Mechanized Verification for Graph Algorithms

Shengyi Wang, Qinxiang Cao, **Aquinas Hobor**

Our goals

- Verify graph-manipulating programs
- All proofs mechanized
- Real code
- Techniques able to handle sizable examples

Graph-manipulating programs

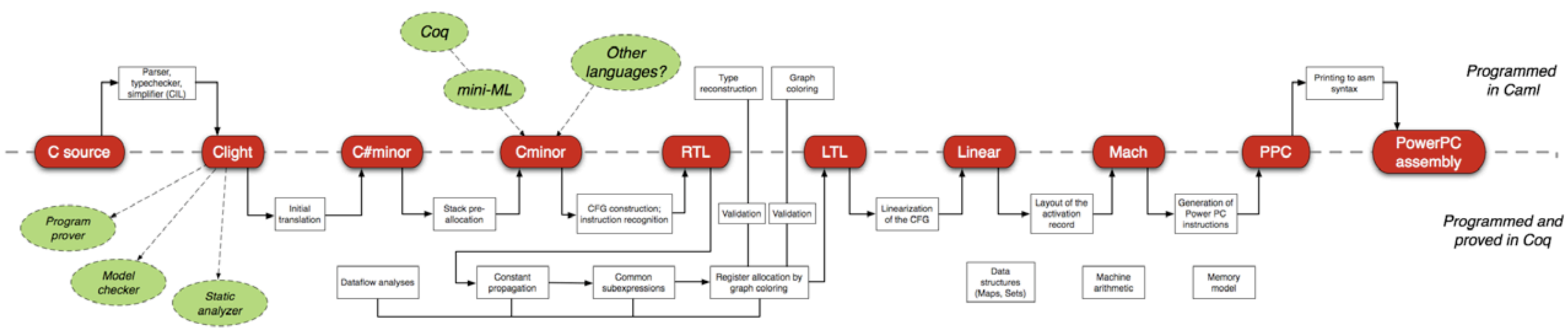
- Heap represented graphs
  - Traditional challenge for verification

- Nontrivial algorithms with “real” specs
  - spanning tree
  - deep copy
  - union-find
  - sizable (~400-line) generational optimized garbage collector for certified compiler (in progress)

Mechanized proofs for real code

- All verification done in Coq

- Target language: CompCert Clight



- Hook into Verified Software Toolchain

Verified Software Toolchain

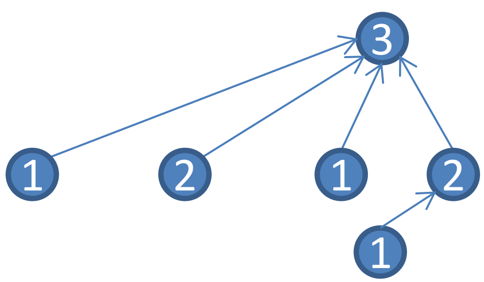
Challenges

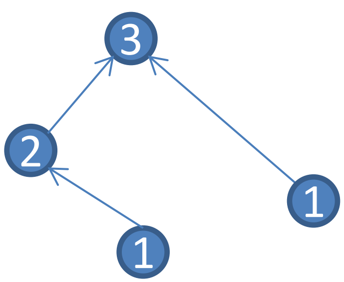
- Separation logic is a little tricky for graph-manipulating structures

- Real code is harder than toy code, sometimes in rather unexpected ways, e.g.
  - Garbage collectors break CompCert's memory model (and type system) due to the typical uniform treatment of data and pointers

Challenges

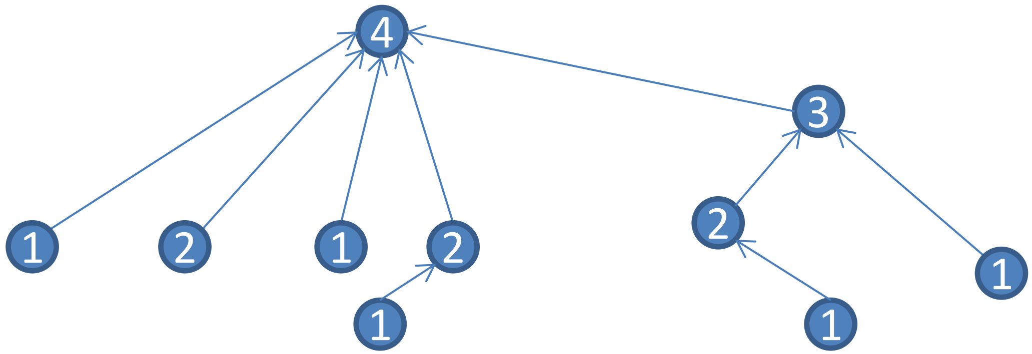
- Graph algorithms are easier to specify relationally rather than functionally
  - No "issues" with termination (esp. in Coq)
  - Some algorithms' "natural specifications" involve nondeterminism (e.g. union-find)
  - Some algorithms do not have easy/natural purely functional implementations (e.g. union-find)





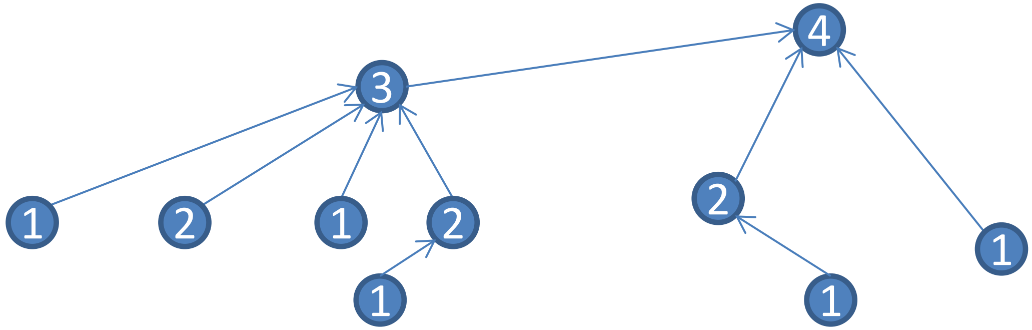
Challenges

- Graph algorithms are easier to specify relationally rather than functionally
  - No “issues” with termination (esp. in Coq)
  - Some algorithms’ “natural specifications” involve nondeterminism (e.g. union-find)
  - Some algorithms do not have easy/natural purely functional implementations (e.g. union-find)



Challenges

- Graph algorithms are easier to specify relationally rather than functionally
  - No "issues" with termination (esp. in Coq)
  - Some algorithms' "natural specifications" involve nondeterminism (e.g. union-find)
  - Some algorithms do not have easy/natural purely functional implementations (e.g. union-find)



Challenges

- Formal graph reasoning is surprisingly subtle, we'd like to reuse definitions, proofs, etc.
  - Reachability
  - Labels
  - Validity
  - Subgraphs

- We'd like generic graphs, and they should be general enough to handle real algorithms.

Some solutions

- Separation logic upgrades: “localization blocks”

```
// {graph(x, γ') ∧ γ(x) = (0, l, r) ∧ mark1(γ, x, γ')}
// ↘ {graph(l, γ')}
ϟ(8)    mark(l);
// ↙ {∃γ''. graph(l, γ'') ∧ mark(γ', l, γ'')}
// { ∃γ''. graph(x, γ'') ∧ γ(x) = (0, l, r) ∧
//   mark1(γ, x, γ') ∧ mark(γ', l, γ'') }
```

Localization is (upgraded) Ramification

Ramify-PQ (Program variables and Quantifiers)

$$\frac{\{L\}\; c\; \{\exists x.\, L_2\} \qquad G_1 \vdash L_1 \star [\![c]\!]\big(\forall x.\, (L_2 \mathrel{-\!\!\star} G_2)\big)}{\{G_1\}\; c\; \{\exists x.\, G_2\}}$$

// $\swarrow$ $\{\exists \gamma''.\ \mathsf{graph}(\mathtt{l}, \gamma'') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'')\}$

// $\left\{\begin{array}{l} \exists \gamma''.\ \mathsf{graph}(\mathtt{x}, \gamma'') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'') \end{array}\right\}$

Localization is (upgraded) Ramification

RAMIFY-PQ (PROGRAM VARIABLES AND QUANTIFIERS)

$$\frac{\{L\}\ c\ \{\exists x.\ L_2\} \qquad G_1 \vdash L_1 \star [\![c]\!]\big(\forall x.\ (L_2 \mathrel{-\!\!\star} G_2)\big)}{\{G_1\}\ c\ \{\exists x.\ G_2\}}$$

SOLVE RAMIFY-PQ

$$\frac{G_1 \vdash L_1 \star F \qquad F \vdash \forall x.\ (L_2 \mathrel{-\!\!\star} G_2)}{G_1 \vdash L_1 \star [\![c]\!]\big(\forall x.\ (L_2 \mathrel{-\!\!\star} G_2)\big)} \quad F \text{ IGNORES } \mathsf{ModVar}(c)$$

A little jig for modified variables

// $\{\mathsf{graph}(\mathtt{x},\gamma) \land \gamma(\mathtt{x}) = (0,l,r)\}$
// $\searrow \{\mathtt{x} \mapsto 0, -, l, r \land \gamma(\mathtt{x}) = (0,l,r)\}$
```
    l = x -> l;
```
$\lightning$(7)
```
    r = x -> r;
    x -> m = 1;
```
// $\swarrow \{\mathtt{x} \mapsto 1, -, \mathtt{l}, \mathtt{r} \land \gamma(\mathtt{x}) = (0,\mathtt{l},\mathtt{r}) \land \exists \gamma'.\ \mathit{mark1}(\gamma,\mathtt{x},\gamma')\}$
// $\{\exists \gamma'.\ \mathsf{graph}(\mathtt{x},\gamma') \land \gamma(\mathtt{x}) = (0,\mathtt{l},\mathtt{r}) \land \mathit{mark1}(\gamma,\mathtt{x},\gamma')\}$

- Uh oh… `l`, `r`, and `x` **are** modified in the localization block…

A little jig for modified variables

$$// \quad \{L_2\}$$

$$// \quad \swarrow \{\exists x, y.\ x = \mathrm{x} \land y = \mathrm{y} \land [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y]L_2\}$$

$$// \quad \{\exists x, y.\ x = \mathrm{x} \land y = \mathrm{y} \land [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y]G_2\}$$

$$// \quad \{G_2\}$$

A little jig for modified variables

// $\{L_2\}$

// $\swarrow \{\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y] L_2\}$

// $\{\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y] G_2\}$

// $\{G_2\}$

$$F \triangleq \forall x, y.\ [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y](L_2 \mathrel{-\!\!\ast} G_2)$$

A little jig for modified variables

$$
\begin{array}{ll}
// & \quad \{L_2\} \\
// & \swarrow \{\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y] L_2\} \\
// & \{\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y] G_2\} \\
// & \{G_2\}
\end{array}
$$

$$F \triangleq \forall x, y.\, [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y](L_2 \mathrel{-\!\!*} G_2)$$

$$G_1 \vdash L_1 \star F \ \mid \ A \vdash B \star \forall x, y.\, [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y](L_2 \mathrel{-\!\!*} G_2)$$

A little jig for modified variables

$$// \quad \{L_2\}$$

$$// \quad \swarrow \{\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y]L_2\}$$

$$// \quad \{\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y]G_2\}$$

$$// \quad \{G_2\}$$

$$F \triangleq \forall x, y.\ [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y](L_2 \mathrel{-\!\!*} G_2)$$

$$G_1 \vdash L_1 \star F \;\Big|\; A \vdash B \star \forall x, y.\ [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y](L_2 \mathrel{-\!\!*} G_2)$$

$$\begin{array}{l|l} F \vdash & (\forall x, y.\ [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y](L_2 \mathrel{-\!\!*} G_2)) \vdash \\ (L'_2 \mathrel{-\!\!*} & \quad (\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y]L_2) \mathrel{-\!\!*} \\ G'_2) & \quad\quad (\exists x, y.\ x = \mathrm{x} \wedge y = \mathrm{y} \wedge [\mathrm{x} \mapsto x][\mathrm{y} \mapsto y]G_2) \end{array}$$

Some other solutions: a sound "graph" predicate in SL

$$\mathsf{graph}(x, \gamma) \;\Leftrightarrow\; x \mapsto \gamma(x) \uplus\!\!\!\!\ast \left(\biguplus\!\!\!\!\ast_{n \in \mathsf{neighbors}(\gamma, x)} \mathsf{graph}(\gamma, n) \right)$$

Some other solutions: a sound "graph" predicate in SL

$$\mathsf{graph}(x,\gamma) \;\Leftrightarrow\; x \mapsto \gamma(x) \circledast \Big(\mathop{\bigcup\!\!\!\!\!\!\ast}_{n \in \mathsf{neighbors}(\gamma, x)} \mathsf{graph}(\gamma, n) \Big)$$

$$\mathsf{graph}(x,\gamma) \;\triangleq\; \mathop{\bigstar}_{v \in reach(\gamma, x)} v \mapsto \gamma(v)$$

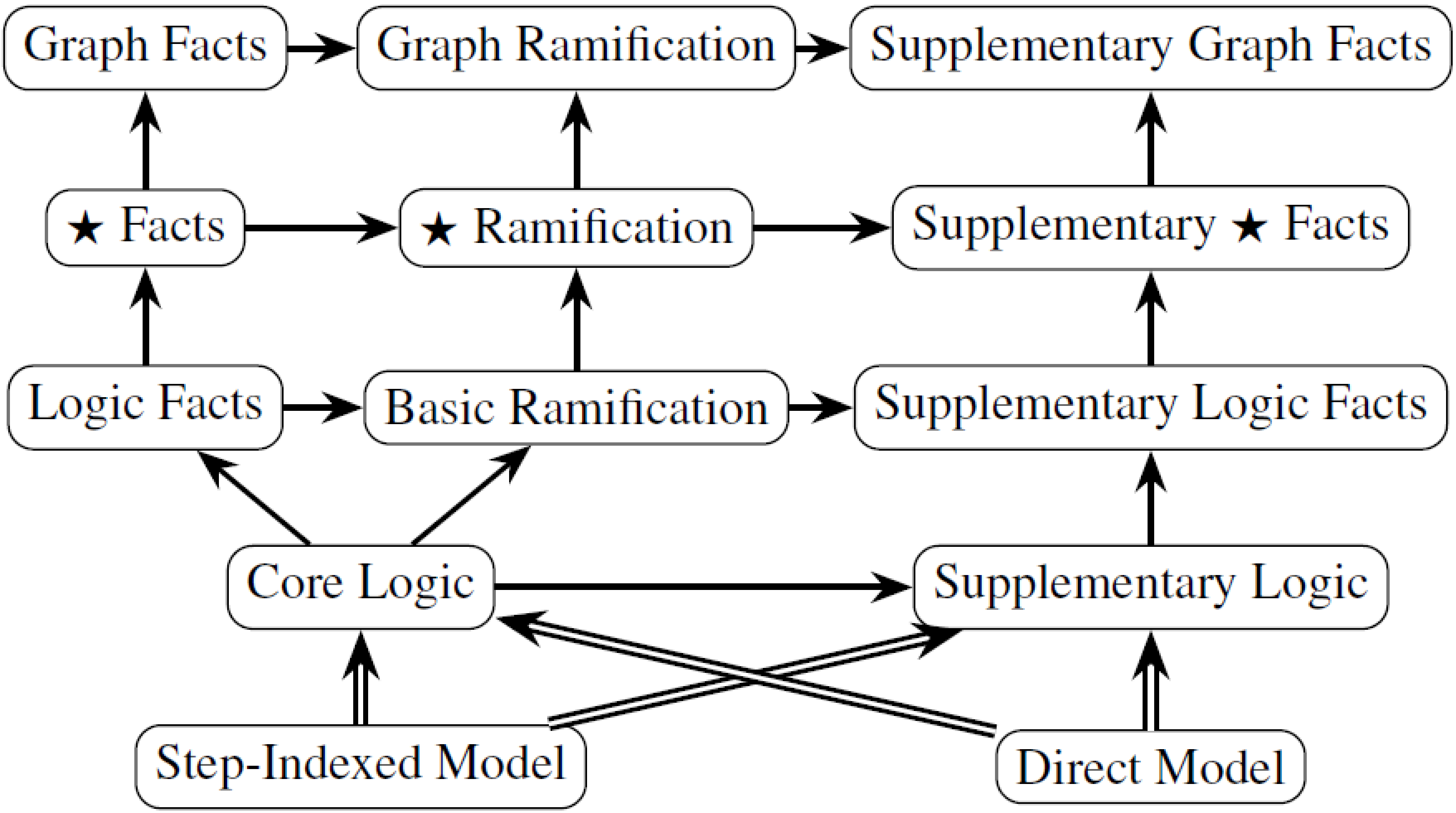
Some other solutions: a sound "graph" predicate in SL

$$\mathsf{graph}(x,\gamma) \;\Leftrightarrow\; x \mapsto \gamma(x) \circledast \Big(\biguplus_{n \in \mathsf{neighbors}(\gamma,x)}^{\star} \mathsf{graph}(\gamma, n) \Big)$$

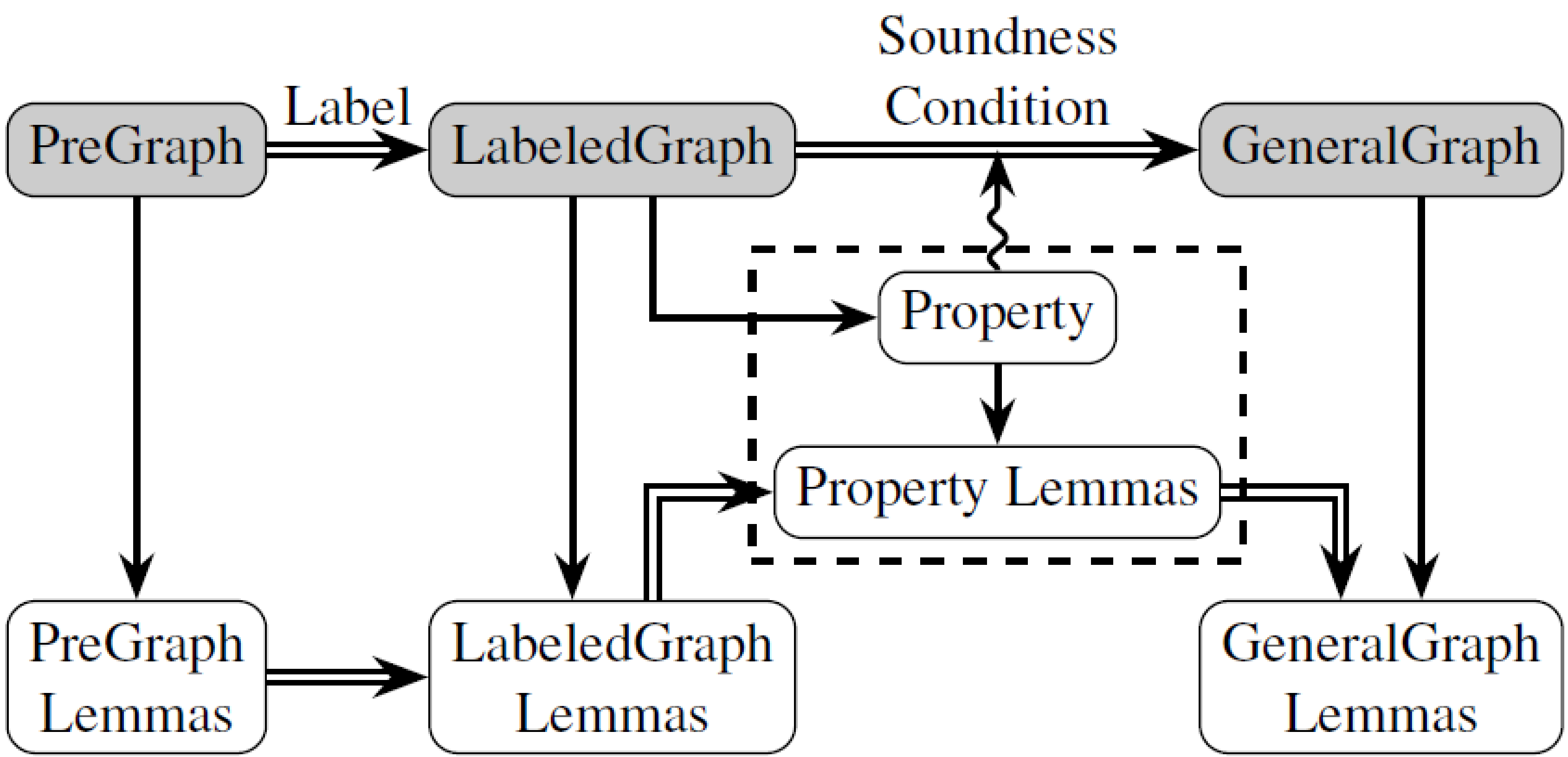
$$\mathsf{graph}(x,\gamma) \;\triangleq\; \bigstar_{v \in reach(\gamma,x)} v \mapsto \gamma(v)$$

$$\forall x, y.\ \Big(P(x) \circledast P(y) \vdash (P(x) \wedge x = y) \vee (P(x) \star P(y)) \Big)$$

Modular
mechanized proof engineering



Some other solutions: A powerful and general graph library



Some other solutions: mechanizing localization blocks in VST

$\{ P_1 \}$	$\{ P_1 \}$	$\{ P_1 \}$	$\{ P_1 \}$
`c1`	`c1`	`c1`	`c1`
$\{ P_2 \}$	$\{ P_2 \}$	$\{ P_2 \}$	$\{ P_2 \}$
↘ $\{ P_3 \}$	$\{ ?F * P_3 \}$	↘ $\{ P_3 \}$	$\{ ?F * P_3 \}$
`c2;`	`c2;`	`c2;`	`c2;`
$\{ P_4 \}$	$\{ ?F * P_4 \}$	$\{ P_4 \}$	$\{ ?F * P_4 \}$
		`c3;`	`c3;`
...	...	↙ $\{ P_5 \}$	$\{ ?F * P_5 \}$
		$\{ P_6 \}$	$\{ P_6 \}$
		...	...

Some future work

- Increase modularity
  - Once you've done one union-find proof, have you done them all?
- Overlaid data structures
  - Common case; can we make them easier?
- Increase confidence of scalability
  - Garbage collector for a "real" client
  - Lots of "undefined operations"
  - Bonus: found a significant performance bug