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

INRIA
July 29, 2019

Our Focus

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.

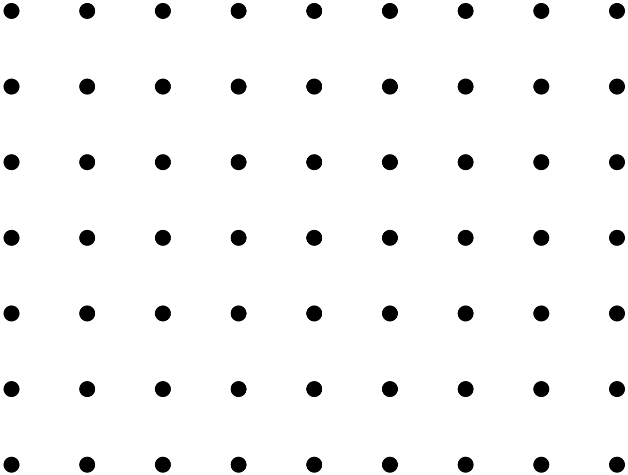
- Between them, CompCert and VST have 50+ person-years worth of development effort. It is highly desirous to fit within their frameworks rather than reinventing the wheel.
- We make no changes to CompCert. We make minimal (approximately 1% of codebase) additions to VST (two new tacticals, assorted lemmas).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

Our Results

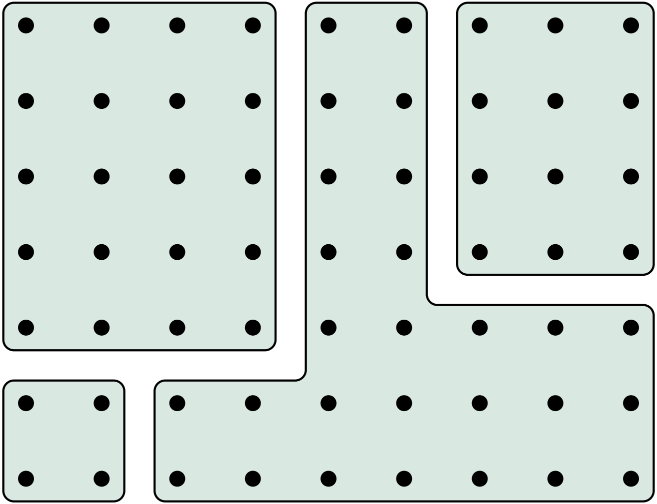
We have verified half a dozen graph algorithms, including:

- 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

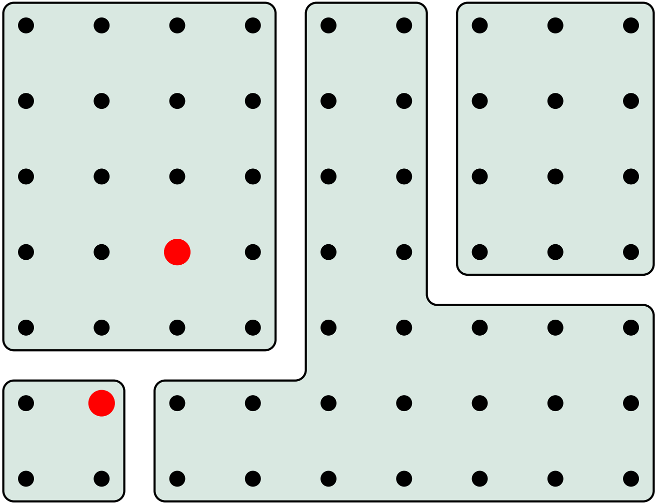
Union-Find Algorithm



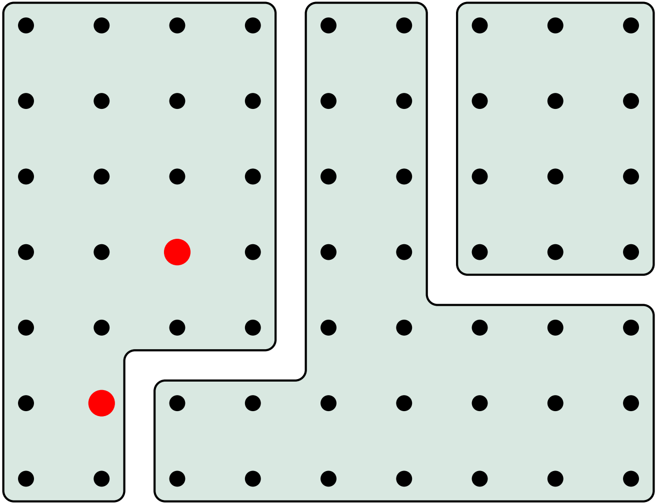
Union-Find Algorithm



Union-Find Algorithm



Union-Find Algorithm

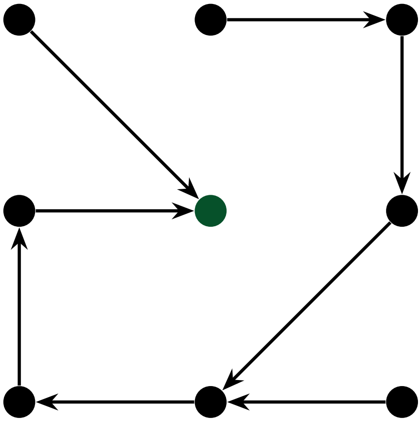


Union-Find Algorithm: Disjoint-Set Data Structure

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

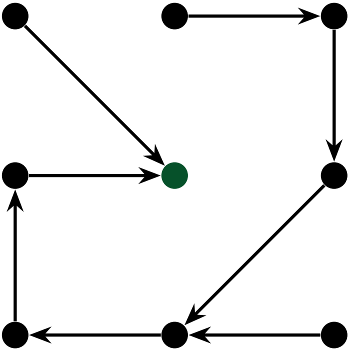
Union-Find Algorithm: Disjoint-Set Data Structure

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



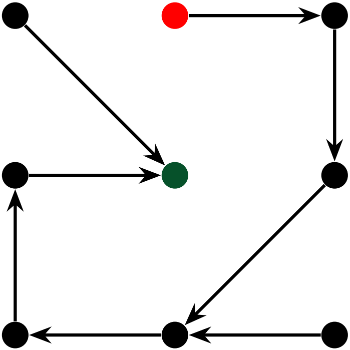
Union-Find Algorithm: Find

```
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;
};
```



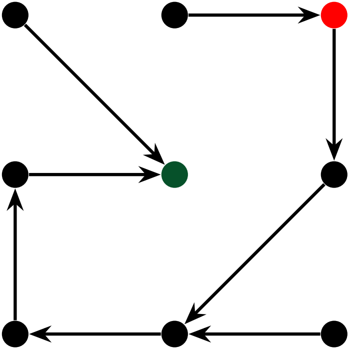
Union-Find Algorithm: Find

```
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;
};
```



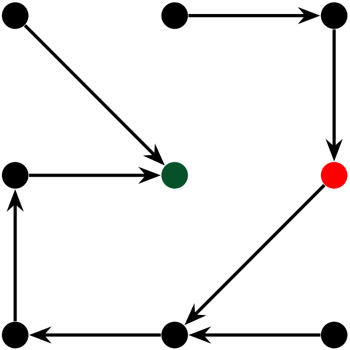
Union-Find Algorithm: Find

```
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;
};
```



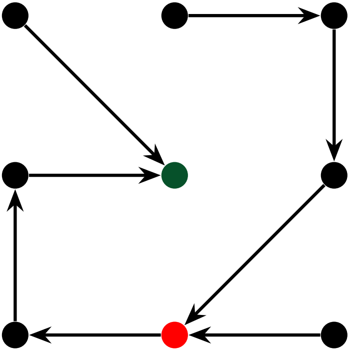
Union-Find Algorithm: Find

```
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;
};
```



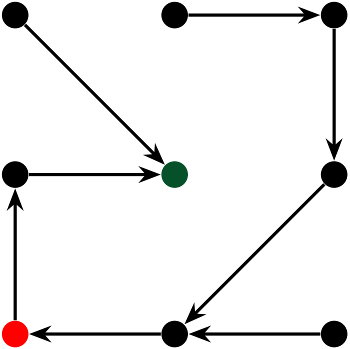
Union-Find Algorithm: Find

```
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;
};
```



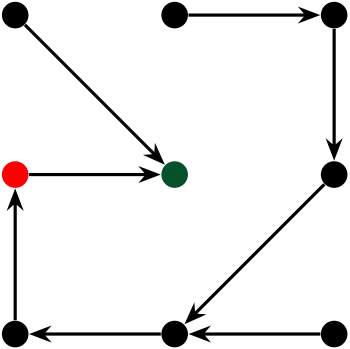
Union-Find Algorithm: Find

```
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;
};
```



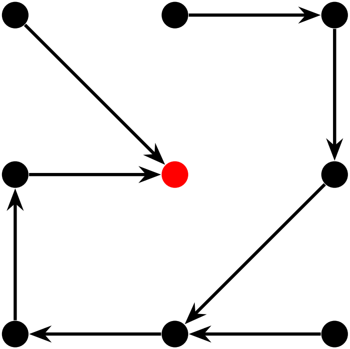
Union-Find Algorithm: Find

```
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;
};
```



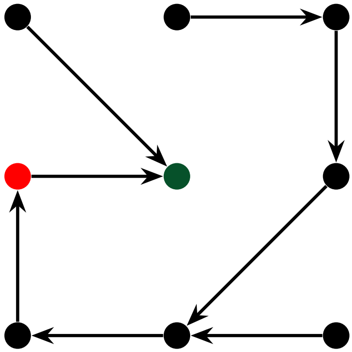
Union-Find Algorithm: Find

```
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;
};
```



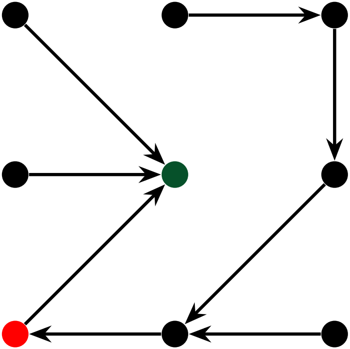
Union-Find Algorithm: Find

```
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;
};
```



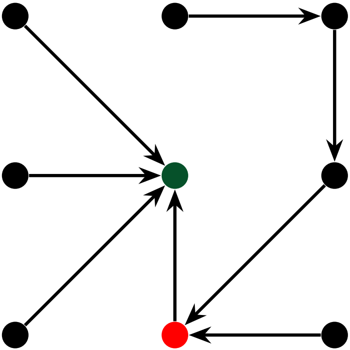
Union-Find Algorithm: Find

```
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;
};
```



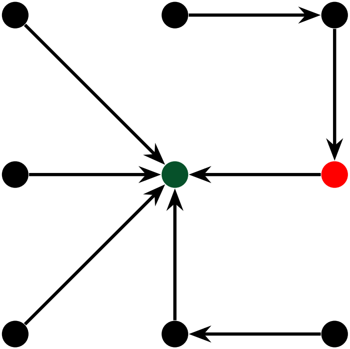
Union-Find Algorithm: Find

```
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;
};
```



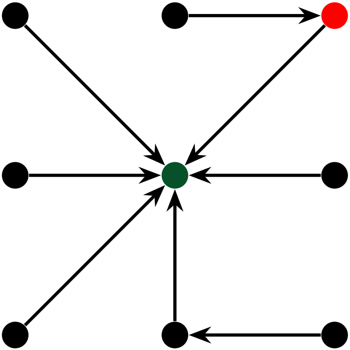
Union-Find Algorithm: Find

```
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;
};
```



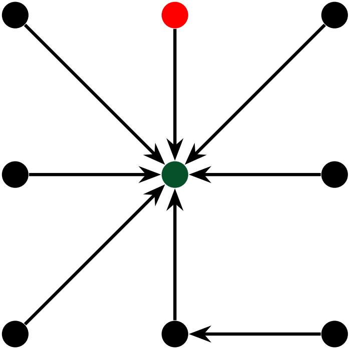
Union-Find Algorithm: Find

```
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;
};
```



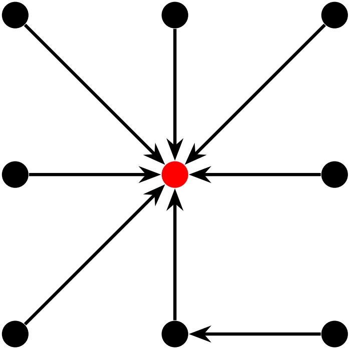
Union-Find Algorithm: Find

```
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;
};
```

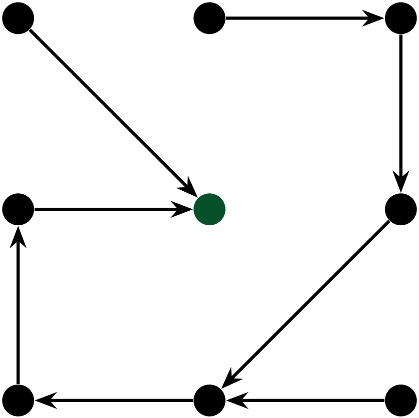


Union-Find Algorithm: Find

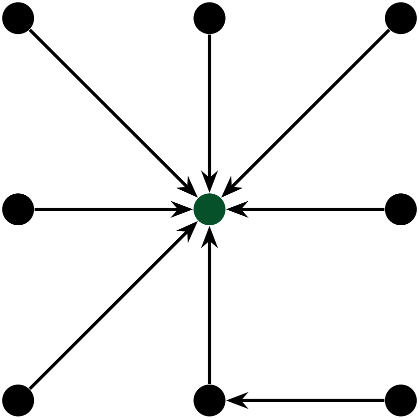
```
struct Node {
    unsigned int rank;
    struct Node *parent;
};
struct Node* find(struct Node* x) {
    struct Node *p, *p0;
    p = x -> parent;
    if (p != x) {
        p0 = find(p);
        p = p0;
        x -> parent = p;
    }
    return p;
};
```



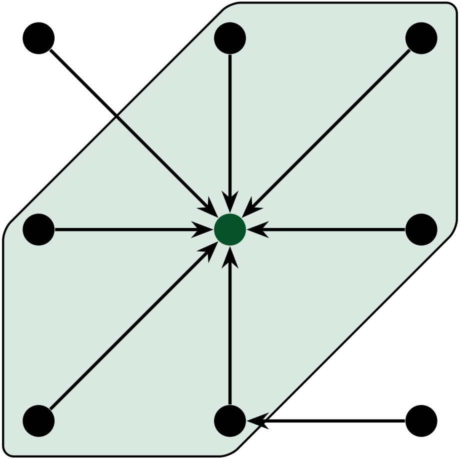
Verifying Graph-Manipulating Algorithm is Hard



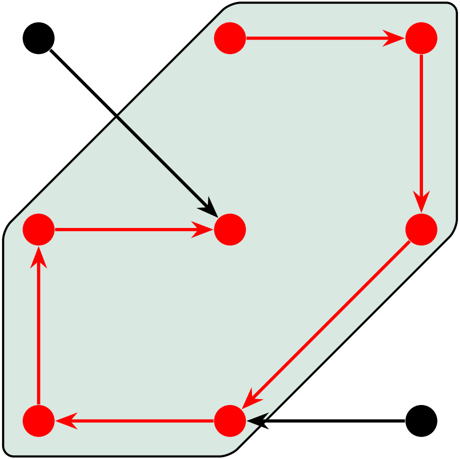
Verifying Graph-Manipulating Algorithm is Hard



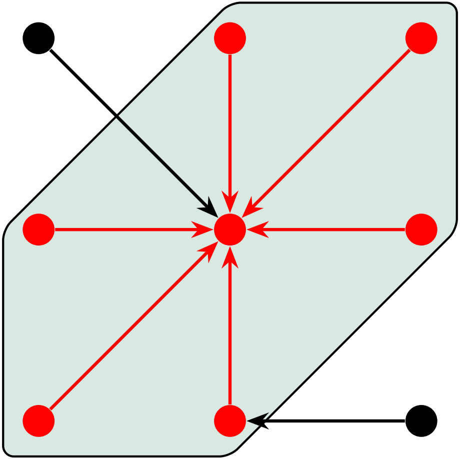
Verifying Graph-Manipulating Algorithm is Hard



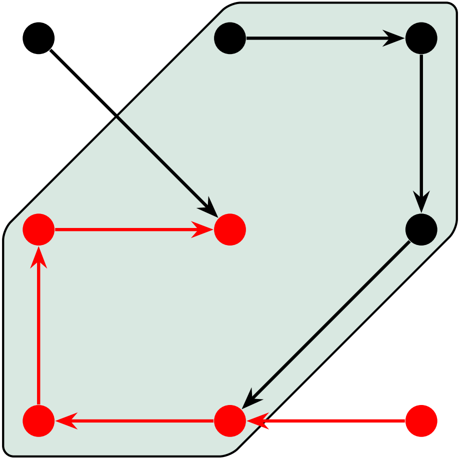
Verifying Graph-Manipulating Algorithm is Hard



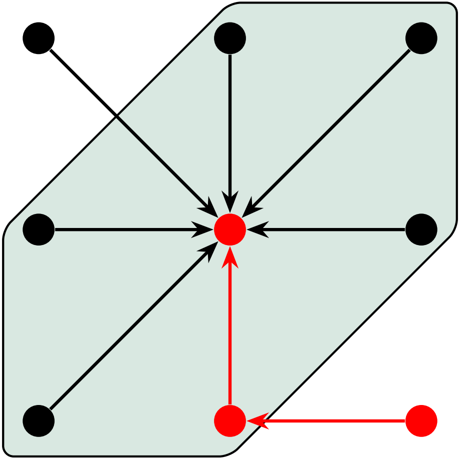
Verifying Graph-Manipulating Algorithm is Hard



Verifying Graph-Manipulating Algorithm is Hard



Verifying Graph-Manipulating Algorithm is Hard

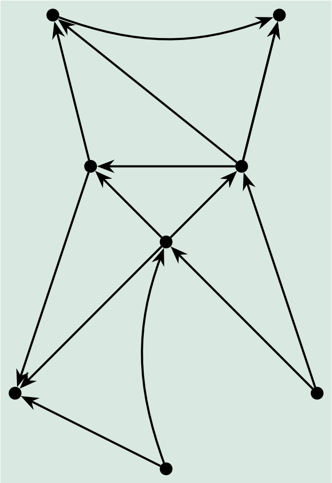


- Motivation ✓
- The Mathematical Graph Library
  - Core Definitions
  - Architecture
  - Selection of Properties
- The Spatial Representation of Graphs
  - CompCert and VST
  - Hoare Logic and Separation Logic
  - Spatial Representation of Graphs
  - Localize Rule
- Verification of the Find function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

Statistics

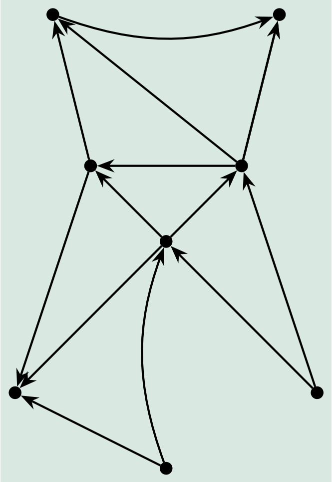
Component	**Files**	**LOC**
Common Utilities	10	2,842
Math Graph Library	19	12,723
Memory Model & Logic	13	2,373
Spatial Graph Library	10	6,458
Integration into VST	12	1,917
Examples (excluding GC)	13	3,290
GC, subdivided into	18	14,170
• mathematical graph	1	5,764
• spatial graph	1	1,618
• function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

Graph Library: A General Definition of Graph



A general definition of graph should have

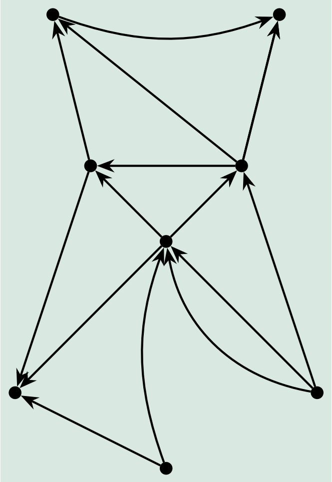
Graph Library: A General Definition of Graph



A general definition of graph should have

- Vertices
- Pairs of vertices as Edges

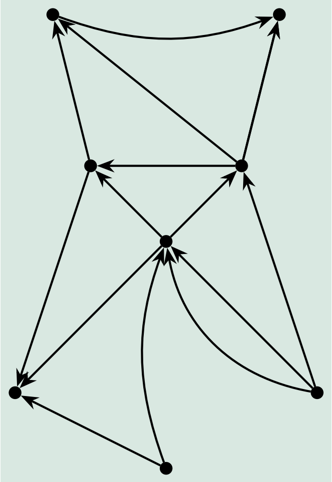
Graph Library: A General Definition of Graph



A general definition of graph should have

- Vertices
- Pairs of vertices as Edges

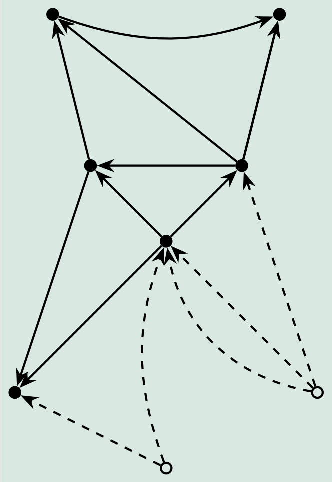
Graph Library: A General Definition of Graph



A general definition of graph should have

- Vertices
- Edges, sources and destinations

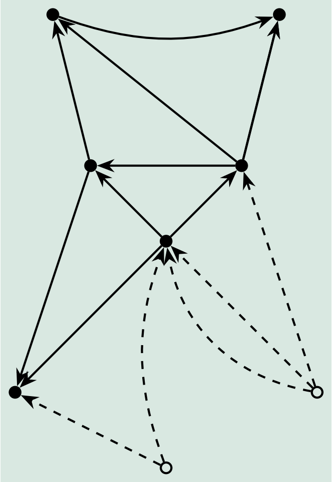
Graph Library: A General Definition of Graph



A general definition of graph should have

- Vertices
- Edges, sources and destinations

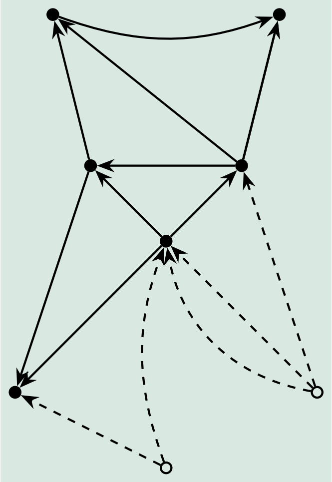
Graph Library: A General Definition of Graph



A general definition of graph should have

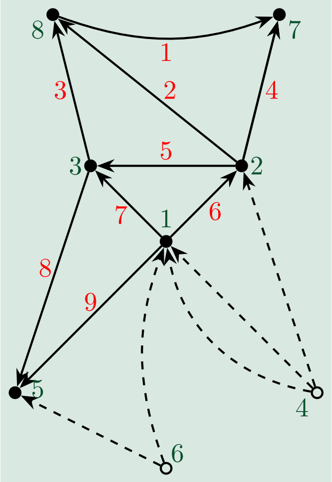
- Vertices
- Edges, sources and destinations
- Validity of vertices and edges

Graph Library: A General Definition of Graph



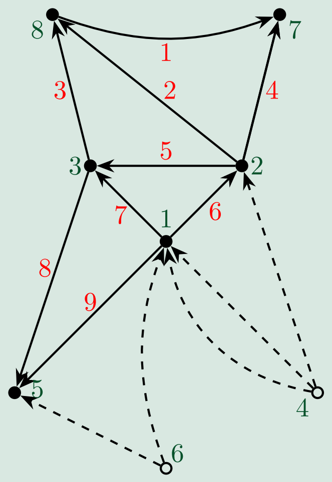
$$\text{PreGraph} \stackrel{\text{def}}{=} \{V,\, E,\, \texttt{vvalid},\, \texttt{evalid},\, \texttt{src},\, \texttt{dst}\}$$

Graph Library: A General Definition of Graph



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

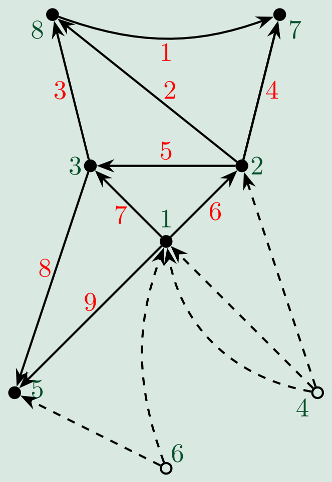
Graph Library: A General Definition of Graph



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

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

Graph Library: A General Definition of Graph

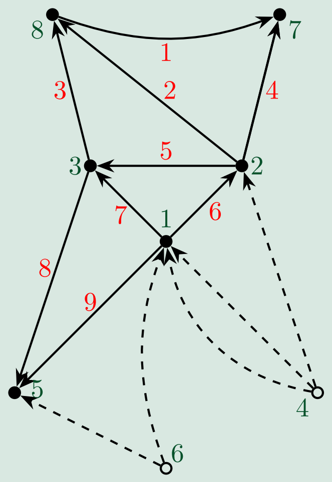


$$\text{PreGraph} \stackrel{\text{def}}{=} \{V, E, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}\}$$

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

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

Graph Library: A General Definition of Graph



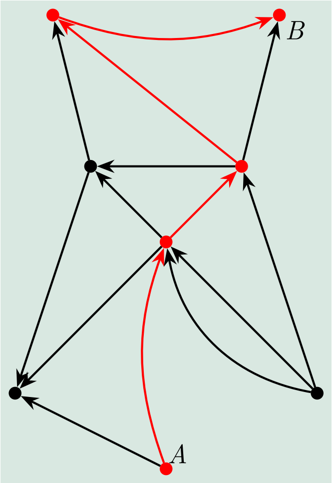
$$\text{PreGraph} \stackrel{\text{def}}{=} \{V, E, \texttt{vvalid}, \texttt{evalid}, \texttt{src}, \texttt{dst}\}$$

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

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

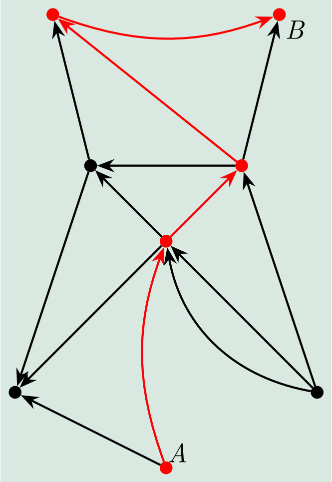
For Example: Acyclic

Graph Library: Definition of Path



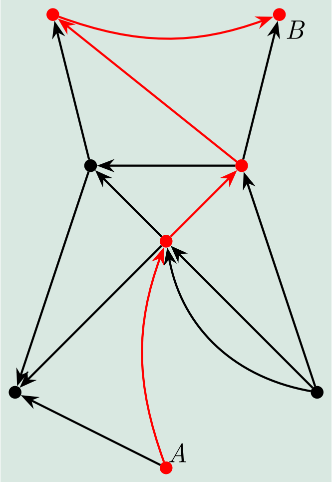
- Path is used in defining reachability.

Graph Library: Definition of Path



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

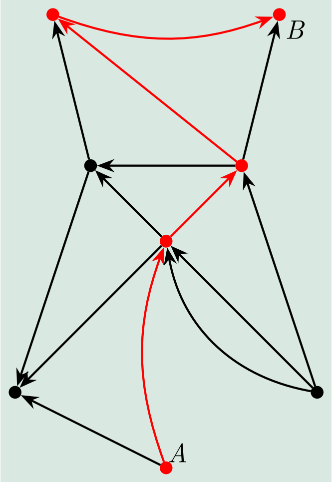
Graph Library: Definition of Path



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

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

Graph Library: Definition of Path

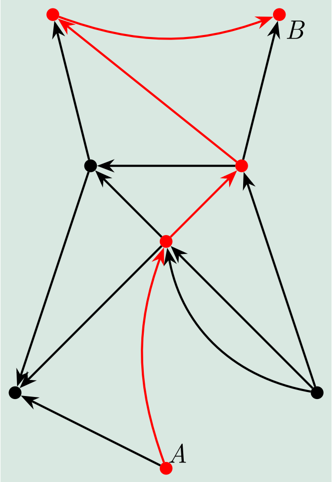


- Path is used in defining reachability.
- A path is a sequence of edges which connect a sequence of vertices.

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

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

Graph Library: Definition of Path



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

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

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

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

Other Derived Definitions: A Peek

$$\begin{aligned}\texttt{s\_evalid}(\gamma, e) \stackrel{\text{def}}{=} {} & \texttt{evalid}(\gamma, e) \wedge \\ & \texttt{vvalid}(\gamma, \texttt{src}(\gamma, e)) \wedge \texttt{vvalid}(\gamma, \texttt{dst}(\gamma, e))\end{aligned}$$

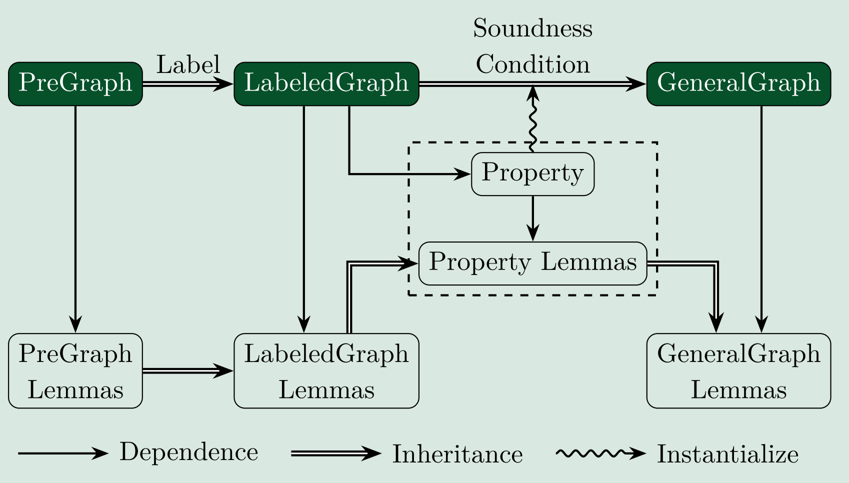
Other Derived Definitions: A Peek

$$\begin{aligned}
\texttt{s_evalid}(\gamma, e) &\stackrel{\text{def}}{=} \texttt{evalid}(\gamma, e) \wedge \\
&\quad\ \texttt{vvalid}(\gamma, \texttt{src}(\gamma, e)) \wedge \texttt{vvalid}(\gamma, \texttt{dst}(\gamma, e)) \\
\texttt{valid_path}\big(\gamma, (v, [])\big) &\stackrel{\text{def}}{=} \texttt{vvalid}(\gamma, v) \\
\texttt{valid_path}\big(\gamma, (v, [e_1, e_2, \ldots, e_n])\big) &\stackrel{\text{def}}{=} v = \texttt{src}(\gamma, e_1) \wedge \texttt{s_evalid}(\gamma, e_1) \wedge \\
&\quad\ \texttt{dst}(\gamma, e_1) = \texttt{src}(\gamma, e_2) \wedge \\
&\quad\ \texttt{s_evalid}(\gamma, e_2) \wedge \ldots
\end{aligned}$$

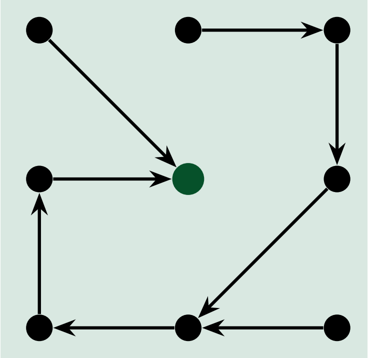
Other Derived Definitions: A Peek

$$\begin{aligned}
\texttt{s_evalid}(\gamma, e) &\stackrel{\text{def}}{=} \texttt{evalid}(\gamma, e) \land \\
&\quad\ \texttt{vvalid}(\gamma, \texttt{src}(\gamma, e)) \land \texttt{vvalid}(\gamma, \texttt{dst}(\gamma, e)) \\
\texttt{valid_path}\big(\gamma, (v, [\,])\big) &\stackrel{\text{def}}{=} \texttt{vvalid}(\gamma, v) \\
\texttt{valid_path}\big(\gamma, (v, [e_1, e_2, \ldots, e_n])\big) &\stackrel{\text{def}}{=} v = \texttt{src}(\gamma, e_1) \land \texttt{s_evalid}(\gamma, e_1) \land \\
&\quad\ \texttt{dst}(\gamma, e_1) = \texttt{src}(\gamma, e_2) \land \\
&\quad\ \texttt{s_evalid}(\gamma, e_2) \land \ldots \\
\texttt{end}\big(\gamma, (v, [\,])\big) &\stackrel{\text{def}}{=} v \\
\texttt{end}\big(\gamma, (v, [e_1, e_2, \ldots, e_n])\big) &\stackrel{\text{def}}{=} \texttt{dst}(\gamma, e_n) \\
\gamma \models s \stackrel{p}{\leadsto} t &\stackrel{\text{def}}{=} \texttt{valid_path}(\gamma, p) \land \texttt{fst}(p) = s \land \texttt{end}(\gamma, p) = t \\
\gamma \models s \leadsto t &\stackrel{\text{def}}{=} \exists p \text{ s.t. } \gamma \models s \stackrel{p}{\leadsto} t
\end{aligned}$$

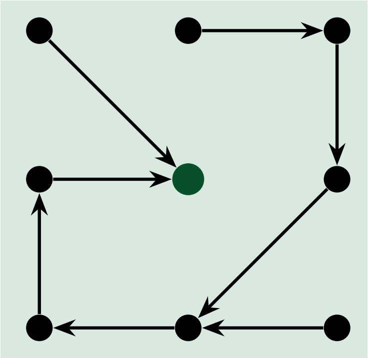
Architecture



Various Properties: MathGraph, LstGraph and FiniteGraph



Various Properties: MathGraph, LstGraph and FiniteGraph



$$\mathrm{MathGraph}(\gamma) \stackrel{\text{def}}{=} \Big\{$$

$$\texttt{null} : V$$

$$\texttt{weak\_valid}(v) \stackrel{\text{def}}{=} v = \texttt{null} \vee \texttt{vvalid}(\gamma, v)$$

$$\texttt{valid\_graph} : \forall e.\, \texttt{evalid}(\gamma, e) \Rightarrow$$

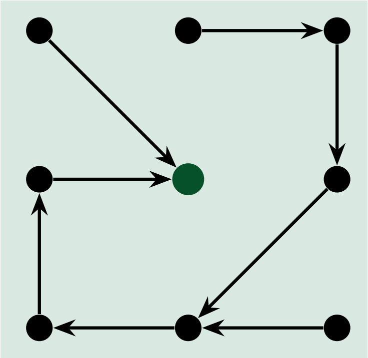
$$\texttt{vvalid}\Big(\gamma, \texttt{src}(\gamma, e)\Big) \wedge$$

$$\texttt{weak\_valid}\Big(\texttt{dst}(\gamma, e)\Big)$$

$$\texttt{valid\_not\_null} : \forall v.\, \texttt{vvalid}(\gamma, v) \Rightarrow$$

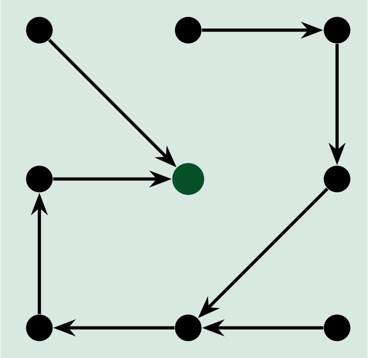
$$v \neq \texttt{null} \Big\}$$

Various Properties: MathGraph, LstGraph and FiniteGraph



$$
\begin{aligned}
\text{LstGraph}(\gamma) \stackrel{\text{def}}{=} \Big\{ & \\
\texttt{out} :\ & V \to E \\
\texttt{only\_one\_edge} :\ & \forall v,\ e.\, \texttt{vvalid}(\gamma, v) \Rightarrow \\
& \Big(\texttt{src}(\gamma, e) = v \wedge \\
& \texttt{evalid}(\gamma, e)\Big) \Leftrightarrow \\
& e = \texttt{out}(v) \\
\texttt{acyclic\_path} :\ & \forall v,\ p.\, \gamma \models v \overset{p}{\leadsto} v \Rightarrow \\
& p = (v, []) \Big\}
\end{aligned}
$$

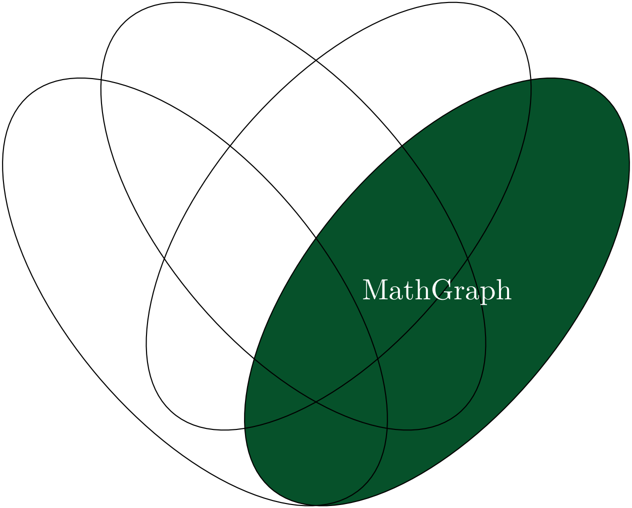
Various Properties: MathGraph, LstGraph and FiniteGraph



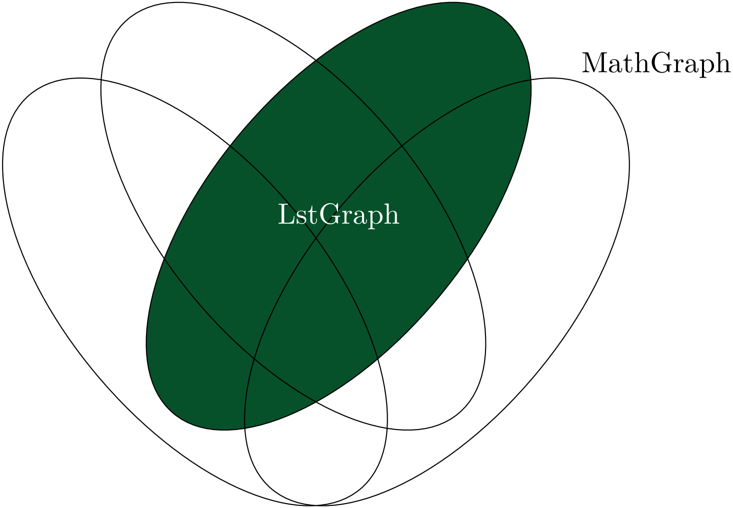
$$\text{FiniteGraph}(\gamma) \stackrel{\text{def}}{=} \Big\{$$

$$\begin{aligned}\texttt{finite\_v} : &\ \exists\, S_v,\ M_v \text{ s.t. } |S_v| \leqslant M_v \land \\ &\ \forall v.\, \texttt{vvalid}(\gamma, v) \Rightarrow v \in S_v \\ \texttt{finite\_e} : &\ \exists\, S_e,\ M_e \text{ s.t. } |S_e| \leqslant M_e \land \\ &\ \forall e.\, \texttt{evalid}(\gamma, e) \Rightarrow e \in S_e \Big\}\end{aligned}$$

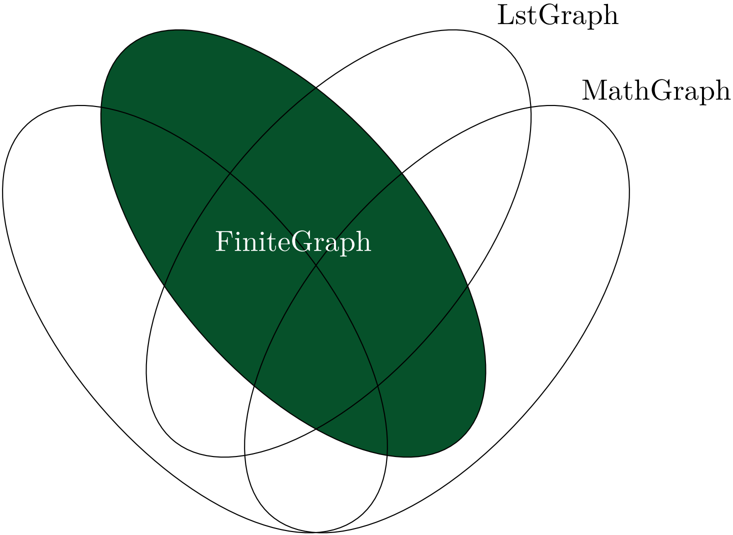
Various Properties



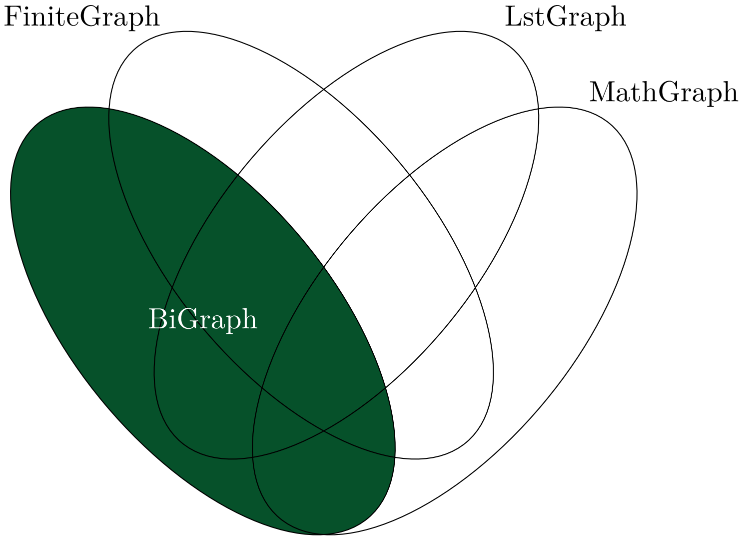
Various Properties



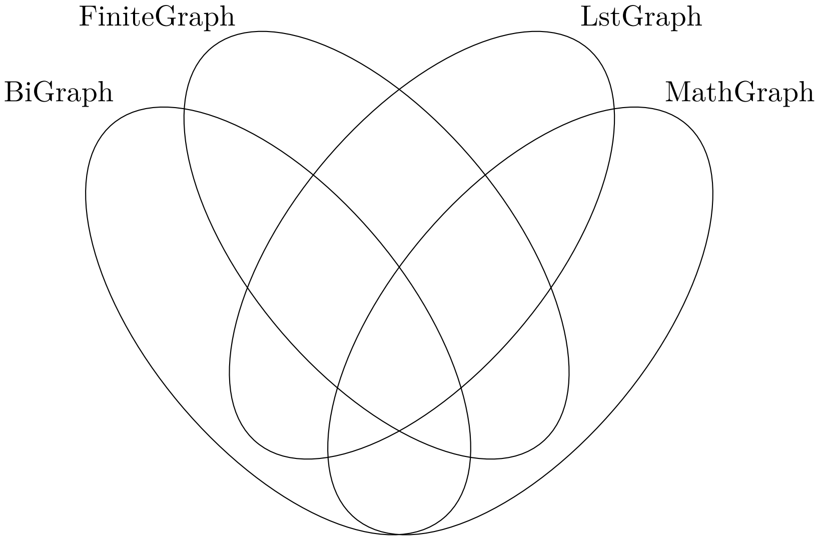
Various Properties



Various Properties

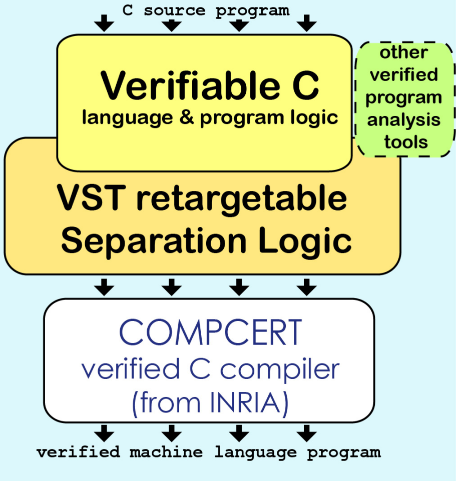


Various Properties



- 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

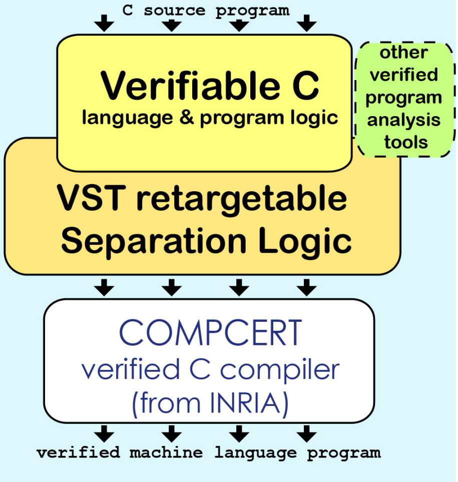
CompCert and VST



- CompCert

(Leroy et al. , Appel et al.)

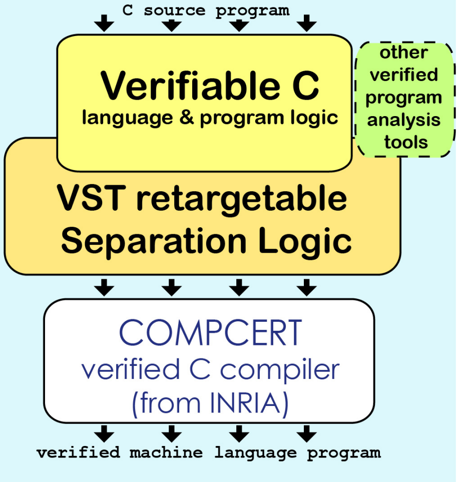
CompCert and VST



- CompCert
  - C → Coq (Clight) → Machine

(Leroy et al. , Appel et al.)

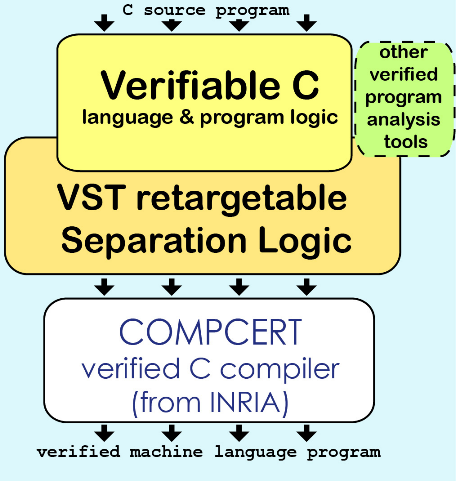
CompCert and VST



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

(Leroy et al. , Appel et al.)

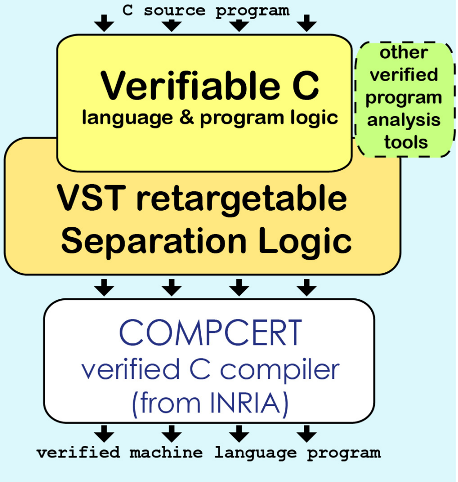
CompCert and VST



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

(Leroy et al. , Appel et al.)

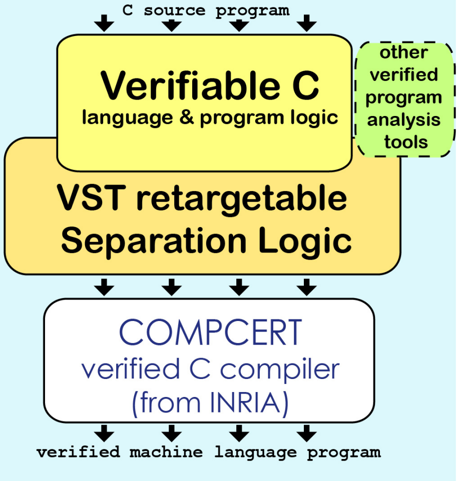
CompCert and VST



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

(Leroy et al. , Appel et al.)

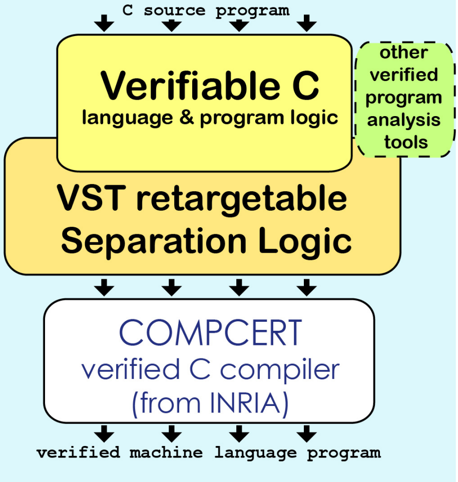
CompCert and VST



- 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

$$\{P\}\, C\, \{Q\}$$

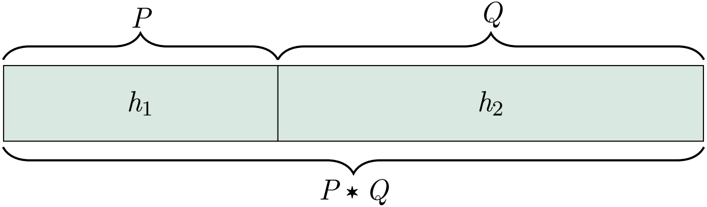
(C. A. R. Hoare)

Recap: Separation Logic

$$P \star Q$$

(Reynolds et al.)

Recap: Separation Logic



$$h \models P \star Q \stackrel{\text{def}}{=} \exists\, h_1, h_2 \text{ s.t. } h_1 \oplus h_2 = h \wedge h_1 \models P \wedge h_2 \models Q$$

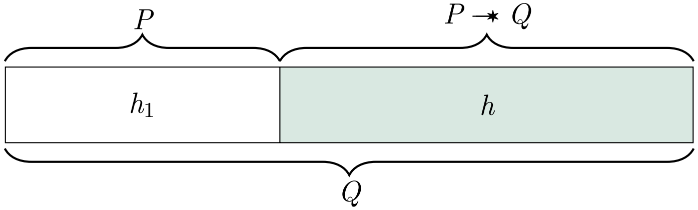
(Reynolds et al.)

Recap: Separation Logic

$P \mathrel{-\!\!*} Q$

(Reynolds et al.)

Recap: Separation Logic



$$h \models P \mathrel{-\!\!*} Q \overset{\text{def}}{=} \forall h_1, h_2\,.\, h_1 \oplus h = h_2 \Rightarrow h_1 \models P \Rightarrow h_2 \models Q$$

(Reynolds et al.)

Recap: Separation Logic

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

(Reynolds et al.)

Recap: Separation Logic

`emp`

(Reynolds et al.)

Recap: Separation Logic

$$a \mapsto v$$

(Reynolds et al.)

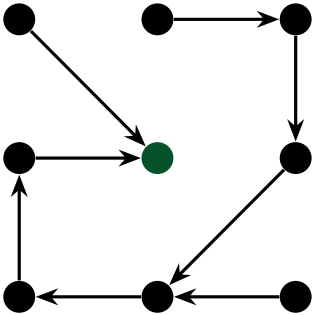
Recap: Separation Logic

$$\frac{\{P\}\ C\,\{Q\}}{\{P \star F\}\ C\,\{Q \star F\}}(\mathtt{mod}(C) \cap \mathtt{fv}(F) = \varnothing)$$

(Reynolds et al.)

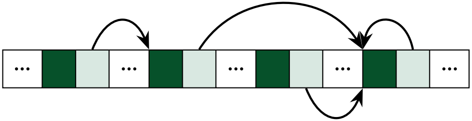
Spatial Representation of Graphs

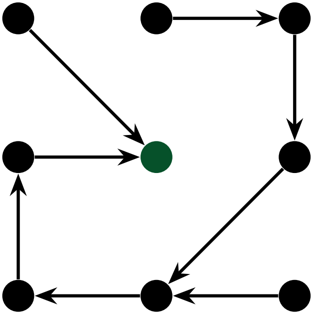
```
struct Node {
    unsigned int rank;
    struct Node *parent;
};
```



Spatial Representation of Graphs

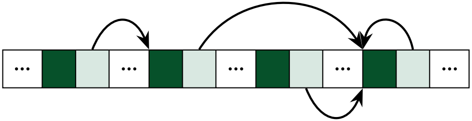
```
struct Node {
    unsigned int rank;
    struct Node *parent;
};
```

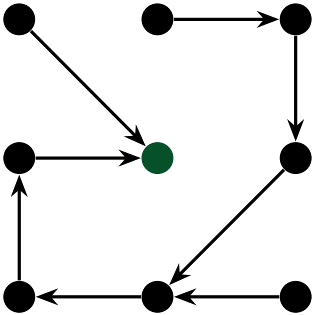




Spatial Representation of Graphs

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

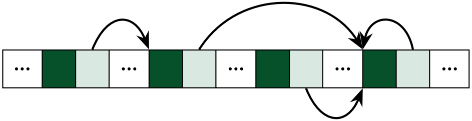


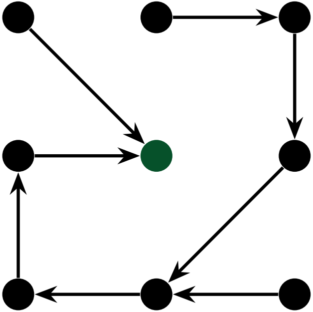


$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma, v)} \texttt{v\_rep}(\gamma, v)$$

Spatial Representation of Graphs

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



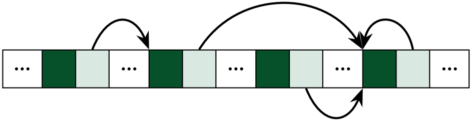


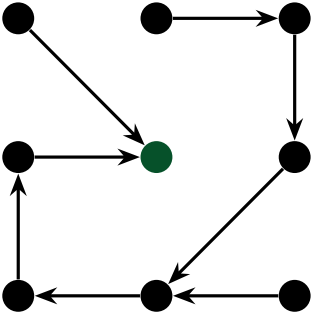
$$\mathtt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\mathtt{vvalid}(\gamma,v)} \mathtt{v\_rep}(\gamma, v)$$

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

Spatial Representation of Graphs

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





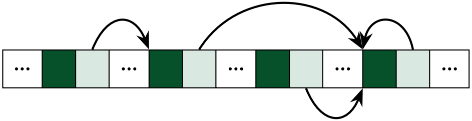
$$\mathtt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\mathtt{vvalid}(\gamma,v)} \mathtt{v\_rep}(\gamma, v)$$

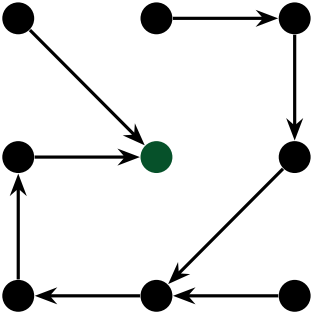
$$\mathop{\bigstar}_{\{v_1,v_2,\ldots,v_n\}} P \stackrel{\text{def}}{=} P(v_1) \star P(v_2) \star \cdots \star P(v_n)$$

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

Spatial Representation of Graphs

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





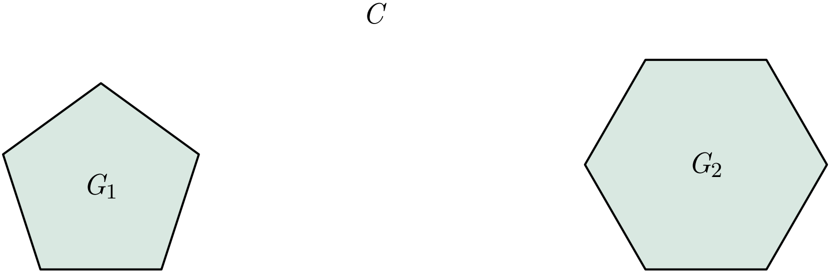
$$\texttt{graph\_rep}(\gamma) \stackrel{\text{def}}{=} \mathop{\bigstar}_{\texttt{vvalid}(\gamma,v)} \texttt{v\_rep}(\gamma, v)$$

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

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

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

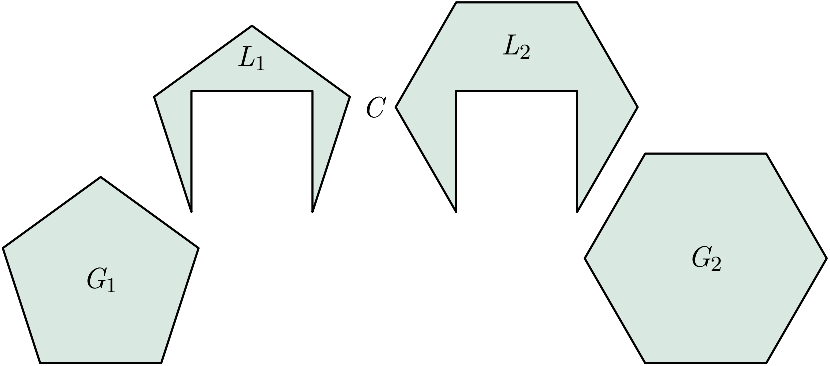
Ramify Rule



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

(Hobor and Villard)

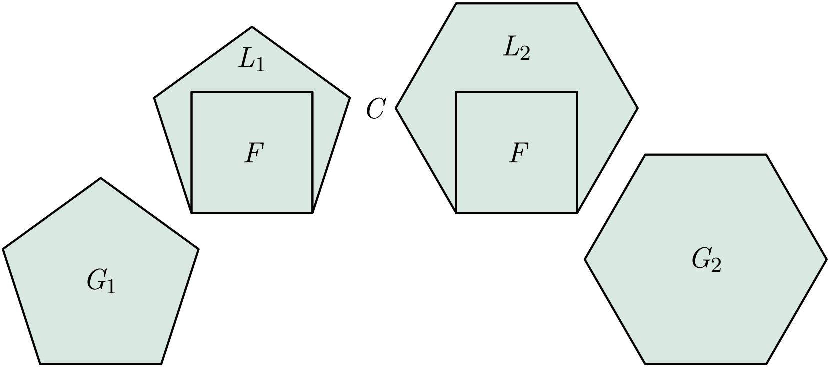
Ramify Rule



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

(Hobor and Villard)

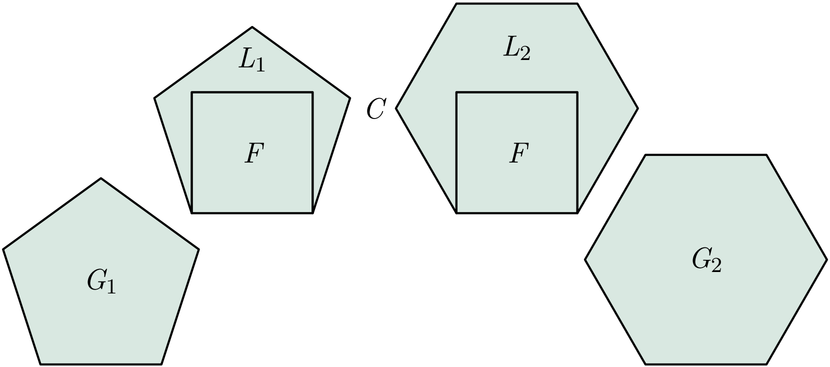
Ramify Rule



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

(Hobor and Villard)

Ramify Rule

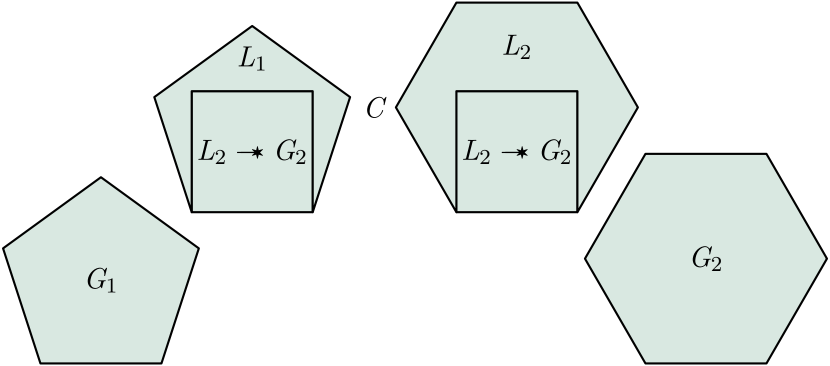


$$\frac{\{L_1\}\ C\,\{L_2\}}{\{G_1\}\ C\,\{G_2\}}$$

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

(Hobor and Villard)

Ramify Rule

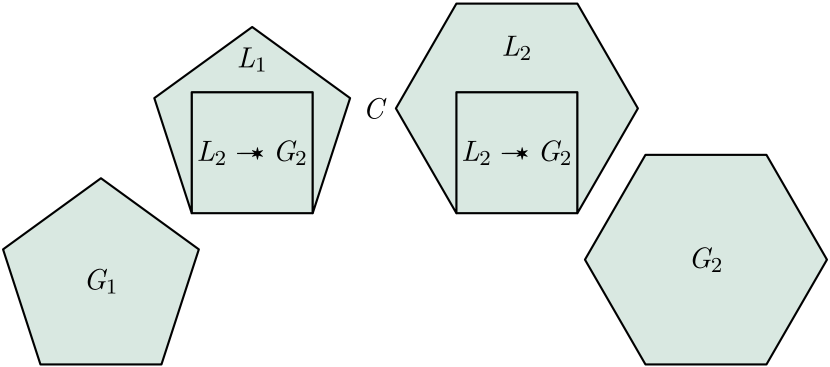


$$\frac{\{L_1\}\ C\ \{L_2\}}{\{G_1\}\ C\ \{G_2\}}$$

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

(Hobor and Villard)

Ramify Rule



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

(Hobor and Villard)

Localize Rule

$$\frac{\{L_1\}\ C\ \{\qquad L_2\} \qquad G_1 \vdash L_1 \star R \qquad R \vdash \qquad L_2 \mathrel{-\!\!\star} G_2}{\{G_1\}\ C\ \{\qquad G_2\}}$$

Localize Rule

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

Localize Rule

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

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

Localize Rule

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

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

Comparing to Ramify rule:

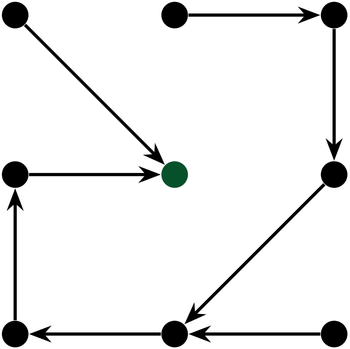
$$\frac{\{L_1\}\ C\,\{L_2\} \quad G_1 \vdash L_1 \star (L_2 \mathrel{-\!\!\star} G_2)}{\{G_1\}\ C\,\{G_2\}}\ (\ddagger)$$

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

- 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

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

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

The Specification of Find

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

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

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

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

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

Proof Skeleton of Find

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

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

```
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);
```

```
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

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

```
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

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

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

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

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

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

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

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

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

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

```
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
```

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

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

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

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);
```

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

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

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

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

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

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

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);
```

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

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

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

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

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

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

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);
```

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

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

```
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 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 Obligation of Find

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

Proof Obligation of Find

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

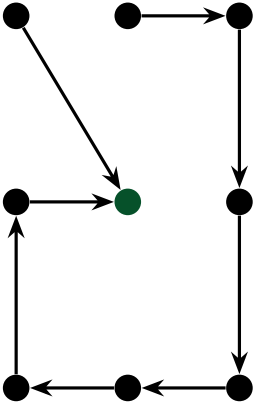
$$\texttt{uf\_eq}(\gamma, \gamma_1) \Rightarrow \texttt{root}(\gamma_1, \texttt{p}, \texttt{p0}) \Rightarrow \texttt{dst}\big(\gamma, \texttt{out}(\texttt{x})\big) = \texttt{p}$$

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

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

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

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

The same specification but a different representation

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

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

$$\begin{aligned}\texttt{graph\_rep}(g, s) \quad \stackrel{\text{def}}{=} \quad & \exists n. \Big(\forall v.\ 0 \leqslant v < n \iff \texttt{vvalid}(\gamma, v) \wedge \\ & \big(n \leqslant \text{MaxInt}/8\big) \wedge \\ & s \mapsto \texttt{map}(\lambda v.\ \texttt{v\_rep}(\gamma, v))\ [0, 1, 2, \ldots, n]\Big)\end{aligned}$$

- 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

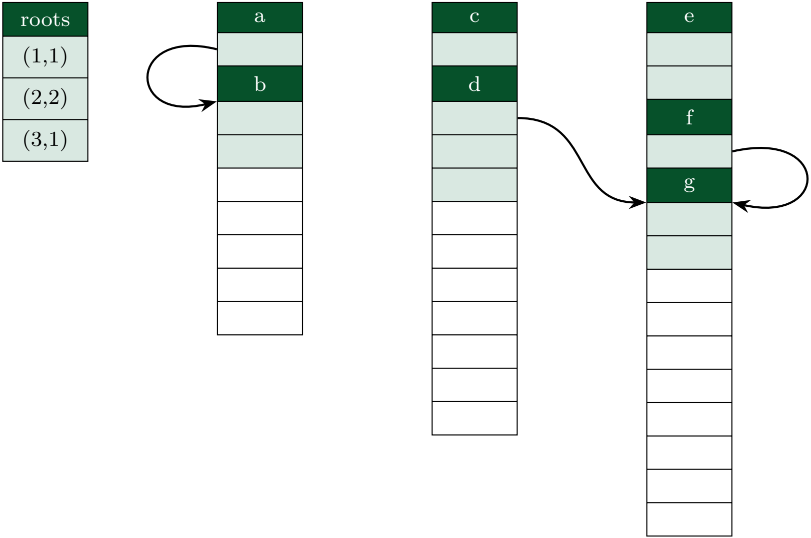
A Generational Garbage Collector

- 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

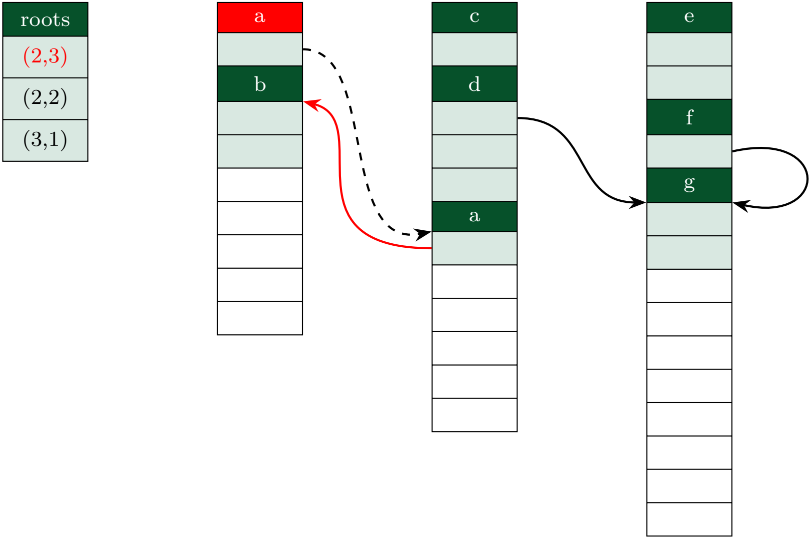
A Generational Garbage Collector

- 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)

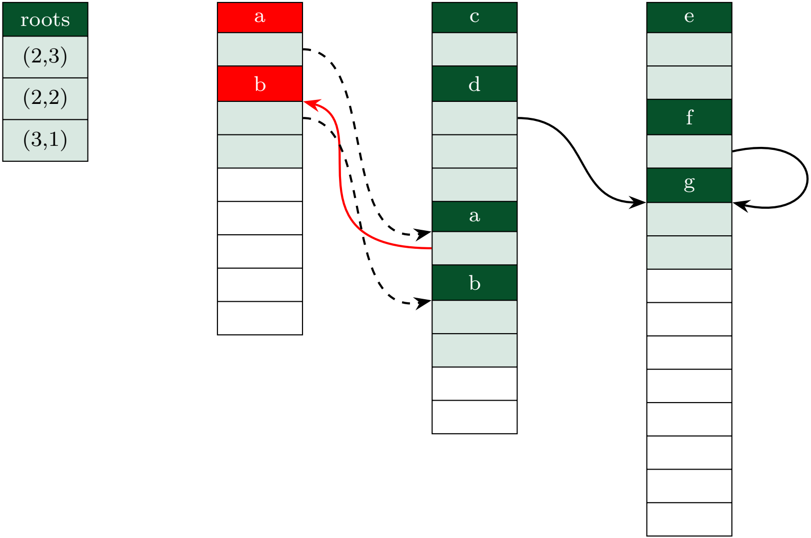
Overview of `forward` and `do_scan`



Overview of `forward` and `do_scan`

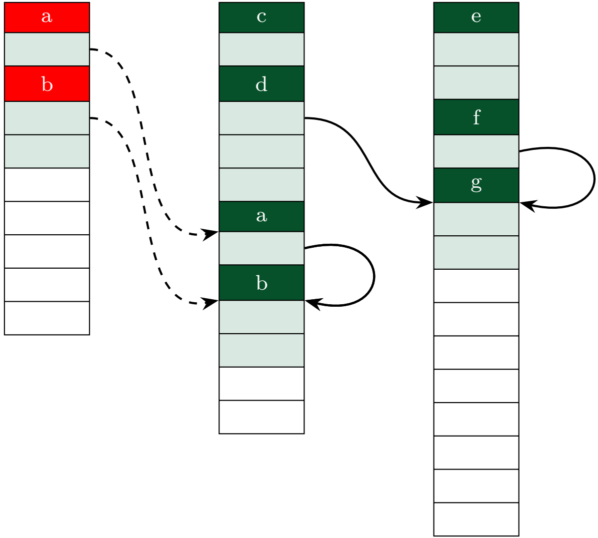


Overview of `forward` and `do_scan`

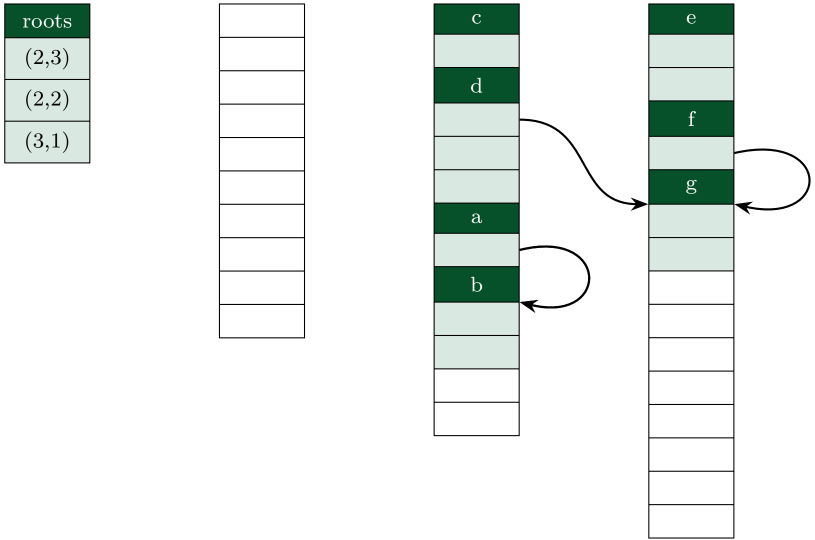


Overview of `forward` and `do_scan`

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



Overview of `forward` and `do_scan`



Bugs in the source C code

- Cheney was executed too conservatively, only part of `to` needs to be scaned.

Bugs in the source C code

- Cheney was executed too conservatively,
  only part of `to` needs to be scaned.
- Overflow in the following calculation:

```
int space_size =
        h->spaces[i].limit - h->spaces[i].start;
```

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.

Separation between pure and spatial reasoning