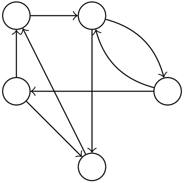
The Ramifications of Sharing in Data Structures

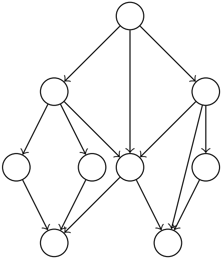
Aquinas Hobor
National University of Singapore

Jules Villard
University College London

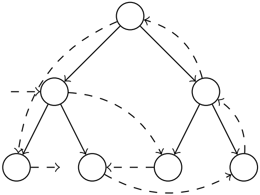
Programs with Sharing in the Wild



Graphs

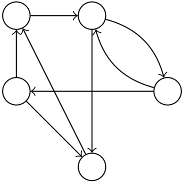


Acyclic graphs (DAGs)

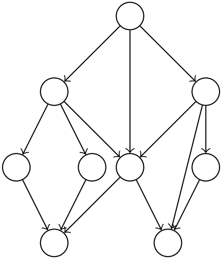


Overlaid data structures
(threaded tree)

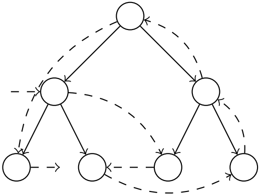
Programs with Sharing in the Wild



Graphs



Acyclic graphs (DAGs)

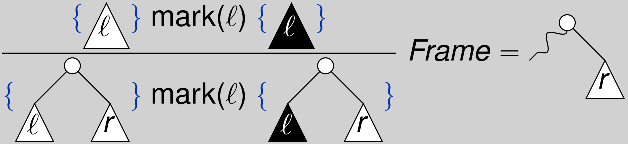


Overlaid data structures (threaded tree)

Verifying Progams with Sharing

- Many techniques applicable (shape analysis, model-checking, . . .)
- Lack of general principles
- Challenge for compositionality

Frame



The AI Frame Problem

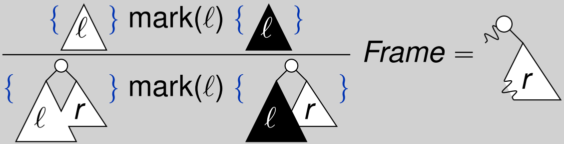
"Describing what does not change as a result of an action"

Success: Separation Logic

- Data structures without sharing (lists, trees, . . .)
- Compositionality based on **disjointness** of memory accesses

Compositional Reasoning for the Heap

Frame



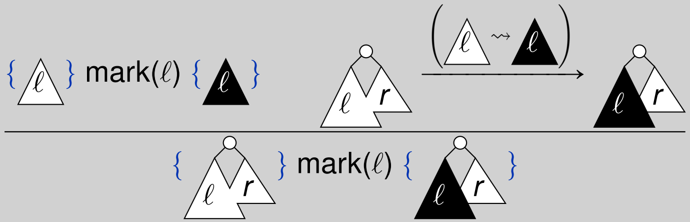
The AI Frame Problem

“Describing what does not change as a result of an action”

Sharing and Frame

- Brittle predicates that shoehorn separation in ($*$)
- Ad-hoc reasoning
- No general solution

Ramification



The AI Ramification Problem

"The ramification problem is concerned with indirect consequences of an action."

Key Points

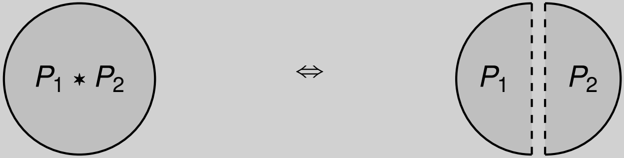
- Embrace sharing when it is natural ($*$, $\circledast$, $\wedge$, ...)
- Separate spatial and mathematical reasoning

Marking a Dag

Tree Logical Predicate [Rey LICS'02]

Separating Conjunction

- $\sigma_1 \bullet \sigma_2$ is the **disjoint union** of σ_1 and σ_2
- $\sigma \models P_1 \star P_2$ iff $\exists \sigma_1, \sigma_2.\, \sigma = \sigma_1 \bullet \sigma_2 \;\&\; \sigma_1 \models P_1 \;\&\; \sigma_2 \models P_2$



Tree Logical Predicate [Rey LICS'02]

Separating Conjunction

- $\sigma_1 \bullet \sigma_2$ is the **disjoint union** of σ_1 and σ_2
- $\sigma \models P_1 \star P_2$ iff $\exists \sigma_1, \sigma_2 . \sigma = \sigma_1 \bullet \sigma_2 \ \& \ \sigma_1 \models P_1 \ \& \ \sigma_2 \models P_2$

Heap Assertions

- emp — empty heap
- $x \mapsto a$ — only x is allocated (`*x == a`)
- $x \mapsto a, b, c$ — $x + 0 \mapsto a \star x + 1 \mapsto b \star x + 2 \mapsto c$

Tree Logical Predicate [Rey LICS'02]

Separating Conjunction

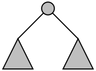
- $\sigma_1 \bullet \sigma_2$ is the **disjoint union** of σ_1 and σ_2
- $\sigma \models P_1 \star P_2$ iff $\exists \sigma_1, \sigma_2 .\, \sigma = \sigma_1 \bullet \sigma_2 \;\&\; \sigma_1 \models P_1 \;\&\; \sigma_2 \models P_2$

- Mathematical trees (terms)

$$\tau \stackrel{\text{def}}{=} E \mid N(v, \tau, \tau)$$

- Spatial trees

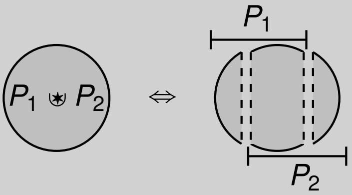
$$\begin{aligned} \text{tree}(x, \tau) \stackrel{\text{def}}{=} & \; (x = 0 \wedge \tau = E \wedge \text{emp}) \qquad \varnothing \\ \vee & \; (\exists \ell, r, v, \tau_\ell, \tau_r .\, \tau = N(v, \tau_\ell, \tau_r) \\ & \;\; \wedge x \mapsto v, \ell, r \star \text{tree}(\ell, \tau_\ell) \star \text{tree}(r, \tau_r)) \end{aligned}$$



Dag Logical Predicate [Rey Unpub'03]

Overlapping Conjunction

- $\sigma \models P_1 \circledast P_2$ iff
  $\exists \sigma_1, \sigma_2, \sigma_3 . \sigma = \sigma_1 \bullet \sigma_2 \bullet \sigma_3 \;\&\; \sigma_1 \bullet \sigma_2 \models P_1 \;\&\; \sigma_2 \bullet \sigma_3 \models P_2$



Dag Logical Predicate [Rey Unpub'03]

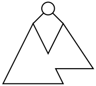
Overlapping Conjunction

- $\sigma \models P_1 \circledast P_2$ iff
  $\exists \sigma_1, \sigma_2, \sigma_3 .\, \sigma = \sigma_1 \bullet \sigma_2 \bullet \sigma_3 \;\&\; \sigma_1 \bullet \sigma_2 \models P_1 \;\&\; \sigma_2 \bullet \sigma_3 \models P_2$

- Mathematical graphs $\gamma \stackrel{\text{def}}{=} (V, L, E)$:
  - V = vertices
  - $L : V \to Val$ = labelling
  - E = edges
- Spatial dags

$$\begin{aligned}\mathsf{dag}(x, \gamma) \stackrel{\text{def}}{=} & \ (x = 0 \wedge \mathsf{emp}) \\ & \vee \exists \ell, r, v.\, \gamma(x) = (v, \ell, r) \wedge \\ & \quad x \mapsto v, \ell, r \star (\mathsf{dag}(\ell, \gamma) \circledast \mathsf{dag}(r, \gamma))\end{aligned}$$

$\emptyset$



Dag Logical Predicate [Rey Unpub'03]

Overlapping Conjunction

- $\sigma \models P_1 \circledast P_2$ iff
  $\exists \sigma_1, \sigma_2, \sigma_3.\, \sigma = \sigma_1 \bullet \sigma_2 \bullet \sigma_3 \;\&\; \sigma_1 \bullet \sigma_2 \models P_1 \;\&\; \sigma_2 \bullet \sigma_3 \models P_2$

- Mathematical graphs $\gamma \stackrel{\text{def}}{=} (V, L, E)$:
  - V = vertices
  - $L : V \to Val$ = labelling
  - E = edges
- Spatial graphs

$$\begin{aligned} \mathsf{graph}(x, \gamma) \stackrel{\text{def}}{=} & (x = 0 \wedge \mathsf{emp}) \\ & \vee \exists \ell, r, v.\, \gamma(x) = (v, \ell, r) \wedge \\ & \quad x \mapsto v, \ell, r \circledast \mathsf{graph}(\ell, \gamma) \circledast \mathsf{graph}(r, \gamma) \end{aligned}$$

Overlapping conjunction $\circledast$

Folklore

- $\circledast$ comes from relevance logic ("relevant conjunction", "sepish", ...)
- $\mathsf{dag}(x, \gamma)$ known in folklore [Rey Unpub'03]

The Frame Rule

$$\frac{\{P\}\ c\ \{Q\}}{\{P \ast F\}\ c\ \{Q \ast F\}}$$

$$\frac{\{\mathsf{tree}(\ell, \tau_\ell)\}\ \mathsf{mark}(\ell)\ \{\mathsf{tree}(\ell, \tau'_\ell)\}}{\{\mathsf{tree}(\ell, \tau_\ell) \ast \mathsf{tree}(r, \tau_r)\}\ \mathsf{mark}(\ell)\ \{\mathsf{tree}(\ell, \tau'_\ell) \ast \mathsf{tree}(r, \tau_r)\}}$$

Challenge

How to use $\circledast$ for verification?

```
struct node {short m; struct node *l,*r;};

void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
  mark_dag(l);
  mark_dag(r);
  d->m = 1;
} // {dag(d, m(γ, d))}
```

Mathematical Marking $m(\gamma, x)$

$$m((V, L, E), x) = (V, L', E)$$

where

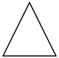
$$L'(y) = \begin{cases} 1 & \text{if } y \in reach\_zero(\gamma, x) \\ L(y) & \text{otherwise} \end{cases}$$

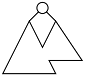
Proof of `mark_dag` Using Framing

```
struct node {short m; struct node *l,*r;};

void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
//
//
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```

Proof of mark_dag Using Framing

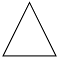
```
struct node {short m; struct node *l,*r;};

void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r * (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
//
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```

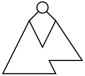




Proof of `mark_dag` Using Framing

```
struct node {short m; struct node *l,*r;};

void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
//
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```



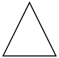


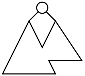
The Frame Rule

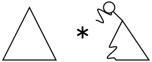
$$\frac{\{P\}\ c\ \{Q\}}{\{P * F\}\ c\ \{Q * F\}}$$

Proof of `mark_dag` Using Framing

```
struct node {short m; struct node *l,*r;};

void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//    ∧ γ(d) = (0, l, r) }
// {dag(l, γ) ∗ ???}
  mark_dag(l);
// {dag(l, m(γ, l)) ∗ ???}
  mark_dag(r);
//

  d->m = 1;
//
} // {dag(d, m(γ, d))}
```







The Frame Rule

$$\frac{\{P\}\ c\ \{Q\}}{\{P \ast F\}\ c\ \{Q \ast F\}}$$

Tailoring Dags to Frame [BOC SPACE'04]

- **Separated** mathematical dag (term)

$$\delta \stackrel{\text{def}}{=} \mathit{Empty} \mid x : \mathit{Node}\ \delta_\ell\ \delta_r \mid \mathit{Ptr}\ x$$

- Spatial **separated** dags

$$\begin{aligned}
\mathsf{pdag}(0, \mathit{Empty}) &\stackrel{\text{def}}{=} \mathsf{emp} \\
\mathsf{pdag}(x, x{:}\mathit{Node}\ v\ \delta_\ell\ \delta_r) &\stackrel{\text{def}}{=} \exists \ell, r.\, x \mapsto v, \ell, r \ast \mathsf{pdag}(\ell, \delta_\ell) \ast \mathsf{pdag}(r, \delta_r) \\
\mathsf{pdag}(x, \mathit{Ptr}\ x) &\stackrel{\text{def}}{=} \mathsf{emp}
\end{aligned}$$

Caveats

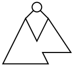
- Predicate depends on the order of traversal
- Complex specifications and invariants

Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
//
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```

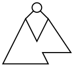
Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ⋆ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r)                    }
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```



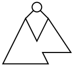
Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```



Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
//  { d ↦ 0, l, r ⋆ (dag(l, γ) ⊛ dag(r, γ))
//    ∧ γ(d) = (0, l, r)                     }
  mark_dag(l);
//
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```



The Ramify Rule of Separation Logic

The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\}}{\{R\}\ c\ \{R'\}}$$

The Ramify Rule of Separation Logic

The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad \mathit{ramify}(P \leadsto Q, R) = R'}{\{R\}\ c\ \{R'\}}$$

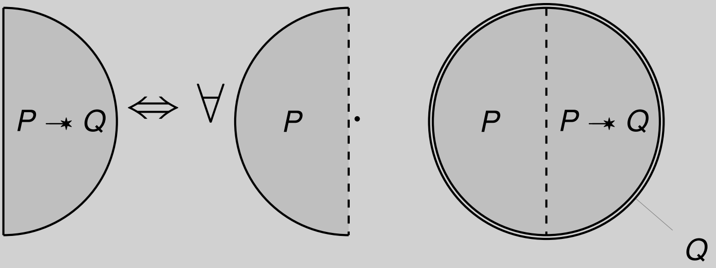
The Ramify Rule of Separation Logic

The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P * (Q -\!\!* R')}{\{R\}\ c\ \{R'\}}$$

Magic Wand

- $\sigma \models P -\!\!* Q$ iff $\forall \sigma' \models P.\ \sigma \bullet \sigma' \models Q$



The Ramify Rule of Separation Logic

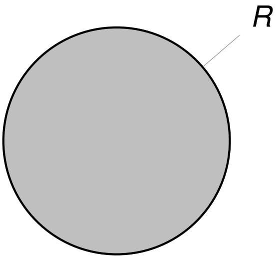
The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P * (Q -\!\!* R')}{\{R\}\ c\ \{R'\}}$$

Magic Wand

- $\sigma \models P -\!\!* Q$ iff $\forall \sigma' \models P. \sigma \bullet \sigma' \models Q$

Ramification Entailment



The Ramify Rule of Separation Logic

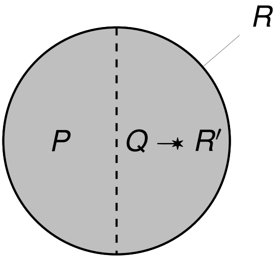
The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P * (Q -\!\!* R')}{\{R\}\ c\ \{R'\}}$$

Magic Wand

- $\sigma \models P -\!\!* Q$ iff $\forall \sigma' \models P.\ \sigma \bullet \sigma' \models Q$

Ramification Entailment



The Ramify Rule of Separation Logic

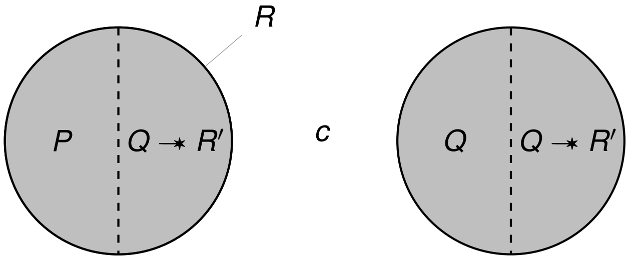
The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P * (Q \mathrel{-\!\!*} R')}{\{R\}\ c\ \{R'\}}$$

Magic Wand

- $\sigma \models P \mathrel{-\!\!*} Q$ iff $\forall \sigma' \models P.\ \sigma \bullet \sigma' \models Q$

Ramification Entailment



The Ramify Rule of Separation Logic

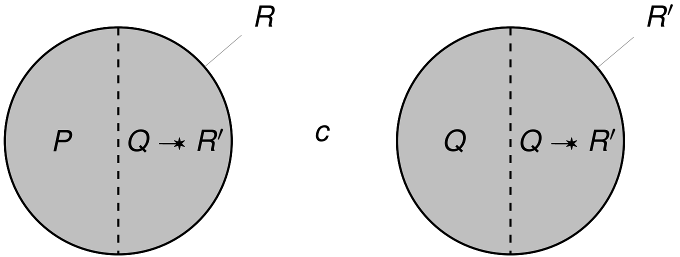
The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P * (Q \mathrel{-\!\!*} R')}{\{R\}\ c\ \{R'\}}$$

Magic Wand

- $\sigma \models P \mathrel{-\!\!*} Q$ iff $\forall \sigma' \models P. \sigma \bullet \sigma' \models Q$

Ramification Entailment



Back to `mark_dag`: First Recursive Call

The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P \star (Q \mathrel{-\!\!\star} R')}{\{R\}\ c\ \{R'\}}$$

// $\{\mathsf{dag}(\mathtt{l}, \gamma) \circledast \mathsf{dag}(\mathtt{r}, \gamma)\}$

```
  mark_dag(l);
```

// ↯

Back to `mark_dag`: First Recursive Call

The Ramify Rule

$$\frac{\{P\}\ c\ \{Q\} \qquad R \vdash P \star (Q \mathrel{-\!\!\star} R')}{\{R\}\ c\ \{R'\}}$$

```
//  {dag(l, γ) ⊛ dag(r, γ)}
  mark_dag(l);
//↯{dag(l, m(γ, l)) ⊛ dag(r, m(γ, l))}
```

Ramification Condition

$$\begin{array}{ll} & \mathsf{dag}(\ell, \gamma) \circledast \mathsf{dag}(r, \gamma) \\ \vdash & \mathsf{dag}(\ell, \gamma) \star (\mathsf{dag}(\ell, m(\gamma, \ell)) \mathrel{-\!\!\star} \\ & \quad \mathsf{dag}(\ell, m(\gamma, \ell)) \circledast \mathsf{dag}(r, m(\gamma, \ell))) \end{array}$$

Ramification Library: Updating DAGs

Lemma — SubDAG Update

$$\frac{\mathit{reach}(\gamma', x) \supseteq \mathit{reach}(\gamma, x) \qquad \mathit{unreach}(\gamma', x) = \mathit{unreach}(\gamma, x)}{\mathsf{dag}(x,\gamma) \circledast \mathsf{dag}(y,\gamma) \vdash \mathsf{dag}(x,\gamma) * (\mathsf{dag}(x,\gamma') \mathrel{-\!\!*} \mathsf{dag}(x,\gamma') \circledast \mathsf{dag}(y,\gamma'))}$$

- $\mathit{reach}(\gamma', x) \supseteq \mathit{reach}(\gamma, x)$: no deallocation
- $\mathit{unreach}(\gamma', x) = \mathit{unreach}(\gamma, x)$: local modification

Lemma — **SubDAG Update**

$$\frac{\mathit{reach}(\gamma', x) \supseteq \mathit{reach}(\gamma, x) \qquad \mathit{unreach}(\gamma', x) = \mathit{unreach}(\gamma, x)}{\mathsf{dag}(x,\gamma) \circledast \mathsf{dag}(y,\gamma) \vdash \mathsf{dag}(x,\gamma) * (\mathsf{dag}(x,\gamma') \mathrel{-\!\!*} \mathsf{dag}(x,\gamma') \circledast \mathsf{dag}(y,\gamma'))}$$

First Recursive Call

```
//  {dag(l, γ) ⊛ dag(r, γ)}
  mark_dag(l);
//ϟ{dag(l, m(γ, l)) ⊛ dag(r, m(γ, l))}
```

$$\mathsf{dag}(\ell,\gamma) \circledast \mathsf{dag}(r,\gamma) \vdash \mathsf{dag}(\ell,\gamma) * (\mathsf{dag}(\ell, m(\gamma,\ell)) \mathrel{-\!\!*} \mathsf{dag}(\ell, m(\gamma,\ell)) \circledast \mathsf{dag}(r, m(\gamma,\ell)))$$

Ramification Library: Updating DAGs

Lemma — **SubDAG Update**

$$\frac{\mathit{reach}(\gamma', x) \supseteq \mathit{reach}(\gamma, x) \qquad \mathit{unreach}(\gamma', x) = \mathit{unreach}(\gamma, x)}{\mathsf{dag}(x,\gamma) \circledast \mathsf{dag}(y,\gamma) \vdash \mathsf{dag}(x,\gamma) * (\mathsf{dag}(x,\gamma') \mathrel{-\!\!*} \mathsf{dag}(x,\gamma') \circledast \mathsf{dag}(y,\gamma'))}$$

First Recursive Call

```
// {dag(l, γ) ⊛ dag(r, γ)}
   mark_dag(l);
// ↯ {dag(l, m(γ, l)) ⊛ dag(r, m(γ, l))}
```

$$\frac{\overline{\mathit{reach}(m(\gamma,\ell),\ell) \supseteq \mathit{reach}(\gamma,\ell)} \qquad \overline{\mathit{unr.}(m(\gamma,\ell),\ell) = \mathit{unr.}(\gamma,\ell)}}{\mathsf{dag}(\ell,\gamma) \circledast \mathsf{dag}(r,\gamma) \vdash \mathsf{dag}(\ell,\gamma) * (\mathsf{dag}(\ell, m(\gamma,\ell)) \mathrel{-\!\!*} \mathsf{dag}(\ell, m(\gamma,\ell)) \circledast \mathsf{dag}(r, m(\gamma,\ell)))}$$

Ramification Library: Updating DAGs

Lemma — **SubDAG Update**

$$\frac{reach(\gamma', x) \supseteq reach(\gamma, x) \qquad unreach(\gamma', x) = unreach(\gamma, x)}{\mathsf{dag}(x,\gamma) \circledast \mathsf{dag}(y,\gamma) \vdash \mathsf{dag}(x,\gamma) * (\mathsf{dag}(x,\gamma') \mathrel{-\!\!*} \mathsf{dag}(x,\gamma') \circledast \mathsf{dag}(y,\gamma'))}$$

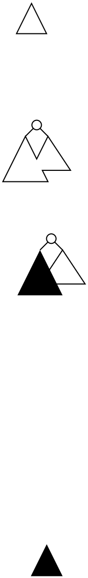
First Recursive Call

```
//  {dag(l, γ) ⊛ dag(r, γ)}
  mark_dag(l);
//↯{dag(l, m(γ, l)) ⊛ dag(r, m(γ, l))}
```

$$\frac{\dfrac{\textbf{math}}{reach(m(\gamma,\ell),\ell) \supseteq reach(\gamma,\ell)} \qquad \dfrac{\textbf{math}}{unr.(m(\gamma,\ell),\ell) = unr.(\gamma,\ell)}}{\mathsf{dag}(\ell,\gamma) \circledast \mathsf{dag}(r,\gamma) \vdash \mathsf{dag}(\ell,\gamma) * (\mathsf{dag}(\ell, m(\gamma,\ell)) \mathrel{-\!\!*} \mathsf{dag}(\ell, m(\gamma,\ell)) \circledast \mathsf{dag}(r, m(\gamma,\ell)))}$$

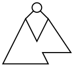
Proof of `mark_dag` using Ramification

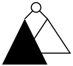
```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
//  { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//    ∧ γ(d) = (0, l, r) }
  mark_dag(l);
//↯ { d ↦ 0, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//    ∧ γ(d) = (0, l, r) }
  mark_dag(r);

//

  d->m = 1;

//

} // {dag(d, m(γ, d))}
```



Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//    ∧ γ(d) = (0, l, r) }
  mark_dag(l);
//↯ { d ↦ 0, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//    ∧ γ(d) = (0, l, r) }
  mark_dag(r);

//

  d->m = 1;

//

} // {dag(d, m(γ, d))}
```



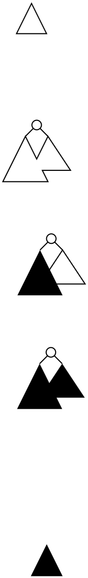


Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//    ∧ γ(d) = (0, l, r) }
  mark_dag(l);
// ↯ { d ↦ 0, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//    ∧ γ(d) = (0, l, r) }
  mark_dag(r);
//
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```

Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(l);
// ↯ { d ↦ 0, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(r);
// ↯ { d ↦ 0, l, r ∗ (dag(l, γ′) ⊛ dag(r, γ′))
//   ∧ γ(d) = (0, l, r) ∧ γ′ = m(m(γ, l), r) }
  d->m = 1;

//

} // {dag(d, m(γ, d))}
```



Second Recursive Call

$$
\begin{array}{ll}
 & \mathsf{dag}(\ell, m(\gamma,\ell)) \circledast \mathsf{dag}(r, m(\gamma,\ell)) \\
\vdash & \mathsf{dag}(r, m(\gamma,\ell)) \ast (\mathsf{dag}(r, m(m(\gamma,\ell), r)) \mathrel{-\!\!\ast} \\
 & \quad \mathsf{dag}(\ell, m(m(\gamma,\ell), r)) \circledast \mathsf{dag}(r, m(m(\gamma,\ell), r)))
\end{array}
$$

Ramification Conditions

Second Recursive Call

$$
\begin{aligned}
& \mathsf{dag}(\ell, m(\gamma,\ell)) \circledast \mathsf{dag}(r, m(\gamma,\ell)) \\
\vdash\ & \mathsf{dag}(r, m(\gamma,\ell)) * (\mathsf{dag}(r, m(m(\gamma,\ell), r)) \mathrel{-\!\!*} \\
& \qquad \mathsf{dag}(\ell, m(m(\gamma,\ell), r)) \circledast \mathsf{dag}(r, m(m(\gamma,\ell), r)))
\end{aligned}
$$

$\Uparrow$

$\ell \leftrightarrow r$

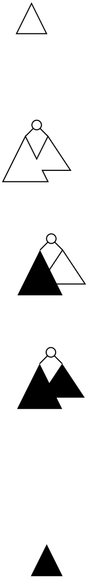
$\gamma \leftarrow m(\gamma, \ell)$

First Recursive Call

$$
\begin{aligned}
& \mathsf{dag}(\ell, \gamma) \circledast \mathsf{dag}(r, \gamma) \\
\vdash\ & \mathsf{dag}(\ell, \gamma) * (\mathsf{dag}(\ell, m(\gamma,\ell)) \mathrel{-\!\!*} \\
& \qquad \mathsf{dag}(\ell, m(\gamma,\ell)) \circledast \mathsf{dag}(r, m(\gamma,\ell)))
\end{aligned}
$$

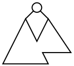
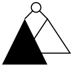
Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(l);
// ↯ { d ↦ 0, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(r);
// ↯ { d ↦ 0, l, r ∗ (dag(l, γ′) ⊛ dag(r, γ′))
//   ∧ γ(d) = (0, l, r) ∧ γ′ = m(m(γ, l), r) }
  d->m = 1;
//
} // {dag(d, m(γ, d))}
```



Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
//   { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//     ∧ γ(d) = (0, l, r) }
  mark_dag(l);
// ϟ { d ↦ 0, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//     ∧ γ(d) = (0, l, r) }
  mark_dag(r);
// ϟ { d ↦ 0, l, r ∗ (dag(l, γ′) ⊛ dag(r, γ′))
//     ∧ γ(d) = (0, l, r) ∧ γ′ = m(m(γ, l), r) }
  d->m = 1;

//

} // {dag(d, m(γ, d))}
```

Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
```

$$// \left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma) \circledast \mathsf{dag}(\mathtt{r}, \gamma)) \\ \wedge\ \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

```
  mark_dag(l);
```

$$// \lightning \left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\gamma, \mathtt{l})) \circledast \mathsf{dag}(\mathtt{r}, m(\gamma, \mathtt{l}))) \\ \wedge\ \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

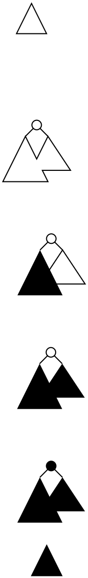
```
  mark_dag(r);
```

$$// \lightning \left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma') \circledast \mathsf{dag}(\mathtt{r}, \gamma')) \\ \wedge\ \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma' = m(m(\gamma, \mathtt{l}), \mathtt{r}) \end{array} \right\}$$

```
  d->m = 1;
```

$$// \left\{ \begin{array}{l} \mathtt{d} \mapsto 1, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma') \circledast \mathsf{dag}(\mathtt{r}, \gamma')) \\ \wedge\ \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma' = m(m(\gamma, \mathtt{l}), \mathtt{r}) \end{array} \right\}$$

```
} // {dag(d, m(γ, d))}
```



Proof of `mark_dag` using Ramification

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
```

$$// \left\{ \begin{array}{l} d \mapsto 0, l, r * (\mathrm{dag}(l, \gamma) \circledast \mathrm{dag}(r, \gamma)) \\ \wedge\ \gamma(d) = (0, l, r) \end{array} \right\}$$

```
  mark_dag(l);
```

$$// \lightning \left\{ \begin{array}{l} d \mapsto 0, l, r * (\mathrm{dag}(l, m(\gamma, l)) \circledast \mathrm{dag}(r, m(\gamma, l))) \\ \wedge\ \gamma(d) = (0, l, r) \end{array} \right\}$$

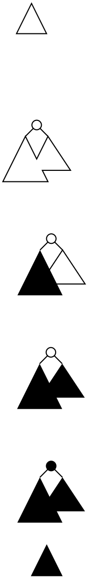
```
  mark_dag(r);
```

$$// \lightning \left\{ \begin{array}{l} d \mapsto 0, l, r * (\mathrm{dag}(l, \gamma') \circledast \mathrm{dag}(r, \gamma')) \\ \wedge\ \gamma(d) = (0, l, r) \wedge \gamma' = m(m(\gamma, l), r) \end{array} \right\}$$

```
  d->m = 1;
```

$$// \left\{ \begin{array}{l} d \mapsto 1, l, r * (\mathrm{dag}(l, \gamma') \circledast \mathrm{dag}(r, \gamma')) \\ \wedge\ \gamma(d) = (0, l, r) \wedge \gamma' = m(m(\gamma, l), r) \end{array} \right\}$$

```
} // {dag(d, m(γ, d))}
```



Establishing the Post-Condition of `mark_dag`

Single Node Marking $m_1(\gamma, x)$

$$m_1((V, L, E), x) = (V, L', E) \text{ where } L'(y) = \begin{cases} 1 & \text{if } y = x \\ L(y) & \text{otherwise} \end{cases}$$

Lemma — Mathematical Marking Facts

$$m(m(m_1(\gamma, x), \ell), r) = m(m_1(m(\gamma, \ell), x), r) =$$
$$m_1(m(m(\gamma, \ell), r), x) = m(m_1(m(\gamma, r), x), \ell) = m(\gamma, x)$$

Post-Condition Entailment

$$\mathtt{d} \mapsto 1, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma') \circledast \mathsf{dag}(\mathtt{r}, \gamma'))$$
$$\land \gamma(d) = (0, \ell, r) \land \gamma' = m(m(\gamma, \ell), r)$$
$$\vdash \mathsf{dag}(\mathtt{d}, m_1(m(m(\gamma, \mathtt{l}), \mathtt{r}), \mathtt{d}))$$
$$\vdash \mathsf{dag}(\mathtt{d}, m(\gamma, \mathtt{d}))$$

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
```

//
$$\left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma) \circledast \mathsf{dag}(\mathtt{r}, \gamma)) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

```
  mark_dag(l);
```

// ↯
$$\left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\gamma, \mathtt{l})) \circledast \mathsf{dag}(\mathtt{r}, m(\gamma, \mathtt{l}))) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

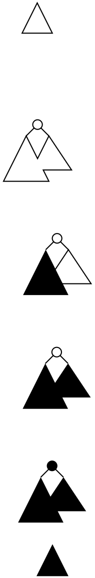
```
  mark_dag(r);
```

// ↯
$$\left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma') \circledast \mathsf{dag}(\mathtt{r}, \gamma')) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma' = m(m(\gamma, \mathtt{l}), \mathtt{r}) \end{array} \right\}$$

```
  d->m = 1;
```

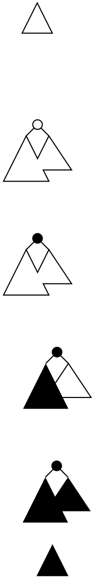
//
$$\left\{ \begin{array}{l} \mathtt{d} \mapsto 1, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma') \circledast \mathsf{dag}(\mathtt{r}, \gamma')) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma' = m(m(\gamma, \mathtt{l}), \mathtt{r}) \end{array} \right\}$$

```
} // {dag(d, m(γ, d))}
```



Robustness of the Proof

```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ 0, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
  d->m = 1;
// { d ↦ 1, l, r ∗ (dag(l, γ) ⊛ dag(r, γ))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(l);
//↯ { d ↦ 1, l, r ∗ (dag(l, m(γ, l)) ⊛ dag(r, m(γ, l)))
//   ∧ γ(d) = (0, l, r) }
  mark_dag(r);
//↯ { d ↦ 1, l, r ∗ (dag(l, γ′) ⊛ dag(r, γ′))
//   ∧ γ(d) = (0, l, r) ∧ γ′ = m(m(γ, l), r) }
} // {dag(d, m(γ, d))}
```



```
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
```

// $$\left\{ \begin{array}{l} \mathtt{d} \mapsto 0, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma) \circledast \mathsf{dag}(\mathtt{r}, \gamma)) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

```
  d->m = 1;
```

// $$\left\{ \begin{array}{l} \mathtt{d} \mapsto 1, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma) \circledast \mathsf{dag}(\mathtt{r}, \gamma)) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

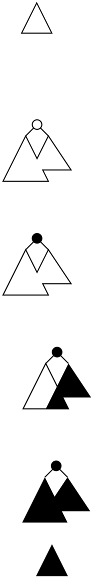
```
  mark_dag(r);
```

// ↯ $$\left\{ \begin{array}{l} \mathtt{d} \mapsto 1, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\gamma, \mathtt{r})) \circledast \mathsf{dag}(\mathtt{r}, m(\gamma, \mathtt{r}))) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \end{array} \right\}$$

```
  mark_dag(l);
```

// ↯ $$\left\{ \begin{array}{l} \mathtt{d} \mapsto 1, \mathtt{l}, \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \gamma') \circledast \mathsf{dag}(\mathtt{r}, \gamma')) \\ \wedge\, \gamma(d) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma' = m(m(\gamma, \mathtt{r}), \mathtt{l}) \end{array} \right\}$$

```
} // {dag(d, m(γ, d))}
```



```
void mark_graph(struct node *g) { // {graph(g, γ)}
  if (!g || g->m) return;
  struct node *l = g->l, *r = g->r;
// { g ↦ 0, l, r ⊛ graph(l, γ) ⊛ graph(r, γ)
//   ∧ γ(g) = (0, l, r) }
  g->m = 1;
// ↯ { g ↦ 1, l, r ⊛ graph(l, γ1) ⊛ graph(r, γ1)
//   ∧ γ(g) = (0, l, r) ∧ γ1 = m1(γ, g) }
  mark_graph(r);
// ↯ { g ↦ 1, l, r ⊛ graph(l, m(γ1, r)) ⊛ graph(r, m(γ1, r))
//   ∧ γ(g) = (0, l, r) ∧ γ1 = m1(γ, g) }
  mark_graph(l);
// ↯ { g ↦ 1, l, r ⊛ graph(l, γ') ⊛ graph(r, γ')
//   ∧ γ(g) = (0, l, r) ∧ γ' = m(m(m1(γ, g), r), l) }
} // {graph(g, m(γ, g))}
```

Marking a Single Node in a Graph

Lemma — Single Graph Node Update

$$\frac{\gamma(x) = (d, \ell, r) \qquad \gamma' = [x \mapsto (d', \ell, r)]\gamma}{\mathsf{graph}(x, \gamma) \vdash x \mapsto d, \ell, r \ast (x \mapsto d', \ell, r \mathrel{-\!\!\ast} \mathsf{graph}(x, \gamma'))}$$

Marking the Root

$$\frac{\gamma(\mathtt{g}) = (0, \mathtt{l}, \mathtt{r}) \qquad \gamma_1 = [x \mapsto (1, \mathtt{l}, \mathtt{r})]\gamma}{\begin{array}{l} \mathtt{g} \mapsto 0, \mathtt{l}, \mathtt{r} \circledast \mathsf{graph}(\mathtt{l}, \gamma) \circledast \mathsf{graph}(\mathtt{r}, \gamma) \\ \vdash \mathtt{g} \mapsto 0, \mathtt{l}, \mathtt{r} \ast (\mathtt{g} \mapsto 1, \mathtt{l}, \mathtt{r} \mathrel{-\!\!\ast} \\ \quad \mathtt{g} \mapsto 1, \mathtt{l}, \mathtt{r} \circledast \mathsf{graph}(\mathtt{l}, \gamma_1) \circledast \mathsf{graph}(\mathtt{r}, \gamma_1) \wedge \gamma_1 = m_1(\gamma, \mathtt{g})) \end{array}}$$

Robustness of the Proof

```
void mark_graph(struct node *g) { // {graph(g, γ)}
  if (!g || g->m) return;
  struct node *l = g->l, *r = g->r;
// { g ↦ 0, l, r ⊛ graph(l, γ) ⊛ graph(r, γ)
//   ∧ γ(g) = (0, l, r) }
  g->m = 1;
//↯ { g ↦ 1, l, r ⊛ graph(l, γ_1) ⊛ graph(r, γ_1)
//    ∧ γ(g) = (0, l, r) ∧ γ_1 = m_1(γ, g) }
  mark_graph(r);
//↯ { g ↦ 1, l, r ⊛ graph(l, m(γ_1, r)) ⊛ graph(r, m(γ_1, r))
//    ∧ γ(g) = (0, l, r) ∧ γ_1 = m_1(γ, g) }
  mark_graph(l);
//↯ { g ↦ 1, l, r ⊛ graph(l, γ') ⊛ graph(r, γ')
//    ∧ γ(g) = (0, l, r) ∧ γ' = m(m(m_1(γ, g), r), l) }
} // {graph(g, m(γ, g))}
```

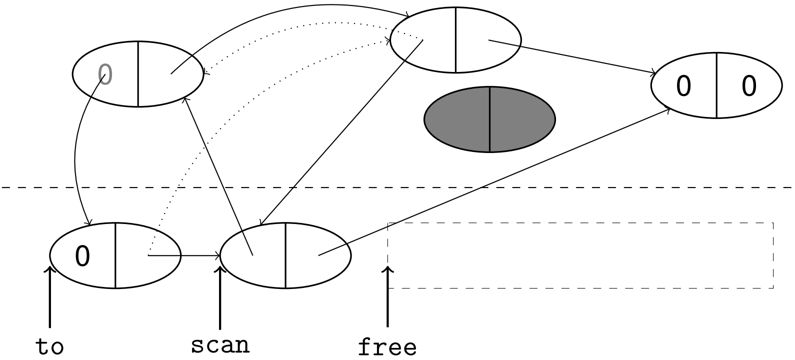
Acid Test: Cheney's GC

Cheney's Copying Garbage Collector

```
void collect(void **r) {
  void *tmp = fromSpace;
  fromSpace = toSpace;
  toSpace = tmp;
  free = toSpace;
  scan = free;
  copy_ref(r);
  while (scan != free) {
    copy((void**)scan);
    copy((void**)(scan+4));
    scan = scan + 8;
  }
}
```

```
void copy(void **p) {
 if (p && *p) {
  void *obj = *p;
  int fwd = *(int*) obj;
  if (fwd &&
      toSpace <= (void*)fwd &&
      (void*)fwd < toSpace+spaceSz){
    *(void**)p = (void*)fwd;
  } else {
    void *newObj = free;
    free = free + 8;
    *(int*)newObj = *(int*)obj;
    *(int*)(newObj + 4) =
           *(int*)(obj + 4);
    *(void**)obj = newObj;
    *(void**)p = newObj;
  }
 }
}
```

State During the Execution



With Framing [BTR POPL'04]

Loop Invariant

$$\mathrm{iso}(\phi, \mathit{FORW}, \mathit{BUSY}) \wedge (\mathit{ALIVE} = \mathit{FORW} \cup \mathit{UNFORW}) \wedge$$
$$\mathit{Reachable}(\mathit{head}, \mathit{tail}, \mathit{ALIVE}, \mathit{root}) \wedge (\mathit{ALIVE} \bot \mathit{NEW}) \wedge$$
$$\mathit{PtrRg}(\mathit{head}, \mathit{ALIVE}) \wedge \mathit{PtrRg}(\mathit{tail}, \mathit{ALIVE}) \wedge \mathit{Tfun}(\mathit{head}, \mathit{ALIVE})$$
$$\wedge\ \mathit{Tfun}(\mathit{tail}, \mathit{ALIVE}) \wedge (\sharp \mathit{ALIVE} \leqslant \sharp \mathit{NEW})(\mathit{root} \in \mathit{FORW}) \wedge$$
$$(\mathit{scan} \leqslant \mathit{free}) \wedge \mathit{Ptr}(\mathit{free}) \wedge \mathit{Ptr}(\mathit{scan}) \wedge \mathit{Ptr}(\mathit{offset}) \wedge$$
$$\mathit{Ptr}(\mathit{maxFree}) \wedge \forall_{*} y \in \mathit{UNFORW}.((\exists z.\,(y, z) \in \mathit{head} \wedge$$
$$y \mapsto z) * (\exists z'.\,(y, z') \in \mathit{tail} \wedge y + 4 \mapsto z)) * \forall_{*} y \in \mathit{FORW}.(\exists z.$$
$$(y, z) \in \phi \wedge y \mapsto z, -) * \forall_{*} y \in \mathit{UNFIN}.((\exists z.\,(y, z) \in \mathit{head} \circ \phi^{\dagger} \wedge$$
$$y \mapsto z) * (\exists z'.\,(y, z') \in \mathit{tail} \circ \phi^{\dagger} \wedge y + 4 \mapsto z')) *$$
$$\forall_{*} y \in \mathit{FIN}.((\exists z.\,(y, z) \in \phi \circ (\mathit{head} \circ \phi^{\dagger}) \wedge y \mapsto z) *$$
$$(\exists z'.\,(y, z') \in \phi \circ (\mathit{tail} \circ \phi^{\dagger}) \wedge y + 4 \mapsto z')) *$$
$$\forall_{*} y \in \mathit{FREE}.y \mapsto -, -$$

Loop Invariant

$$\mathrm{iso}(\phi, \mathit{FORW}, \mathit{BUSY}) \wedge (\mathit{ALIVE} = \mathit{FORW} \cup \mathit{UNFORW}) \wedge$$
$$\mathit{Reachable}(\mathit{head}, \mathit{tail}, \mathit{ALIVE}, \mathit{root}) \wedge (\mathit{ALIVE} \perp \mathit{NEW}) \wedge$$
$$\mathit{PtrRg}(\mathit{head}, \mathit{ALIVE}) \wedge \mathit{PtrRg}(\mathit{tail}, \mathit{ALIVE}) \wedge \mathit{Tfun}(\mathit{head}, \mathit{ALIVE})$$
$$\wedge\ \mathit{Tfun}(\mathit{tail}, \mathit{ALIVE}) \wedge (\sharp\mathit{ALIVE} \leqslant \sharp\mathit{NEW})(\mathit{root} \in \mathit{FORW}) \wedge$$
$$(\mathit{scan} \leqslant \mathit{free}) \wedge \mathit{Ptr}(\mathit{free}) \wedge \mathit{Ptr}(\mathit{scan}) \wedge \mathit{Ptr}(\mathit{offset}) \wedge$$
$$\mathit{Ptr}(\mathit{maxFree}) \wedge \forall_{*} y \in \mathit{UNFORW}.((\exists z.\,(y, z) \in \mathit{head} \wedge$$
$$y \mapsto z) * (\exists z'.\,(y, z') \in \mathit{tail} \wedge y + 4 \mapsto z)) * \forall_{*} y \in \mathit{FORW}.(\exists z.$$
$$(y, z) \in \phi \wedge y \mapsto z, -) * \forall_{*} y \in \mathit{UNFIN}.((\exists z.\,(y, z) \in \mathit{head} \circ \phi^{\dagger} \wedge$$
$$y \mapsto z) * (\exists z'.\,(y, z') \in \mathit{tail} \circ \phi^{\dagger} \wedge y + 4 \mapsto z')) *$$
$$\forall_{*} y \in \mathit{FIN}.((\exists z.\,(y, z) \in \phi \circ (\mathit{head} \circ \phi^{\dagger}) \wedge y \mapsto z) *$$
$$(\exists z'.\,(y, z') \in \phi \circ (\mathit{tail} \circ \phi^{\dagger}) \wedge y + 4 \mapsto z')) *$$
$$\forall_{*} y \in \mathit{FREE}.y \mapsto -, -$$

With Framing [BTR POPL'04]

Loop Invariant

$$\mathrm{iso}(\phi, \mathit{FORW}, \mathit{BUSY}) \wedge (\mathit{ALIVE} = \mathit{FORW} \cup \mathit{UNFORW}) \wedge$$
$$\mathit{Reachable}(\mathit{head}, \mathit{tail}, \mathit{ALIVE}, \mathit{root}) \wedge (\mathit{ALIVE} \perp \mathit{NEW}) \wedge$$
$$\mathit{PtrRg}(\mathit{head}, \mathit{ALIVE}) \wedge \mathit{PtrRg}(\mathit{tail}, \mathit{ALIVE}) \wedge \mathit{Tfun}(\mathit{head}, \mathit{ALIVE})$$
$$\wedge\ \mathit{Tfun}(\mathit{tail}, \mathit{ALIVE}) \wedge (\sharp \mathit{ALIVE} \leqslant \sharp \mathit{NEW})(\mathit{root} \in \mathit{FORW}) \wedge$$
$$(\mathit{scan} \leqslant \mathit{free}) \wedge \mathit{Ptr}(\mathit{free}) \wedge \mathit{Ptr}(\mathit{scan}) \wedge \mathit{Ptr}(\mathit{offset}) \wedge$$
$$\mathit{Ptr}(\mathit{maxFree}) \wedge \forall_* y \in \mathit{UNFORW}.((\exists z.\, (y, z) \in \mathit{head} \wedge$$
$$y \mapsto z) * (\exists z'.\, (y, z') \in \mathit{tail} \wedge y + 4 \mapsto z)) * \forall_* y \in \mathit{FORW}.(\exists z.$$
$$(y, z) \in \phi \wedge y \mapsto z, -) * \forall_* y \in \mathit{UNFIN}.((\exists z.\, (y, z) \in \mathit{head} \circ \phi^\dagger \wedge$$
$$y \mapsto z) * (\exists z'.\, (y, z') \in \mathit{tail} \circ \phi^\dagger \wedge y + 4 \mapsto z')) *$$
$$\forall_* y \in \mathit{FIN}.((\exists z.\, (y, z) \in \phi \circ (\mathit{head} \circ \phi^\dagger) \wedge y \mapsto z) *$$
$$(\exists z'.\, (y, z') \in \phi \circ (\mathit{tail} \circ \phi^\dagger) \wedge y + 4 \mapsto z')) *$$
$$\forall_* y \in \mathit{FREE}.y \mapsto -, -$$

With Framing [BTR POPL'04]

Loop Invariant

$$\mathrm{iso}(\phi, FORW, BUSY) \wedge (ALIVE = FORW \cup UNFORW) \wedge$$
$$Reachable(head, tail, ALIVE, root) \wedge (ALIVE \perp NEW) \wedge$$
$$PtrRg(head, ALIVE) \wedge PtrRg(tail, ALIVE) \wedge Tfun(head, ALIVE)$$
$$\wedge\ Tfun(tail, ALIVE) \wedge (\sharp ALIVE \leqslant \sharp NEW)(root \in FORW) \wedge$$
$$(scan \leqslant free) \wedge Ptr(free) \wedge Ptr(scan) \wedge Ptr(offset) \wedge$$
$$Ptr(maxFree) \wedge \forall_{\ast} y \in UNFORW.((\exists z.\,(y, z) \in head \wedge$$
$$y \mapsto z) \ast (\exists z'.\,(y, z') \in tail \wedge y + 4 \mapsto z)) \ast \forall_{\ast} y \in FORW.(\exists z.$$
$$(y, z) \in \phi \wedge y \mapsto z, -) \ast \forall_{\ast} y \in UNFIN.((\exists z.\,(y, z) \in head \circ \phi^{\dagger} \wedge$$
$$y \mapsto z) \ast (\exists z'.\,(y, z') \in tail \circ \phi^{\dagger} \wedge y + 4 \mapsto z')) \ast$$
$$\forall_{\ast} y \in FIN.((\exists z.\,(y, z) \in \phi \circ (head \circ \phi^{\dagger}) \wedge y \mapsto z) \ast$$
$$(\exists z'.\,(y, z') \in \phi \circ (tail \circ \phi^{\dagger}) \wedge y + 4 \mapsto z')) \ast$$
$$\forall_{\ast} y \in FREE.y \mapsto -, -$$

With Framing [BTR POPL'04]

Loop Invariant

$$\mathrm{iso}(\phi, FORW, BUSY) \wedge (ALIVE = FORW \cup UNFORW) \wedge Reachable(head, tail, ALIVE, root) \wedge (ALIVE \perp NEW) \wedge PtrRg(head, ALIVE) \wedge PtrRg(tail, ALIVE) \wedge Tfun(head, ALIVE) \wedge Tfun(tail, ALIVE) \wedge (\sharp ALIVE \leqslant \sharp NEW)(root \in FORW) \wedge (scan \leqslant free) \wedge Ptr(free) \wedge Ptr(scan) \wedge Ptr(offset) \wedge Ptr(maxFree) \wedge \forall_{*} y \in UNFORW.((\exists z.\,(y,z) \in head \wedge y \mapsto z) * (\exists z'.\,(y,z') \in tail \wedge y+4 \mapsto z)) * \forall_{*} y \in FORW.(\exists z.\,(y,z) \in \phi \wedge y \mapsto z, -) * \forall_{*} y \in UNFIN.((\exists z.\,(y,z) \in head \circ \phi^{\dagger} \wedge y \mapsto z) * (\exists z'.\,(y,z') \in tail \circ \phi^{\dagger} \wedge y+4 \mapsto z')) * \forall_{*} y \in FIN.((\exists z.\,(y,z) \in \phi \circ (head \circ \phi^{\dagger}) \wedge y \mapsto z) * (\exists z'.\,(y,z') \in \phi \circ (tail \circ \phi^{\dagger}) \wedge y+4 \mapsto z')) * \forall_{*} y \in FREE.y \mapsto -, -$$

With Framing [BTR POPL'04]

Loop Invariant

$\mathrm{iso}(\phi, \mathit{FORW}, \mathit{BUSY}) \wedge (\mathit{ALIVE} = \mathit{FORW} \cup \mathit{UNFORW}) \wedge$
$\mathit{Reachable}(\mathit{head}, \mathit{tail}, \mathit{ALIVE}, \mathit{root}) \wedge (\mathit{ALIVE} \perp \mathit{NEW}) \wedge$
$\mathit{PtrRg}(\mathit{head}, \mathit{ALIVE}) \wedge \mathit{PtrRg}(\mathit{tail}, \mathit{ALIVE}) \wedge \mathit{Tfun}(\mathit{head}, \mathit{ALIVE})$
$\wedge\ \mathit{Tfun}(\mathit{tail}, \mathit{ALIVE}) \wedge (\sharp\mathit{ALIVE} \leqslant \sharp\mathit{NEW})(\mathit{root} \in \mathit{FORW}) \wedge$
$(\mathit{scan} \leqslant \mathit{free}) \wedge \mathit{Ptr}(\mathit{free}) \wedge \mathit{Ptr}(\mathit{scan}) \wedge \mathit{Ptr}(\mathit{offset}) \wedge$
$\mathit{Ptr}(\mathit{maxFree}) \wedge \forall_{\ast} y \in \mathit{UNFORW}.((\exists z.\,(y,z) \in \mathit{head} \wedge$
$y \mapsto z) \ast (\exists z'.\,(y,z') \in \mathit{tail} \wedge y + 4 \mapsto z)) \ast \forall_{\ast} y \in \mathit{FORW}.(\exists z.$
$(y,z) \in \phi \wedge y \mapsto z, -) \ast \forall_{\ast} y \in \mathit{UNFIN}.((\exists z.\,(y,z) \in \mathit{head} \circ \phi^{\dagger} \wedge$
$y \mapsto z) \ast (\exists z'.\,(y,z') \in \mathit{tail} \circ \phi^{\dagger} \wedge y + 4 \mapsto z')) \ast$
$\forall_{\ast} y \in \mathit{FIN}.((\exists z.\,(y,z) \in \phi \circ (\mathit{head} \circ \phi^{\dagger}) \wedge y \mapsto z) \ast$
$(\exists z'.\,(y,z') \in \phi \circ (\mathit{tail} \circ \phi^{\dagger}) \wedge y + 4 \mapsto z')) \ast$
$\forall_{\ast} y \in \mathit{FREE}.y \mapsto -, -$

$$\mathsf{graph}(x,\gamma) \stackrel{\mathrm{def}}{=} (x = 0 \land \mathsf{emp}) \lor \exists l, r.\, \gamma(x) = (l, r) \land x \mapsto l, r \circledast \mathsf{graph}(l,\gamma) \circledast \mathsf{graph}(r,\gamma)$$

Loop Invariant

$$\mathsf{r} \mapsto \mathsf{to} \star (\mathsf{graph}(\mathsf{to},\gamma) \circledast \mathsf{fromsp}) \star \mathsf{pool}(\mathsf{free}) \land \gamma @\mathsf{to} \approx \gamma_0 @ r_0 \land \mathit{cheney}(\gamma, \mathsf{scan}, \mathsf{free})$$

$$\mathsf{graph}(x,\gamma) \stackrel{\text{def}}{=} (x = 0 \wedge \mathsf{emp}) \vee \exists l, r.\, \gamma(x) = (l, r) \wedge x \mapsto l, r \circledast \mathsf{graph}(l,\gamma) \circledast \mathsf{graph}(r,\gamma)$$

Loop Invariant

$$\mathsf{r} \mapsto \mathsf{to} \star (\mathsf{graph}(\mathsf{to},\gamma) \circledast \mathsf{fromsp}) \star \mathsf{pool}(\mathsf{free}) \wedge \gamma @ \mathsf{to} \approx \gamma_0 @ r_0 \wedge \mathit{cheney}(\gamma, \mathsf{scan}, \mathsf{free})$$

Mathematical Predicate

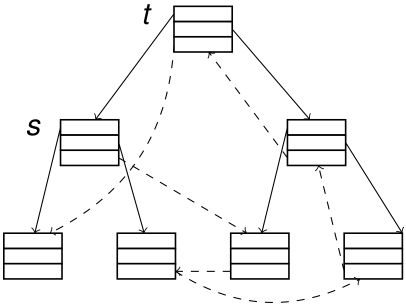
$$\mathit{cheney}(\gamma, s, f) \stackrel{\text{def}}{=} \mathit{to}(s) \wedge \mathit{to}(f) \wedge$$
$$|\{v \mid \mathit{copied}(\gamma, v)\}| = (f - \mathtt{to})/2 \wedge \{\mathtt{to}, \ldots, f-2\} \subseteq \gamma \downarrow \mathtt{to} \wedge$$
$$\forall v \in \gamma.\, \forall a, b.\, \gamma(v) = (a, b) \Rightarrow$$
$$(\mathit{to}(v) \wedge ((v < s \wedge \mathit{to}(a)) \vee (v \geqslant s \wedge \mathit{from}(a)))$$
$$\wedge ((v{+}1 < s \wedge \mathit{to}(b)) \vee (v{+}1 \geqslant s \wedge \mathit{from}(b)))) \vee$$
$$(\mathit{from}(v) \wedge \mathit{from}(b) \wedge (\mathit{to}(a) \Rightarrow \gamma @ b \approx \gamma @ (\gamma(a).2)))$$

More from the Paper…

Classical Conjunction

- $\sigma \models P_1 \wedge P_2$ iff $\sigma \models P_1$ & $\sigma \models P_2$

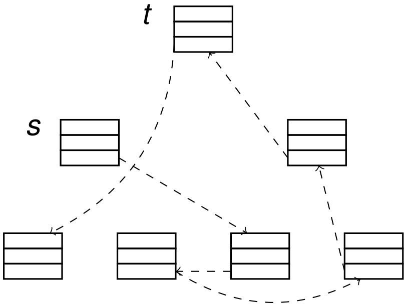
- list(s) $\wedge$ tree(t)



Classical Conjunction

- $\sigma \models P_1 \wedge P_2$ iff $\sigma \models P_1$ & $\sigma \models P_2$

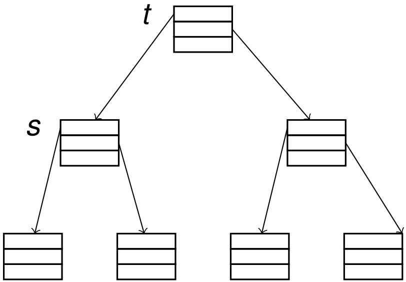
- $\mathsf{list}(s) \wedge \mathsf{tree}(t)$



Classical Conjunction

- $\sigma \models P_1 \wedge P_2$ iff $\sigma \models P_1$ & $\sigma \models P_2$

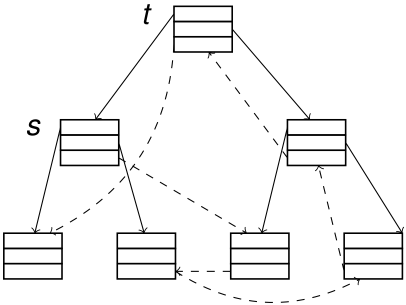
- list(s) $\wedge$ tree(t)



Classical Conjunction

- $\sigma \models P_1 \wedge P_2$ iff $\sigma \models P_1$ & $\sigma \models P_2$

- $\mathsf{list}(s) \wedge \mathsf{tree}(t)$



Even more Stuff from the Paper

Ramification Library

- Generic simplifications, *e.g.*

Disjoint Ramification

$$\frac{R \vdash P * (P' \mathrel{-\!\!*} R') \qquad S \vdash Q * (Q' \mathrel{-\!\!*} S')}{R * S \vdash P * Q * (P' * Q' \mathrel{-\!\!*} R' * S')}$$

- Specific graph & dag lemmas

Additional Program Proofs

$\{\mathsf{dag}(x, \delta)\}$ y = copy_dag(x) $\{\mathsf{dag}(x, \delta) * \mathsf{dag}(y, \delta')\}$

$\{\mathsf{graph}(x, \gamma)\}$ span(x) $\{\mathsf{tree}(x, \tau) \land \mathit{reach}(\tau, x) = \mathit{reach}(\gamma, x)\}$

$\{\mathsf{graph}(x, \gamma))\}$ dispose_graph $\{\mathsf{emp}\}$

Conclusion

Sharing in Data Structures

- Naturally expressed using $\ast$, $\circledast$ and $\wedge$
- Prevents natural applications of the frame rule

Ramify Rule

- By-hand, concise, compositional proofs
- Moves the complexity from space land to math land
- Valid in any separation logic

Future Work

Prove…

- …more programs
- …concurrent ones
- …more automatically
- …machine checked

The Ramifications of Sharing in Data Structures

Aquinas Hobor
National University of Singapore

Jules Villard
University College London