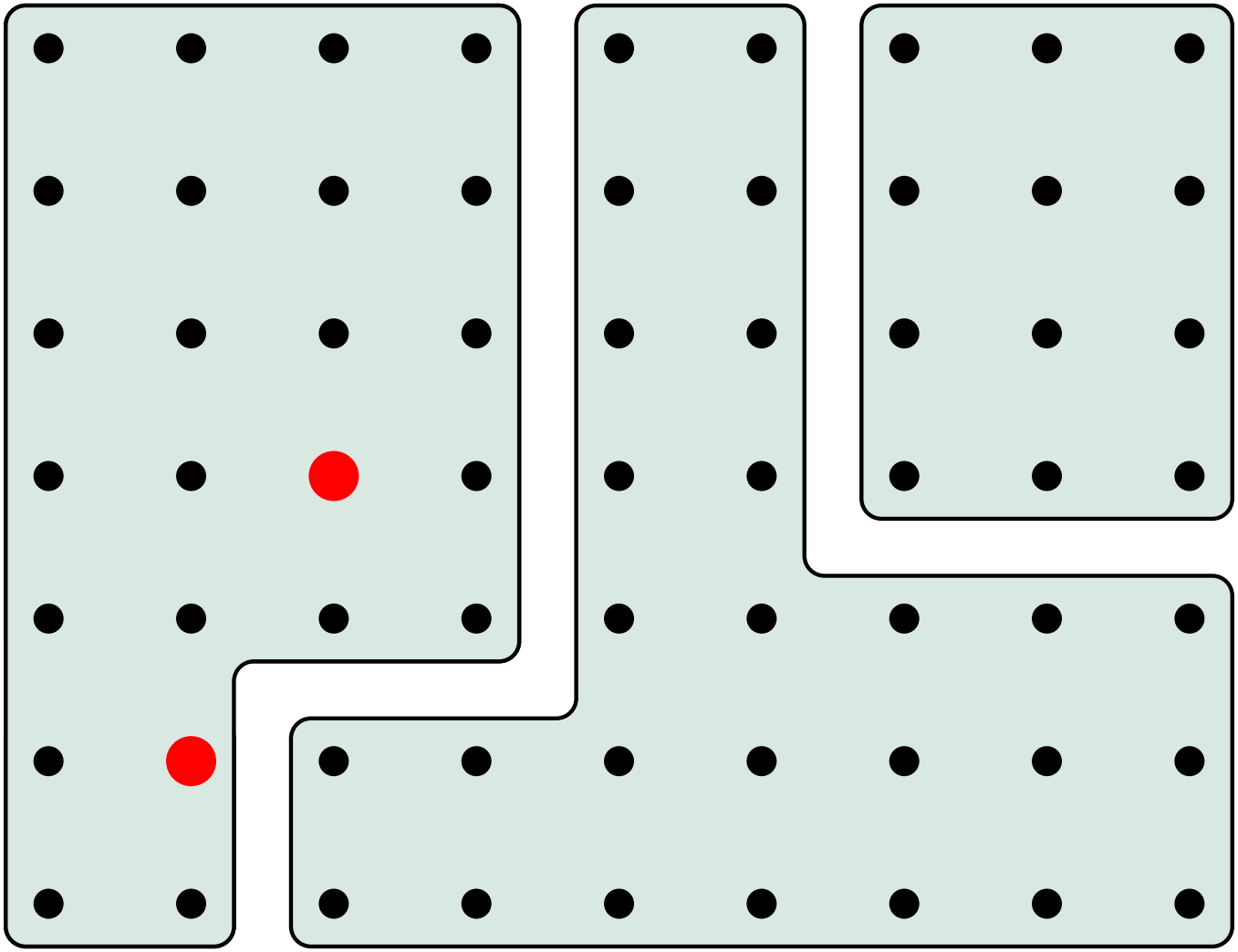
Union-Find Algorithm: Problem



Union-Find Algorithm: Problem

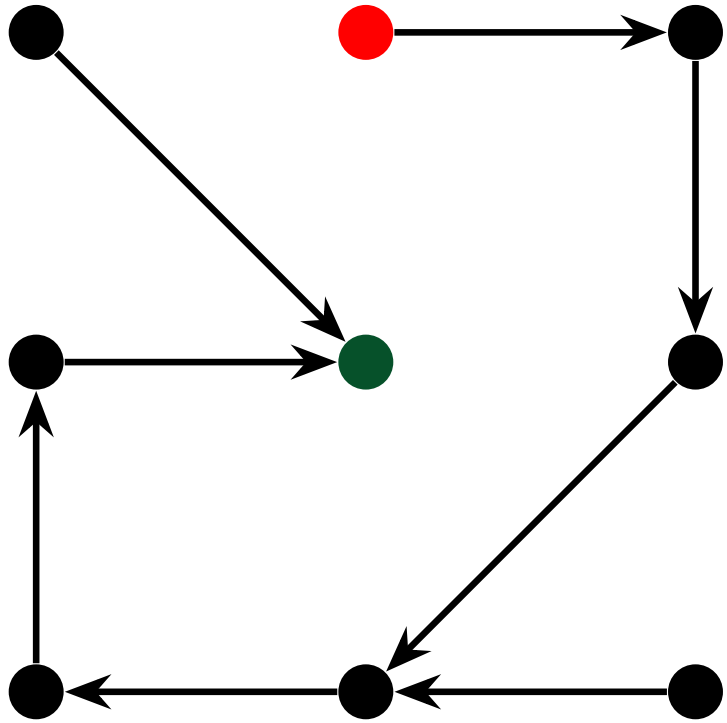


Union-Find Algorithm: Problem



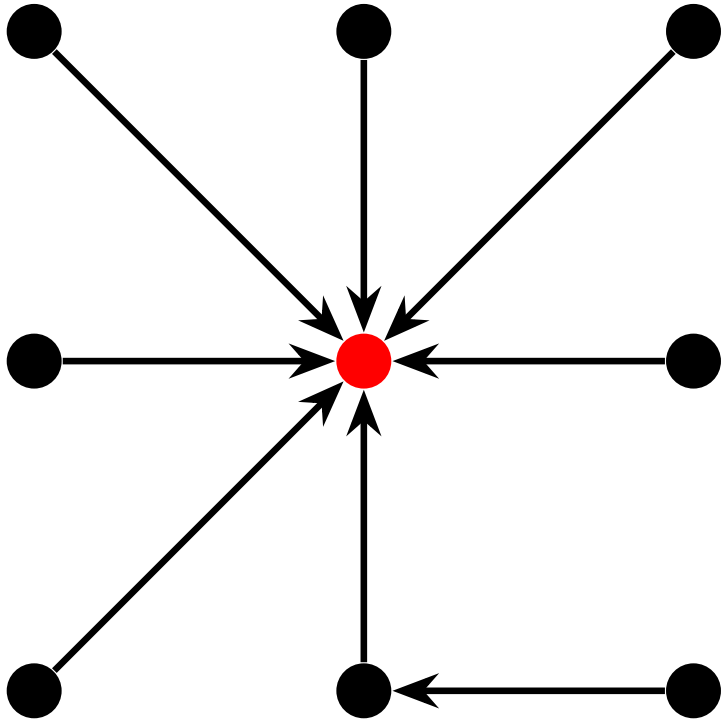
Union-Find Algorithm: Find

```
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;
};
```



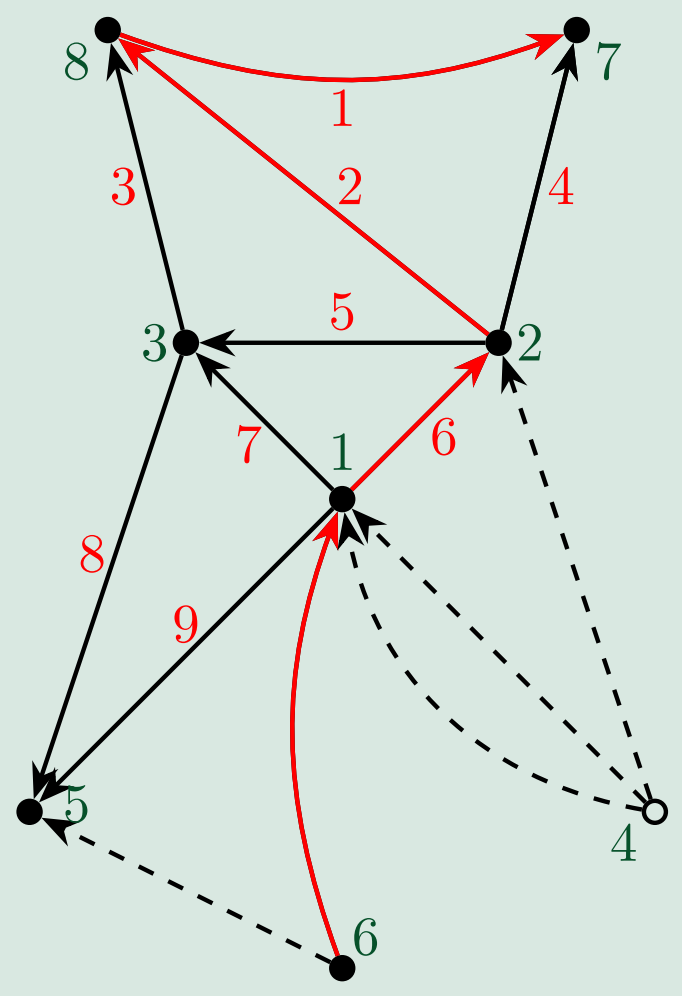
Union-Find Algorithm: The Specification of Find

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

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

- How to define γ, 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 γ' is ret?

Graph Library: Definition of Graph and Path



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

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

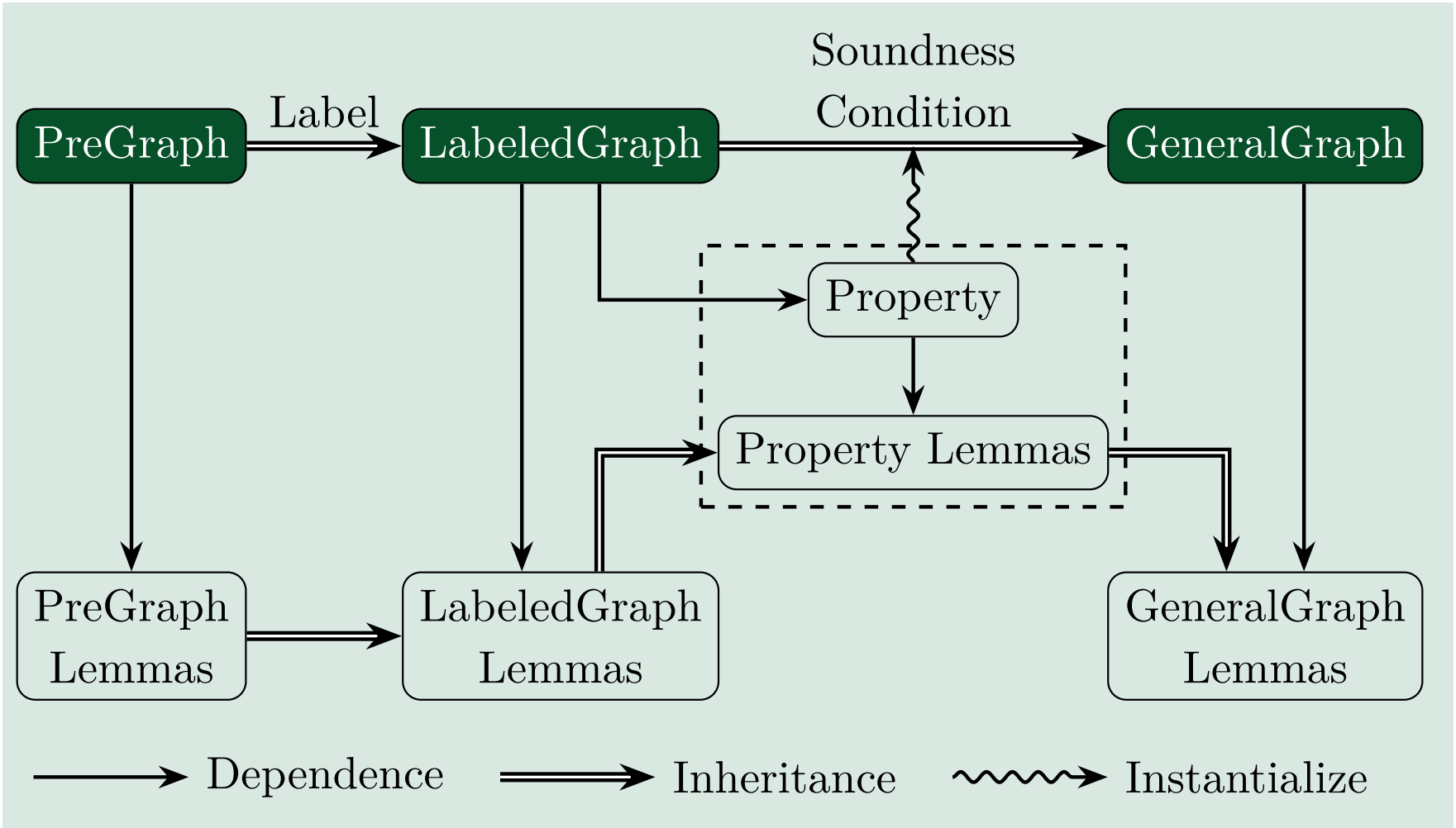
$$\text{GeneralGraph} \stackrel{\text{def}}{=} \{\text{LabeledGraph},\ \texttt{sound\_gg}\}$$

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

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

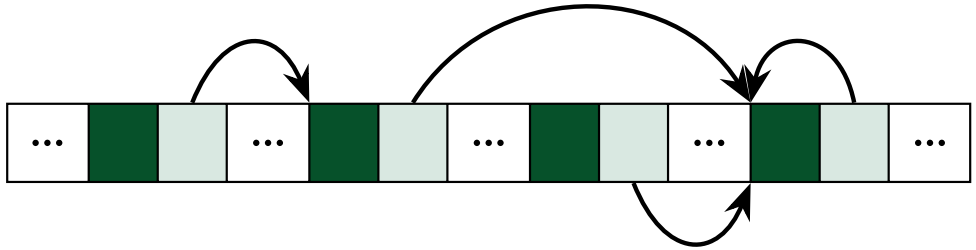
$$\gamma \vDash s \rightsquigarrow t \stackrel{\text{def}}{=} \exists p \text{ s.t. } \gamma \vDash s \overset{p}{\rightsquigarrow} t$$

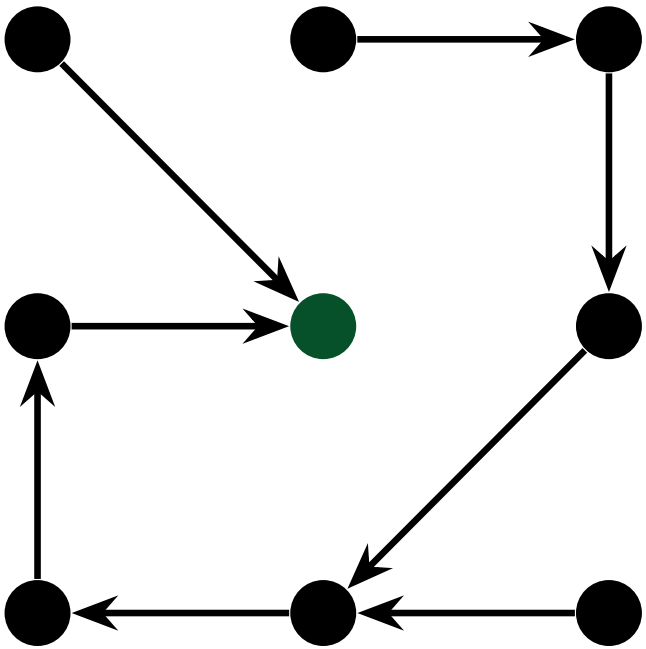
Architecture



Spatial Representation of Graphs

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





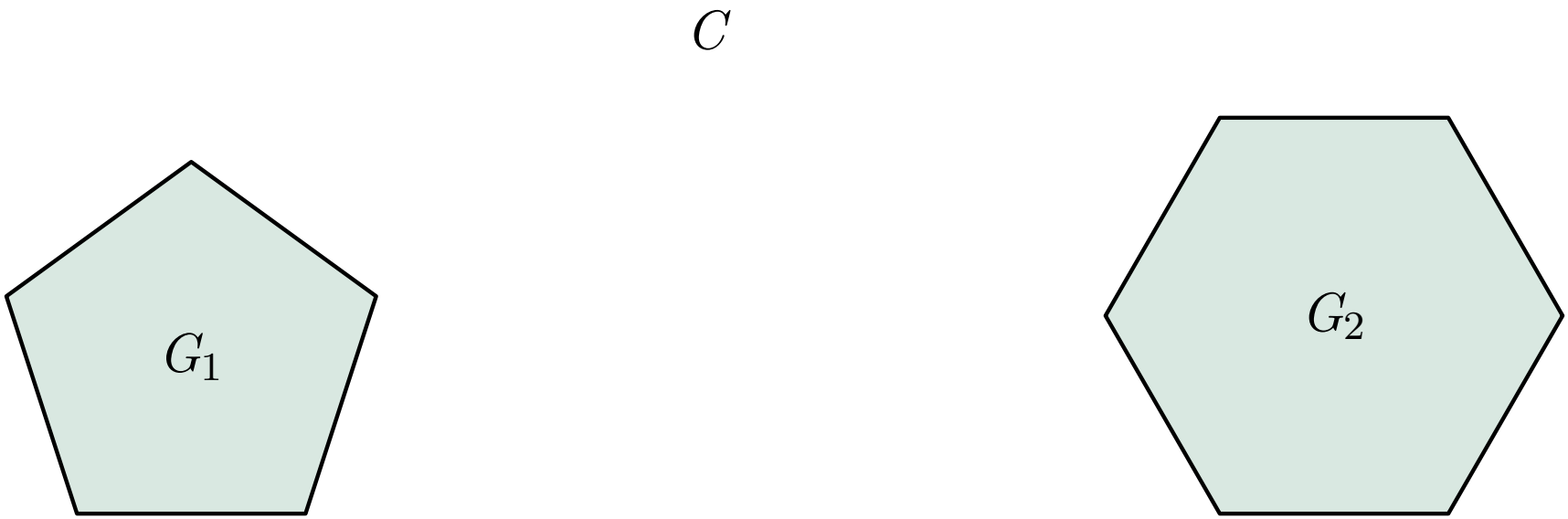
$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \bigstar_{\texttt{vvalid}(\gamma,v)} \texttt{v\_rep}(\gamma, v)$$

$$\bigstar_{\{v_1,v_2,\ldots,v_n\}} P \stackrel{\text{def}}{=} P(v_1) * P(v_2) * \cdots * P(v_n)$$

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

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

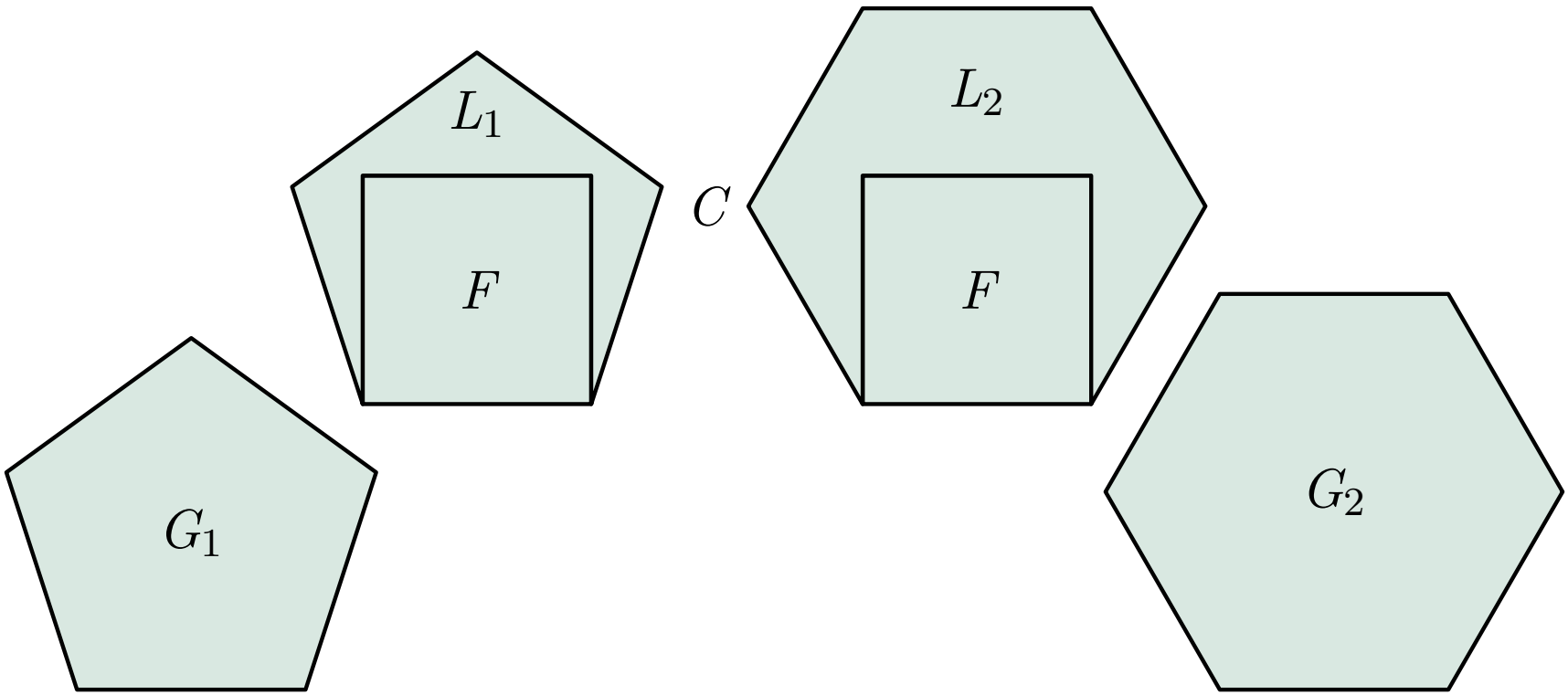
Ramify Rule



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

(Hobor and Villard)

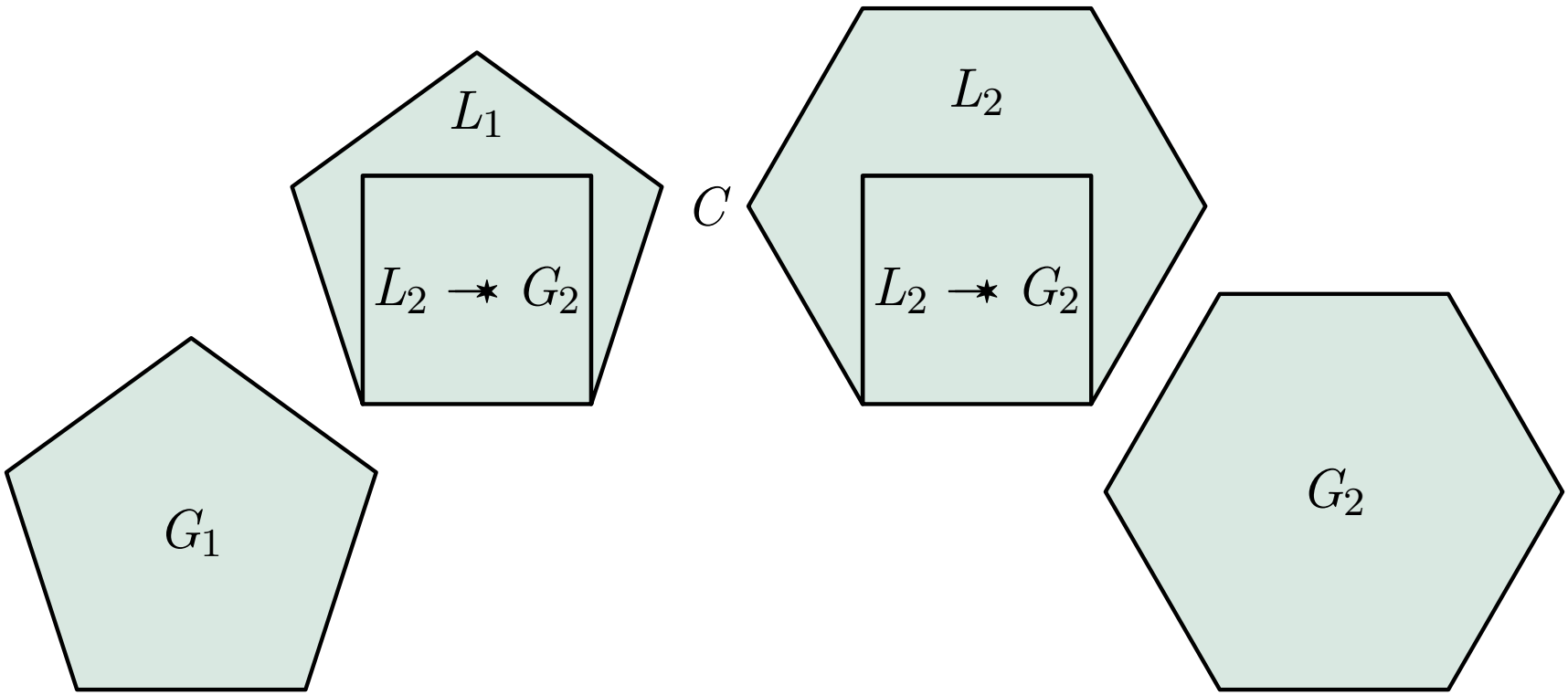
Ramify Rule



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

(Hobor and Villard)

Ramify Rule



$$\frac{\{L_1\}\, C\, \{L_2\} \quad G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)}{\{G_1\}\, C\, \{G_2\}}\ (\mathtt{mod}(C) \cap \mathtt{fv}(L_2 \mathrel{-\!\!*} G_2) = \varnothing)$$

(Hobor and Villard)

Our Localize Rule

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

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

Comparing to Hobor and Villard's Ramify rule:

$$\frac{\{L_1\}\ C\{L_2\} \quad G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)}{\{G_1\}\ C\{G_2\}}\ (\ddagger)$$

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

The Specification of Find

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

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

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

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

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

Proof Skeleton of Find

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

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

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

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

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

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

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

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

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

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

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

Proof Obligation of Find

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

$$\texttt{uf\_eq}(\gamma, \gamma_1) \Rightarrow \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \Rightarrow \texttt{dst}\big(\gamma, \texttt{out}(\texttt{x})\big) = \texttt{p}$$

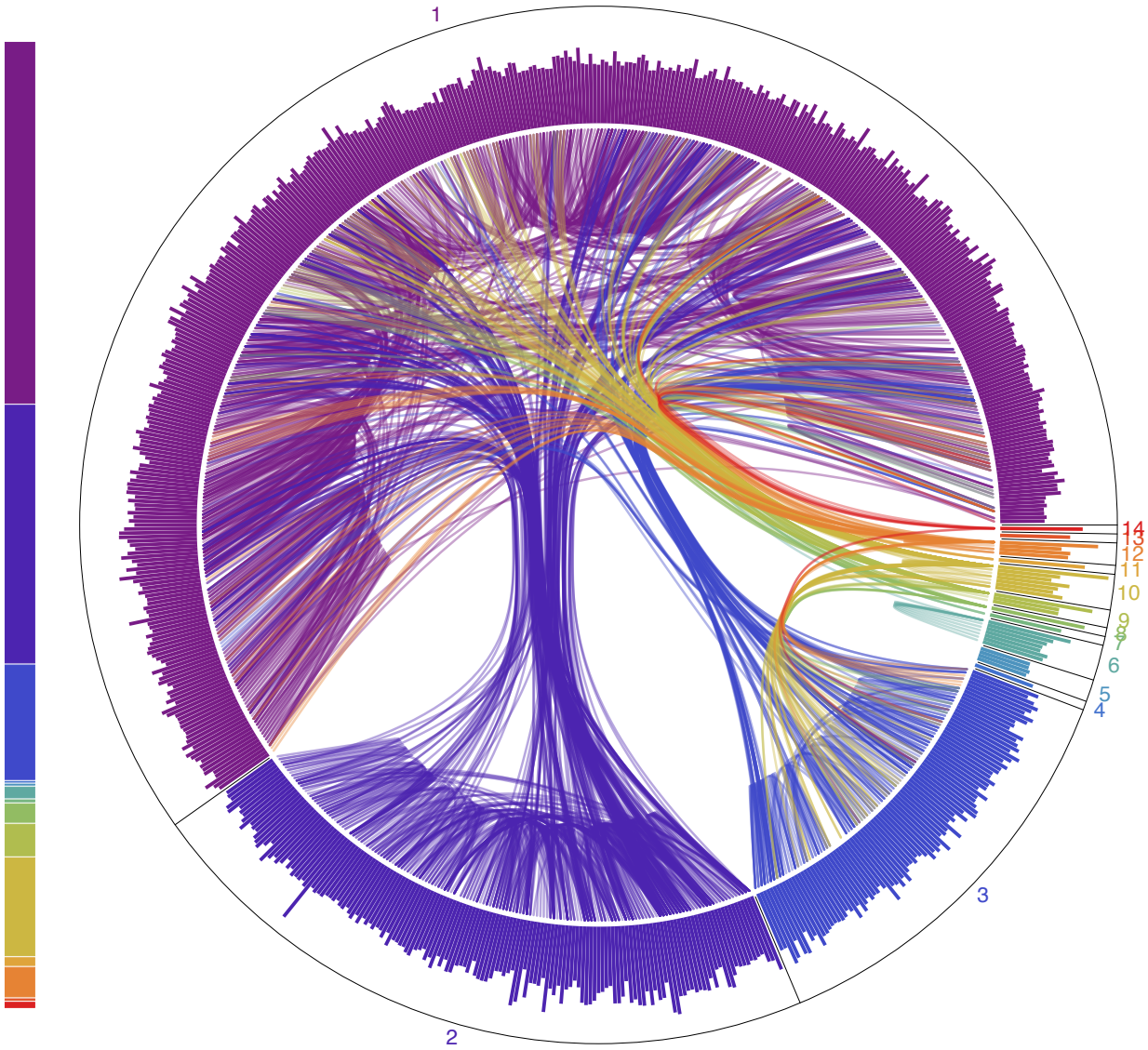
$$\gamma_2 = \texttt{redirect\_parent}(\gamma_1, \texttt{x}, \texttt{p0}) \Rightarrow$$

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

A Generational Garbage Collector

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



- 1: GCGraph.v
- 2: gc_correct.v
- 3: spatial_gcgraph.v
- 4: verif_ls_block.v
- 5: verif_conversion.v
- 6: verif_create_heap.v
- 7: verif_create_space.v
- 8: verif_do_generation.v
- 9: verif_do_scan.v
- 10: verif_forward.v
- 11: verif_forward_roots.v
- 12: verif_garbage_collect.v
- 13: verif_make_tinfo.v
- 14: verif_resume.v

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.