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 Programs with Sharing

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

Frame

\[
\{ \begin{array}{c}
\triangleleft \\
\ell \\
r
\end{array} \} \quad \text{mark}(\ell) \quad \{ \begin{array}{c}
\triangledown \\
\ell \\
r
\end{array} \} \\
\begin{array}{c}
\bigcirc \\
\ell \\
r
\end{array} \quad \text{Frame} = \\
\begin{array}{c}
\bigcirc \\
r
\end{array}
\]

The AI Frame Problem

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

Success: Separation Logic

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

Introduction
Compositional Reasoning for the Heap

Frame

\[
\begin{align*}
\{ \ell \} & \quad \text{mark}(\ell) \quad \{ \ell \} \\
\{ \ell, r \} & \quad \text{mark}(\ell) \quad \{ \ell, r \}
\end{align*}
\]

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
This Talk: Ramification

Ramification

\[
\{ \triangleleft \} \text{ mark}(\ell) \{ \triangleleft \} \quad \frac{\triangleleft \sim \triangleleft}{\triangleleft \triangleright} \quad \{ \triangleleft \} \text{ mark}(\ell) \{ \triangleleft \}
\]

The AI Ramification Problem

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

Key Points

- Embrace sharing when it is natural (\(*, \&\), \(\wedge\), …)
- Separate spatial and mathematical reasoning

Introduction
Marking a Dag
Separating Conjunction

- $\sigma_1 \bullet \sigma_2$ is the **disjoint union** of $\sigma_1$ and $\sigma_2$
- $\sigma \models P_1 \ast P_2$ iff $\exists \sigma_1, \sigma_2. \sigma = \sigma_1 \bullet \sigma_2 \& \sigma_1 \models P_1 \& \sigma_2 \models P_2$
### Separating Conjunction

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

### Heap Assertions

- `emp` empty heap
- `x ↦ a` only `x` is allocated ($\star x == a$)
- `x ↦ a, b, c` $x + 0 \leftrightarrow a \cdot x + 1 \leftrightarrow b \cdot x + 2 \leftrightarrow c$
Tree Logical Predicate [Rey LICS’02]

Separating Conjunction

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

- Mathematical trees (terms)

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

- Spatial trees

$$\text{tree}(x, \tau) \overset{\text{def}}{=} (x = 0 \land \tau = E \land \text{emp})$$

$$\lor \left( \exists \ell, r, v, \tau_\ell, \tau_r. \tau = N(v, \tau_\ell, \tau_r) \land x \mapsto v, \ell, r \ast \text{tree}(\ell, \tau_\ell) \ast \text{tree}(r, \tau_r) \right)$$
Dag Logical Predicate [Rey Unpub’03]

Overlapping Conjunction

- \( \sigma \models P_1 \star P_2 \) iff

\[ \exists \sigma_1, \sigma_2, \sigma_3. \sigma = \sigma_1 \bullet \sigma_2 \bullet \sigma_3 \land \sigma_1 \bullet \sigma_2 \models P_1 \land \sigma_2 \bullet \sigma_3 \models P_2 \]
Overlapping Conjunction

- $\sigma \models P_1 \odot P_2$ iff
  $\exists \sigma_1, \sigma_2, \sigma_3. \sigma = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \land \sigma_1 \cdot \sigma_2 \models P_1 \land \sigma_2 \cdot \sigma_3 \models P_2$

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

- Spatial dags

\[
\text{dag}(x, \gamma) \overset{\text{def}}{=} (x = 0 \land \text{emp})
\]

\[
\lor \exists \ell, r, v. \gamma(x) = (v, \ell, r) \land
\text{x} \mapsto v, \ell, r \ast (\text{dag}(\ell, \gamma) \odot \text{dag}(r, \gamma))
\]
# Overlapping Conjunction

- $\sigma \models P_1 \otimes P_2$ iff
  \[ \exists \sigma_1, \sigma_2, \sigma_3. \sigma = \sigma_1 \bullet \sigma_2 \bullet \sigma_3 \land \sigma_1 \bullet \sigma_2 \models P_1 \land \sigma_2 \bullet \sigma_3 \models P_2 \]

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

- Spatial graphs

  \[
  \text{graph}(x, \gamma) \overset{\text{def}}{=} (x = 0 \land \text{emp})
  \]

  \[
  \lor \exists \ell, r, v. \gamma(x) = (v, \ell, r) \land
  \]

  \[
  x \leftrightarrow (v, \ell, r \otimes \text{graph}(\ell, \gamma) \otimes \text{graph}(r, \gamma))
  \]
Overlapping conjunction

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

The Frame Rule

\[
\frac{\{P\} \circ \{Q\}}{\{P \times F\} \circ \{Q \times F\}}
\]

\[
\frac{\{\text{tree}(\ell, \tau_\ell)\} \text{mark}(\ell) \{\text{tree}(\ell, \tau'_\ell)\}}{\{\text{tree}(\ell, \tau_\ell) \times \text{tree}(r, \tau_r)\} \text{mark}(\ell) \{\text{tree}(\ell, \tau'_\ell) \times \text{tree}(r, \tau_r)\}}
\]

Challenge

How to use ☑ for verification?
Specification of `mark_dag`

```c
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

$$L'(y) = \begin{cases} 
1 & \text{if } y \in \text{reach}_\text{zero}(\gamma, x) \\
L(y) & \text{otherwise}
\end{cases}$$
Proof of `mark_dag` Using Framing

```c
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;

    //
}
```

The Frame Rule

Marking a 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

```c
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)) }
    // \[ \wedge \gamma(d) = (0,l,r) \]
    //
    mark_dag(l);
    //
    mark_dag(r);
    //
    d->m = 1;
    //
} // {dag(d, m(γ, d))}
```

The Frame Rule

\[ \{P\} c \{Q\} \]
\[ \{P \ast F\} c \{Q \ast F\} \]
Proof of `mark_dag` Using Framing

```c
struct node {int 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

\[
\begin{array}{c}
\{P\} c \{Q\} \\
\{P \ast F\} c \{Q \ast F\}
\end{array}
\]
Tailoring Dags to Frame [BOC SPACE’04]

- **Separated** mathematical dag (term)

  \[ \delta \overset{\text{def}}{=} \text{Empty} \mid x : \text{Node} \delta_{\ell} \delta_{r} \mid \text{Ptr } x \]

- Spatial **separated** dags

  \[
  \begin{align*}
  \text{pdag}(0, \text{Empty}) & \overset{\text{def}}{=} \text{emp} \\
  \text{pdag}(x, x: \text{Node } v \delta_{\ell} \delta_{r}) & \overset{\text{def}}{=} \exists \ell, r. x \mapsto v, \ell, r \ast \text{pdag}(\ell, \delta_{\ell}) \ast \text{pdag}(r, \delta_{r}) \\
  \text{pdag}(x, \text{Ptr } x) & \overset{\text{def}}{=} \text{emp}
  \end{align*}
  \]

---

### Caveats

- Predicate depends on the order of traversal
- Complex specifications and invariants
Proof of `mark_dag` using Ramification

```c
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

```c
void mark_dag(struct node *d) { // {dag(d, γ)}
    if (!d || d->m) return;
    struct node *l = d->l, *r = d->r;
    // {d \rightarrow 0, l, r \ast (dag(l, γ) \ast dag(r, γ))}
    // {d \rightarrow 0, l, r \ast (dag(l, γ) \ast dag(r, γ))}
    // \land \gamma(d) = (0, l, r)
    mark_dag(l);
    //
    mark_dag(r);
    //
    d->m = 1;
    //
}
```

Marking a Dag • Using Ramification
Proof of `mark_dag` using Ramification

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

    struct node *l = d->l, *r = d->r;

    // d ÞÑ 0, l, r Ÿ (dag(l, γ) Ÿ dag(r, γ))
    //    ^ γ(d) = (0, l, r)
    mark_dag(l);

    //
    mark_dag(r);

    //
    d->m = 1;

    // {dag(d, m(γ, d))}
}
```
Proof of \texttt{mark\_dag} using Ramification

1. \texttt{void mark\_dag(struct node *d) \{ // \{dag(d, \gamma)\}}
2. \quad \textit{if (!d || d->m) return;}
3. \quad \texttt{struct node *l = d->l, *r = d->r;}
4. \quad \texttt{// \{ d \mapsto 0, l, r * (dag(l, \gamma) \circ dag(r, \gamma))\}}
5. \quad \texttt{\quad \wedge \gamma(d) = (0, l, r)}
6. \quad \texttt{mark\_dag(l);} \texttt{\}}
7. \quad \texttt{// \}}
8. \quad \texttt{mark\_dag(r);} \texttt{\}}
9. \quad \texttt{// \}}
10. \quad \texttt{d->m = 1;} \texttt{\}}
11. \quad \texttt{// \{dag(d, m(\gamma, d))\}}
12. \quad \texttt{\}} \texttt{// \{dag(d, \gamma)\}}
The Ramify Rule of Separation Logic

The Ramify Rule

\[
\{P\} \Rightarrow \{Q\} \\
\hline
\{R\} \Rightarrow \{R'\}
\]
### The Ramify Rule of Separation Logic

**The Ramify Rule**

\[
\begin{array}{c}
\{P\} \text{ c } \{Q\} \\
\text{ramify}(P \rightsquigarrow Q, R) = R'
\end{array}
\]

\[
\{R\} \text{ c } \{R'\}
\]
The Ramify Rule of Separation Logic

The Ramify Rule

\[
\begin{aligned}
\{P\} &\c c \{Q\} & R &\vdash P \ast (Q \rightarrow R') \\
\{R\} &\c c \{R'\}
\end{aligned}
\]

Magic Wand

- \(\sigma \models P \rightarrow Q\) iff \(\forall \sigma' \models P. \sigma \bullet \sigma' \models Q\)

Marking a Dag • Using Ramification
The Ramify Rule of Separation Logic

The Ramify Rule

\[
\{P\} c \{Q\} \quad R \vdash P \ast (Q \ast R') \\
\{R\} c \{R'\}
\]

Magic Wand

- \(\sigma \models P \ast Q\) iff \(\forall \sigma' \models P. \sigma \bullet \sigma' \models Q\)

Ramification Entailment

Marking a Dag • Using Ramification
The Ramify Rule of Separation Logic

The Ramify Rule

\[ \{P\} \sqcap \{Q\} \quad R \models P \ast (Q \rightarrow R') \]

\[ \{R\} \sqcap \{R'\} \]

Magic Wand

- \( \sigma \models P \rightarrow Q \) iff \( \forall \sigma' \models P. \sigma \cdot \sigma' \models Q \)

Ramification Entailment

Marking a Dag • Using Ramification
The Ramify Rule of Separation Logic

The Ramify Rule

\[
\{P\} \mathbin{c} \{Q\} \quad R \vdash P \star (Q \rightarrow R')
\]

\[
\{R\} \mathbin{c} \{R'\}
\]

Magic Wand

\[\sigma \models P \rightarrow Q \iff \forall \sigma' \models P. \sigma \bullet \sigma' \models Q\]

Ramification Entailment

Marking a Dag • Using Ramification
The Ramify Rule of Separation Logic

**The Ramify Rule**

\[
\{P\} \cap \{Q\} \quad R \models P \cdot (Q \rightarrow R') \\
\{R\} \cap \{R'\}
\]

**Magic Wand**

- \(\sigma \models P \rightarrow Q\) iff \(\forall \sigma' \models P. \sigma \bullet \sigma' \models Q\)

**Ramification Entailment**
Back to \texttt{mark\_dag}: First Recursive Call

The Ramify Rule

\[
\begin{align*}
\{P\} & \text{ c } \{Q\} & & R \leftarrow P \ast (Q \rightarrow R') \\
& & \Rightarrow & \{R\} \text{ c } \{R'\}
\end{align*}
\]

// \{\texttt{dag}(l, \gamma) \uplus \texttt{dag}(r, \gamma)\}

\texttt{mark\_dag}(l);

// \frac{1}{2}
Back to `mark_dag`: First Recursive Call

**The Ramify Rule**

\[
\begin{align*}
\{P\} &\cap \{Q\} \\
R &\vdash P \star (Q \rightarrow R') \\
\{R\} &\cap \{R'\}
\end{align*}
\]

```
// {dag(l, γ) ⊗ dag(r, γ)}
mark_dag(l);
// \{\text{dag}(l, m(γ, 1)) \otimes \text{dag}(r, m(γ, 1))\}
```

**Ramification Condition**

\[
\begin{align*}
\text{dag}(\ell, γ) &\otimes \text{dag}(r, γ) \\
\vdash \text{dag}(\ell, γ) \star (\text{dag}(\ell, m(γ, \ell)) \rightarrow \text{dag}(\ell, m(γ, \ell)) \otimes \text{dag}(r, m(γ, \ell)))
\end{align*}
\]
<table>
<thead>
<tr>
<th>Lemma</th>
<th>SubDAG Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{reach}(\gamma', x) \supseteq \text{reach}(\gamma, x)$</td>
<td>$\text{unreach}(\gamma', x) = \text{unreach}(\gamma, x)$</td>
</tr>
<tr>
<td>$\text{dag}(x, \gamma) \uplus \text{dag}(y, \gamma)$ $\vdash$ $\text{dag}(x, \gamma) \ast$</td>
<td>$(\text{dag}(x, \gamma') \rightarrow$</td>
</tr>
<tr>
<td></td>
<td>$\text{dag}(x, \gamma') \uplus \text{dag}(y, \gamma'))$</td>
</tr>
</tbody>
</table>

- $\text{reach}(\gamma', x) \supseteq \text{reach}(\gamma, x)$: no deallocation
- $\text{unreach}(\gamma', x) = \text{unreach}(\gamma, x)$: local modification
## Ramification Library: Updating DAGs

<table>
<thead>
<tr>
<th>Lemma</th>
<th>SubDAG Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{reach}(\gamma', x) \supseteq \text{reach}(\gamma, x) )</td>
<td>( \text{unreach}(\gamma', x) = \text{unreach}(\gamma, x) )</td>
</tr>
<tr>
<td>( \text{dag}(x, \gamma) \uplus \text{dag}(y, \gamma) \vdash \text{dag}(x, \gamma) \ast )</td>
<td>(( \text{dag}(x, \gamma') \rightarrow )</td>
</tr>
<tr>
<td>( \text{dag}(x, \gamma') \uplus \text{dag}(y, \gamma') ))</td>
<td>(( \text{dag}(x, \gamma') \rightarrow )</td>
</tr>
</tbody>
</table>

### First Recursive Call

```c
// {dag(l, \gamma) \uplus dag(r, \gamma)}
mark_dag(l);
// \{dag(l, m(\gamma, l)) \uplus dag(r, m(\gamma, l))}\n
\text{dag}(\ell, \gamma) \uplus \text{dag}(r, \gamma) \vdash \text{dag}(\ell, \gamma) \ast (\text{dag}(\ell, m(\gamma, \ell)) \rightarrow \text{dag}(\ell, m(\gamma, \ell)) \uplus \text{dag}(r, m(\gamma, \ell)))
```
### Lemma SubDAG Update

<table>
<thead>
<tr>
<th>reach($\gamma', x$) $\supseteq$ reach($\gamma, x$)</th>
<th>unreach($\gamma', x$) = unreach($\gamma, x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dag}(x, \gamma) \Join \text{dag}(y, \gamma) \vdash \text{dag}(x, \gamma) \ast (\text{dag}(x, \gamma') \rightarrow$</td>
<td></td>
</tr>
<tr>
<td>$\text{dag}(x, \gamma') \Join \text{dag}(y, \gamma')))$</td>
<td></td>
</tr>
</tbody>
</table>

### First Recursive Call

```cpp
// {dag(l, \gamma) \Join dag(r, \gamma)}
mark_dag(l);
//½ {dag(l, m(\gamma, l)) \Join dag(r, m(\gamma, l))}

reach(m(\gamma, l), l) \supseteq reach(\gamma, l)
unr.(m(\gamma, l), l) = unr.(\gamma, l)

\text{dag}(l, \gamma) \Join \text{dag}(r, \gamma) \vdash \text{dag}(l, \gamma) \ast (\text{dag}(l, m(\gamma, l)) \rightarrow$
\text{dag}(l, m(\gamma, l)) \Join \text{dag}(r, m(\gamma, l)))
```
### Lemma SubDAG Update

\[
\begin{align*}
\text{reach}(\gamma', x) & \supseteq \text{reach}(\gamma, x) \quad \text{unreach}(\gamma', x) & = \text{unreach}(\gamma, x) \\
\text{dag}(x, \gamma) \oplus \text{dag}(y, \gamma) & \vdash \text{dag}(x, \gamma) \star \\
& (\text{dag}(x, \gamma') \rightarrow \\
& \text{dag}(x, \gamma') \oplus \text{dag}(y, \gamma'))
\end{align*}
\]

### First Recursive Call

```plaintext
// {dag(l, \gamma) \oplus dag(r, \gamma)}
mark_dag(l);
// \{dag(l, m(\gamma, l)) \oplus dag(r, m(\gamma, l))\}

\textbf{math} \quad \textbf{math}

\begin{align*}
\text{reach}(m(\gamma, \ell), \ell) & \supseteq \text{reach}(\gamma, \ell) \\
\text{unreach}(m(\gamma, \ell), \ell) & = \text{unreach}(\gamma, \ell) \\
\text{dag}(\ell, \gamma) \oplus \text{dag}(r, \gamma) & \vdash \text{dag}(\ell, \gamma) \star \text{dag}(\ell, m(\gamma, \ell)) \rightarrow \\
& \text{dag}(\ell, m(\gamma, \ell)) \oplus \text{dag}(r, m(\gamma, \ell))
\end{align*}
```

---

**Ramification Library: Updating DAGs**
**Proof of `mark_dag` using Ramification**

```c
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

```c
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;
}
```

Marking a Dag • Using Ramification
Proof of mark_dag using Ramification

```c
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

```c
void mark_dag(struct node *d) { // {dag(d, γ)}
    if (!d || d->m) return;
    struct node *l = d->l, *r = d->r;
    // { d ← 0,1,r * (dag(l, γ) ⨀ dag(r, γ)) } \\
    // { \wedge γ(d') = (0,1,r) }
    mark_dag(l);
    // \n    // { d ← 0,1,r * (dag(l, m(γ, 1)) ⨀ dag(r, m(γ, 1))) } \\
    // { \wedge γ(d') = (0,1,r) }
    mark_dag(r);
    // \n    // { d ← 0,1,r * (dag(l, γ') ⨀ dag(r, γ')) } \\
    // { \wedge γ(d') = (0,1,r) \wedge γ' = m(m(γ, 1), r) }
    d->m = 1;
}
```
### Second Recursive Call

\[
\text{dag}(\ell, m(\gamma, \ell)) \ast \text{dag}(r, m(\gamma, \ell)) \\
\vdash \text{dag}(r, m(\gamma, \ell)) \ast (\text{dag}(r, m(m(\gamma, \ell), r)) \rightarrow \\
\text{dag}(\ell, m(m(\gamma, \ell), r)) \ast \text{dag}(r, m(m(\gamma, \ell), r)))
\]

Marking a Dag • Using Ramification
## Ramification Conditions

### Second Recursive Call

\[
dag(\ell, m(\gamma, \ell)) \star \dag(r, m(\gamma, \ell))
\]

\[
\vdash \dag(r, m(\gamma, \ell)) \star (\dag(r, m(m(\gamma, \ell), r)) \to \star \\
\dag(\ell, m(m(\gamma, \ell), r)) \star \dag(r, m(m(\gamma, \ell), r)))
\]

\[
\ell \leftrightarrow r
\]

\[
\gamma \leftarrow m(\gamma, \ell)
\]

### First Recursive Call

\[
dag(\ell, \gamma) \star \dag(r, \gamma)
\]

\[
\vdash \dag(\ell, \gamma) \star (\dag(\ell, m(\gamma, \ell)) \to \star \\
\dag(\ell, m(\gamma, \ell)) \star \dag(r, m(\gamma, \ell)))
\]
Proof of `mark_dag` using Ramification

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

    struct node *l = d->l, *r = d->r;
    // \( \{ \text{d} \mapsto 0, l, r * (\text{dag}(l, γ) \uplus \text{dag}(r, γ)) \} \)
    // \( \land γ(d) = (0, l, r) \)
    mark_dag(l);

    // \( \{ \text{d} \mapsto 0, l, r * (\text{dag}(l, m(γ, l)) \uplus \text{dag}(r, m(γ, l))) \} \)
    // \( \land γ(d) = (0, l, r) \)
    mark_dag(r);

    // \( \{ \text{d} \mapsto 0, l, r * (\text{dag}(l, γ') \uplus \text{dag}(r, γ')) \} \)
    // \( \land γ(d) = (0, l, r) \land γ' = m(m(γ, 1), r) \)
    d->m = 1;

} // {dag(d, m(γ, d))}
```
Proof of `mark_dag` using Ramification

```c
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;
}
```

Marking a Dag • Using Ramification
Proof of \texttt{mark\_dag} using Ramification

\begin{verbatim}
1 void mark_dag(struct node *d) { // \{dag(d, \gamma)\}
2     if (!d || d->m) return;
3     struct node *l = d->l, *r = d->r;
4     // \{d \leftrightarrow 0,1,r \ast (dag(l, \gamma) \ast dag(r, \gamma)) \}
5         // \{\gamma(d') = (0,1,r)\}
6     mark_dag(l);
7     // \{d \leftrightarrow 0,1,r \ast (dag(l, m(\gamma,l)) \ast dag(r, m(\gamma,l))) \}
8         // \{\gamma(d') = (0,1,r)\}
9     mark_dag(r);
10    // \{d \leftrightarrow 0,1,r \ast (dag(l, \gamma') \ast dag(r, \gamma')) \}
11         // \{\gamma(d') = (0,1,r) \land \gamma' = m(m(\gamma,l),r)\}
12     d->m = 1;
13     // \{d \leftrightarrow 1,1,r \ast (dag(l, \gamma') \ast dag(r, \gamma')) \}
14         // \{\gamma(d) = (0,1,r) \land \gamma' = m(m(\gamma,l),r)\}
15 } // \{dag(d, m(\gamma, d))\}
\end{verbatim}
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 \rightarrow 0, l, r * (dag(l, γ) \uplus dag(r, γ))}
// \quad \wedge γ(d') = (0, l, r)
mark_dag(l);

// {d \rightarrow 0, l, r * (dag(l, m(γ, l)) \uplus dag(r, m(γ, l)))}
// \quad \wedge γ(d') = (0, l, r)
mark_dag(r);

// \downarrow {d \rightarrow 0, l, r * (dag(l, γ') \uplus dag(r, γ'))}
// \quad \wedge γ(d') = (0, l, r) \land γ' = m(m(γ, l), r)

d->m = 1;

// {d \rightarrow 1, l, r * (dag(l, γ') \uplus dag(r, γ'))}
// \quad \wedge γ(d) = (0, l, r) \land γ' = m(m(γ, l), r)
} // {dag(d, m(γ, d))}
Establishing the Post-Condition of `mark_dag`

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

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

### Lemma

\[
m(m(m_1(\gamma, x), \ell), r) = m(m_1(m(\gamma, \ell), x), r) = m_1(m(m(\gamma, \ell), r), x) = m(m_1(m(\gamma, r), x), \ell) = m(\gamma, x)
\]

### Post-Condition Entailment

\[
d \mapsto 1, l, r \ast (\text{dag}(l, \gamma') \oplus \text{dag}(r, \gamma')) \\
\land \gamma(d) = (0, \ell, r) \land \gamma' = m(m(\gamma, \ell), r) \\
\vdash \text{dag}(d, m_1(m(m(\gamma, l), r), d)) \\
\vdash \text{dag}(d, m(\gamma, d))
\]
void mark_dag(struct node *d) { // {dag(d, γ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
  // {d \mapsto 0,1,r \ast (\texttt{dag}(l, γ) \# \texttt{dag}(r, γ))}
  // \{ \wedge γ(d) = (0,1,r) \}
  mark_dag(l);
  // \{d \mapsto 0,1,r \ast (\texttt{dag}(l, \texttt{m}(γ,1)) \# \texttt{dag}(r, \texttt{m}(γ,1)))\}
  // \{ \wedge γ(d) = (0,1,r) \}
  mark_dag(r);
  // \{d \mapsto 0,1,r \ast (\texttt{dag}(l, γ') \# \texttt{dag}(r, γ'))\}
  // \{ \wedge γ(d) = (0,1,r) \wedge γ' = \texttt{m}(\texttt{m}(γ,1),r) \}
  d->m = 1;
  // \{d \mapsto 1,1,r \ast (\texttt{dag}(l, γ') \# \texttt{dag}(r, γ'))\}
  // \{ \wedge γ(d) = (0,1,r) \wedge γ' = \texttt{m}(\texttt{m}(γ,1),r) \}
} // {\texttt{dag}(d, m(γ,d))}
Robustness of the Proof

```c
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))}
```
Robustness of the Proof

```c
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(r);
    // {d ↦ 1, l, r * (dag(l, m(γ, r)) ⊗ dag(r, m(γ, r)))}
    // {γ(d) = (0, l, r)}
    mark_dag(l);
    // {d ↦ 1, l, r * (dag(l, γ') ⊗ dag(r, γ'))}
    // {γ(d) = (0, l, r) ∧ γ' = m(m(γ, r), l)}
    // {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

<table>
<thead>
<tr>
<th>Lemma</th>
<th>Single Graph Node Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma(x) = (d, \ell, r) )</td>
<td>( \gamma' = [x \mapsto (d', \ell, r)] \gamma )</td>
</tr>
<tr>
<td>( \text{graph}(x, \gamma) \vdash x \leftrightarrow d, \ell, r \ast (x \mapsto d', \ell, r \rightarrow \text{graph}(x, \gamma')) )</td>
<td></td>
</tr>
</tbody>
</table>

Marking the Root

| \( \gamma(g) = (0, 1, r) \) | \( \gamma_1 = [x \mapsto (1, 1, r)] \gamma \) |
| \( g \leftarrow 0, 1, r \ast \text{graph}(1, \gamma) \ast \text{graph}(r, \gamma) \) | \( \vdash g \leftarrow 0, 1, r \ast (g \leftarrow 1, 1, r \rightarrow \ast) \) |
| \( g \leftarrow 1, 1, r \ast \text{graph}(1, \gamma_1) \ast \text{graph}(r, \gamma_1) \land \gamma_1 = m_1(\gamma, g) \) |
Robustness of the Proof

1 void mark_graph(struct node *g) { // {graph(g, γ)}
2     if (!g || g->m) return;
3     struct node *l = g->l, *r = g->r;
4     // {g ← 0, l, r ⊙ graph(l, γ) ⊙ graph(r, γ)}
5     // {γ(g) = (0, l, r)}
6     g->m = 1;
7     // {g ← 1, l, r ⊙ graph(l, γ_1) ⊙ graph(r, γ_1)}
8     // {γ(g) = (0, l, r) ∧ γ_1 = m_1(γ, g)}
9     mark_graph(r);
10    // {g ← 1, l, r ⊙ graph(l, m(γ_1, r)) ⊙ graph(r, m(γ_1, r))}
11    // {γ(g) = (0, l, r) ∧ γ_1 = m_1(γ, g)}
12    mark_graph(l);
13    // {g ← 1, l, r ⊙ graph(l, γ') ⊙ graph(r, γ')}
14    // {γ(g) = (0, l, r) ∧ γ' = m(m(m_1(γ, g), r), l)}
15 } // {graph(g, m(γ, g))}
Acid Test: Cheney’s GC
Cheney’s Copying Garbage Collector

```c
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;
        }
    }
}
```

Acid Test: Cheney’s GC
Acid Test: Cheney’s GC

State During the Execution

Diagram showing the state during the execution of Cheney’s GC.
Loop Invariant

\[\text{iso}(\phi, \text{FORW}, \text{BUSY}) \land (\text{ALIVE} = \text{FORW} \cup \text{UNFORW}) \land
\text{Reachable}(\text{head}, \text{tail}, \text{ALIVE}, \text{root}) \land (\text{ALIVE} \downarrow \text{NEW}) \land
\text{PtrRg}(\text{head}, \text{ALIVE}) \land \text{PtrRg}(\text{tail}, \text{ALIVE}) \land \text{Tfun}(\text{head}, \text{ALIVE}) \land \text{Tfun}(\text{tail}, \text{ALIVE}) \land (\#\text{ALIVE} \leq \#\text{NEW})(\text{root} \in \text{FORW}) \land
(\text{scan} \leq \text{free}) \land \text{Ptr}(\text{free}) \land \text{Ptr}(\text{scan}) \land \text{Ptr}(\text{offset}) \land
\text{Ptr}(\text{maxFree}) \land \forall \ast y \in \text{UNFORW}.((\exists z. (y, z) \in \text{head} \land
y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \land y + 4 \mapsto z)) \ast \forall \ast y \in \text{FORW}.(\exists z.
(y, z) \in \phi \land y \mapsto z, -) \ast \forall \ast y \in \text{UNFIN}.((\exists z. (y, z) \in \text{head} \circ \phi^\dagger \land
y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \circ \phi^\dagger \land y + 4 \mapsto z')) \ast
\forall \ast y \in \text{FIN}.((\exists z. (y, z) \in \phi \circ (\text{head} \circ \phi^\dagger) \land y \mapsto z) \ast
(\exists z'. (y, z') \in \phi \circ (\text{tail} \circ \phi^\dagger) \land y + 4 \mapsto z')) \ast
\forall \ast y \in \text{FREE}.y \mapsto -, -\]
With Framing [BTR POPL’04]

<table>
<thead>
<tr>
<th>Loop Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso((\phi, \text{FORW}, \text{BUSY})) &amp; ((\text{ALIVE} = \text{FORW} \cup \text{UNFORW})) &amp;</td>
</tr>
<tr>
<td>Reachable(head, tail, ALIVE, root) &amp; ((\text{ALIVE} \downarrow \text{NEW})) &amp;</td>
</tr>
<tr>
<td>(\text{ PtrRg(head, ALIVE)} \land \text{PtrRg(tail, ALIVE)} \land \text{Tfun(head, ALIVE)} \land \text{Tfun(tail, ALIVE)} \land (#\text{ALIVE} \leq #\text{NEW})(\text{root} \in \text{FORW}) \land</td>
</tr>
<tr>
<td>(\text{scan} \leq \text{free}) \land \text{Ptr(free)} \land \text{Ptr(scan)} \land \text{Ptr(offset)} \land</td>
</tr>
<tr>
<td>\text{Ptr(maxFree)} \land \forall \ast y \in \text{UNFORW}.((\exists z. (y, z) \in \text{head} \land</td>
</tr>
<tr>
<td>y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \land y + 4 \mapsto z)) \ast \forall \ast y \in \text{FORW}.(\exists z.</td>
</tr>
<tr>
<td>(y, z) \in \phi \land y \mapsto z, \land) \ast \forall \ast y \in \text{UNFIN}.((\exists z. (y, z) \in \text{head} \circ \phi^\dagger \land</td>
</tr>
<tr>
<td>y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \circ \phi^\dagger \land y + 4 \mapsto z'))) \ast</td>
</tr>
<tr>
<td>\forall \ast y \in \text{FIN}.((\exists z. (y, z) \in \phi \circ (\text{head} \circ \phi^\dagger) \land y \mapsto z) \ast</td>
</tr>
<tr>
<td>(\exists z'. (y, z') \in \phi \circ (\text{tail} \circ \phi^\dagger) \land y + 4 \mapsto z'))) \ast</td>
</tr>
<tr>
<td>\forall \ast y \in \text{FREE}.y \mapsto -, -</td>
</tr>
</tbody>
</table>
### Loop Invariant

\[
\text{iso}(\phi, \text{FORW}, \text{BUSY}) \land (\text{ALIVE} = \text{FORW} \cup \text{UNFORW}) \land \\
\text{Reachable}(\text{head}, \text{tail}, \text{ALIVE}, \text{root}) \land (\text{ALIVE} \downarrow \text{NEW}) \land \\
\text{PtrRg}(\text{head}, \text{ALIVE}) \land \text{PtrRg}(\text{tail}, \text{ALIVE}) \land \text{Tfun}(\text{head}, \text{ALIVE}) \land \\
\text{Tfun}(\text{tail}, \text{ALIVE}) \land (\#\text{ALIVE} \leq \#\text{NEW})(\text{root} \in \text{FORW}) \land \\
(\text{scan} \leq \text{free}) \land \text{Ptr}(\text{free}) \land \text{Ptr}(\text{scan}) \land \text{Ptr}(\text{offset}) \land \\
\text{Ptr}(\text{maxFree}) \land \forall_\star y \in \text{UNFORW}.((\exists z. (y, z) \in \text{head} \land \\
y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \land y + 4 \mapsto z)) \ast \forall_\star y \in \text{FORW}.(\exists z. \\
(y, z) \in \phi \land y \mapsto z, \neg) \ast \forall_\star y \in \text{UNFIN}.((\exists z. (y, z) \in \text{head} \circ \phi^\dagger \land \\
y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \circ \phi^\dagger \land y + 4 \mapsto z'))) \ast \\
\forall_\star y \in \text{FIN}.((\exists z. (y, z) \in \phi \circ (\text{head} \circ \phi^\dagger) \land y \mapsto z) \ast \\
(\exists z'. (y, z') \in \phi \circ (\text{tail} \circ \phi^\dagger) \land y + 4 \mapsto z'))) \ast \\
\forall_\star y \in \text{FREE}.y \mapsto -, -, 
\]
Loop Invariant

iso(φ, FORW, BUSY) \land (ALIVE = FORW \cup UNFORW) \land
Reachable(head, tail, ALIVE, root) \land (ALIVE \downarrow NEW) \land
PtrRg(head, ALIVE) \land PtrRg(tail, ALIVE) \land Tfun(head, ALIVE)
\land Tfun(tail, ALIVE) \land (\#ALIVE \leq \#NEW)(root \in FORW) \land
(scan \leq free) \land Ptr(free) \land Ptr(scan) \land Ptr(offset) \land
Ptr(maxFree) \land \forall y \in UNFORW.((\exists z. (y, z) \in head \land
y \mapsto z) \ast (\exists z'. (y, z') \in tail \land y + 4 \mapsto z)) \ast \forall y \in FORW.(\exists z.
(y, z) \in \phi \land y \mapsto z, ) \ast \forall y \in UNFIN.((\exists z. (y, z) \in head \circ \phi^\dagger \land
y \mapsto z) \ast (\exists z'. (y, z') \in tail \circ \phi^\dagger \land y + 4 \mapsto z')) \ast
\forall y \in FIN.((\exists z. (y, z) \in \phi \circ (head \circ \phi^\dagger) \land y \mapsto z) \ast
(\exists z'. (y, z') \in \phi \circ (tail \circ \phi^\dagger) \land y + 4 \mapsto z')) \ast
\forall y \in FREE.y \mapsto -, -
<table>
<thead>
<tr>
<th>Loop Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso(φ, FORW, BUSY) \land (ALIVE = FORW \cup UNFORW) \land</td>
</tr>
<tr>
<td>Reachable(head, tail, ALIVE, root) \land (ALIVE \downarrow NEW) \land</td>
</tr>
<tr>
<td>PtrRg(head, ALIVE) \land PtrRg(tail, ALIVE) \land Tfun(head, ALIVE)</td>
</tr>
<tr>
<td>\land Tfun(tail, ALIVE) \land (#ALIVE \leq #NEW)(root \in FORW) \land</td>
</tr>
<tr>
<td>(scan \leq free) \land Ptr(free) \land Ptr(scan) \land Ptr(offset) \land</td>
</tr>
<tr>
<td>Ptr(maxFree) \land \forall_y \in UNFORW.((\exists_z. (y, z) \in head \land</td>
</tr>
<tr>
<td>y \mapsto z) \ast (\exists_{z'}. (y, z') \in tail \land y + 4 \mapsto z)) \ast \forall_y \in FORW.((\exists_z.</td>
</tr>
<tr>
<td>(y, z) \in \phi \land y \mapsto z, _ ) \ast \forall_y \in UNFIN.((\exists_z. (y, z) \in head \circ \phi^\uparrow \land</td>
</tr>
<tr>
<td>y \mapsto z) \ast (\exists_{z'}. (y, z') \in tail \circ \phi^\uparrow \land y + 4 \mapsto z'))) \ast</td>
</tr>
<tr>
<td>\forall_y \in FIN.((\exists_z. (y, z) \in \phi \circ (head \circ \phi^\uparrow) \land y \mapsto z) \ast</td>
</tr>
<tr>
<td>(\exists_{z'}. (y, z') \in \phi \circ (tail \circ \phi^\uparrow) \land y + 4 \mapsto z'))) \ast</td>
</tr>
<tr>
<td>\forall_y \in FREE.y \mapsto _ , _</td>
</tr>
</tbody>
</table>
## Loop Invariant

\[ \text{iso}(\phi, \text{FORW}, \text{BUSY}) \land (\text{ALIVE} = \text{FORW} \cup \text{UNFORW}) \land \\
\text{Reachable}(\text{head}, \text{tail}, \text{ALIVE}, \text{root}) \land (\text{ALIVE} \downarrow \text{NEW}) \land \\
\text{PtrRg}(\text{head}, \text{ALIVE}) \land \text{PtrRg}(\text{tail}, \text{ALIVE}) \land \text{Tfun}(\text{head}, \text{ALIVE}) \\
\land \text{Tfun}(\text{tail}, \text{ALIVE}) \land (\#\text{ALIVE} \leq \#\text{NEW})(\text{root} \in \text{FORW}) \land \\
(\text{scan} \leq \text{free}) \land \text{Ptr}(\text{free}) \land \text{Ptr}(\text{scan}) \land \text{Ptr}(\text{offset}) \land \\
\text{Ptr}(\text{maxFree}) \land \forall_y \in \text{UNFORW}.((\exists z. (y, z) \in \text{head} \land \\
y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \land y + 4 \mapsto z)) \ast \forall_y \in \text{FORW}.(\exists z. \\
(y, z) \in \phi \land y \mapsto z, -) \ast \forall_y \in \text{UNFIN}.((\exists z. (y, z) \in \text{head} \circ \phi^\uparrow \land \\
y \mapsto z) \ast (\exists z'. (y, z') \in \text{tail} \circ \phi^\uparrow \land y + 4 \mapsto z')) \ast \\
\forall_y \in \text{FIN}.((\exists z. (y, z) \in \phi \circ (\text{head} \circ \phi^\uparrow) \land y \mapsto z) \ast \\
(\exists z'. (y, z') \in \phi \circ (\text{tail} \circ \phi^\uparrow) \land y + 4 \mapsto z')) \ast \\
\forall_y \in \text{FREE}.y \mapsto -,- \]
With Ramification

\[ \text{graph}(x, \gamma) \overset{\text{def}}{=} (x = 0 \land \text{emp}) \lor \exists l, r. \gamma(x) = (l, r) \land \\
x \mapsto l, r \bowtie \text{graph}(l, \gamma) \bowtie \text{graph}(r, \gamma) \]

Loop Invariant

\[
\begin{align*}
    r &\mapsto \text{to} \bowtie (\text{graph(to, } \gamma) \bowtie \text{fromsp}) \bowtie \text{pool(free)} \land \\
    \gamma @ \text{to} &\approx \gamma_0 @ r_0 \land \text{cheney}(\gamma, \text{scan, free})
\end{align*}
\]
With Ramification

\[
\text{graph}(x, \gamma) \overset{\text{def}}{=} (x = 0 \land \text{emp}) \lor \exists l, r. \gamma(x) = (l, r) \land x \mapsto l, r \ast \text{graph}(l, \gamma) \ast \text{graph}(r, \gamma)
\]

<table>
<thead>
<tr>
<th>Loop Invariant</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
    r & \mapsto \text{to} \ast (\text{graph}(\text{to}, \gamma) \ast \text{fromsp}) \ast \text{pool}(\text{free}) \land \\
    \gamma @ \text{to} & \approx \gamma_0 @ r_0 \land \text{cheney}(\gamma, \text{scan}, \text{free})
\end{align*}
\] |

<table>
<thead>
<tr>
<th>Mathematical Predicate</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
    \text{cheney}(\gamma, s, f) & \overset{\text{def}}{=} \text{to}(s) \land \text{to}(f) \land \\
    |\{v \mid \text{copied}(\gamma, v)\}| & = (f - \text{to}) / 2 \land \{\text{to}, \ldots, f - 2\} \subseteq \gamma \downarrow \text{to} \land \\
    \forall v \in \gamma. \forall a, b. \gamma(v) & = (a, b) \Rightarrow \\
    (\text{to}(v) \land ((v < s \land \text{to}(a)) \lor (v \geq s \land \text{from}(a)))) \land \\
    ((v + 1 < s \land \text{to}(b)) \lor (v + 1 \geq s \land \text{from}(b))) \lor \\
    (\text{from}(v) \land \text{from}(b) \land (\text{to}(a) \Rightarrow \gamma @ b \approx \gamma @ (\gamma(a) . 2)))
\end{align*}
\] |

Acid Test: Cheney’s GC
More from the Paper…
Overlaid Data Structures

Classical Conjunction

- \( \sigma \models P_1 \land P_2 \iff \sigma \models P_1 \land \sigma \models P_2 \)

- list(s) \( \land \) tree(t)

More from the Paper…
Overlaid Data Structures

Classical Conjunction

- \( \sigma \models P_1 \land P_2 \iff \sigma \models P_1 \land \sigma \models P_2 \)

- \( \text{list}(s) \land \text{tree}(t) \)
Overlaid Data Structures

Classical Conjunction

- \( \sigma \models P_1 \land P_2 \iff \sigma \models P_1 \land \sigma \models P_2 \)

- \( \text{list}(s) \land \text{tree}(t) \)
Overlaid Data Structures

Classical Conjunction

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

- $\text{list}(s) \land \text{tree}(t)$
### Ramification Library

- **Generic simplifications, *e.g.*

\[
R \vdash P \cdot (P' \rightarrow R') \quad S \vdash Q \cdot (Q' \rightarrow S') \\
\frac{}{R \cdot S \vdash P \cdot Q \cdot (P' \cdot Q' \rightarrow R' \cdot S')}
\]

- **Specific graph & dag lemmas**

### Additional Program Proofs

\[
\{\text{dag}(x, \delta)\} \ y = \text{copy_dag}(x) \ \{\text{dag}(x, \delta) \cdot \text{dag}(y, \delta')\}
\]

\[
\{\text{graph}(x, \gamma)\} \ \text{span}(x) \ \{\text{tree}(x, \tau) \land \text{reach}(\tau, x) = \text{reach}(\gamma, x)\}
\]

\[
\{\text{graph}(x, \gamma)\} \ \text{dispose_graph} \ \{\text{emp}\}
\]
Conclusion
Summary

### Sharing in Data Structures
- Naturally expressed using $\ast$, $\otimes$ 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