The Ramify Rule of Separation Logic

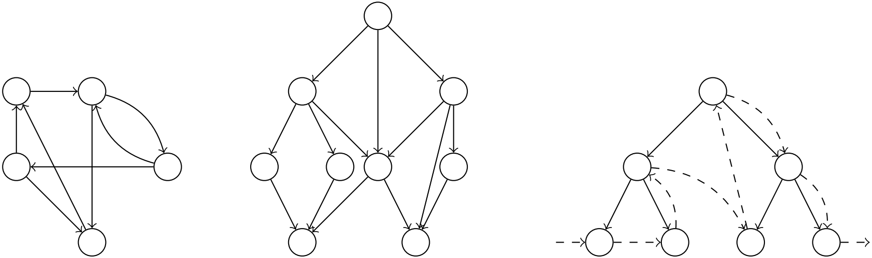
Compositional Reasoning for Sharing

Jules Villard[1]

Joint work with Aquinas Hobor[2]

[1]University College London
[2]National University of Singapore

Programs with Sharing in the Wild



Graphs

Acyclic graphs (DAGs)

Overlaid data structures (threaded tree)

Programs with Sharing

- Everywhere
- Many variations over a few core principles (traversal, marking, copying, ...)
- Short programs, intricate reasoning
- Lots of pointer swinging (tree rotation, Schorr-Waite, ...)
- Challenge for compositionality

Compositional Formal Verification

- Reasoning about a system by reasoning about its parts in isolation
- System = Program
- Parts = Functions
- Reasoning = $\{P\}\ c\ \{Q\}$

Compositionality for Pointer Programs

Success: Separation Logic

- The frame rule provides compositional reasoning:

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

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

An answer to the **frame problem**:

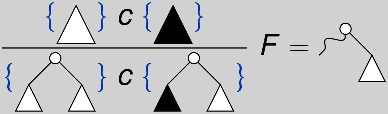
> *"Describing what does not change as a result of an action"*

Compositionality for Pointer Programs

Success: Separation Logic

- The frame rule provides compositional reasoning:

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

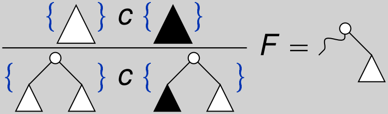


- Data structures without sharing (lists, trees, ...)
- Compositionality based on **disjointness** of memory accesses

An answer to the **frame problem**:

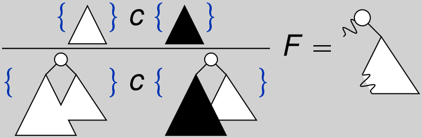
> *"Describing what does not change as a result of an action"*

Frame



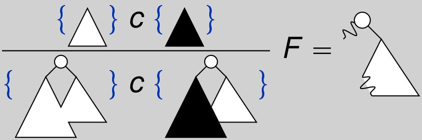
Framing vs Data Structures with Sharing

Frame



Framing vs Data Structures with Sharing

Frame



Previous Attempts

- Contrived predicates that circumvent the sharing
- Leads to compositional, but ad-hoc reasoning
- No general solution

Ramification Problem in AI:

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

Ramification Rule of Separation Logic

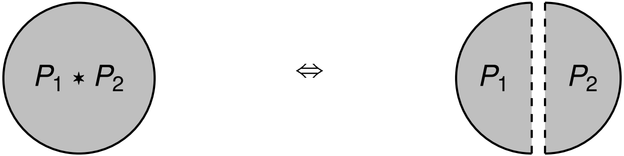
- Embrace sharing
- Concise, compositional proofs
- Expose and resolve global effects of local actions uniformly
- All within vanilla separation logic

Separation, Frame, and Trees

Frame

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

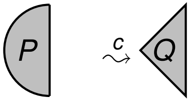
- $\sigma_1 \bullet \sigma_2$ is the disjoint union of σ_1 and σ_2
- $\sigma \models P_1 \star P_2$ iff $\exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \bullet \sigma_2\ \&\ \sigma_1 \models P_1\ \&\ \sigma_2 \models P_2$



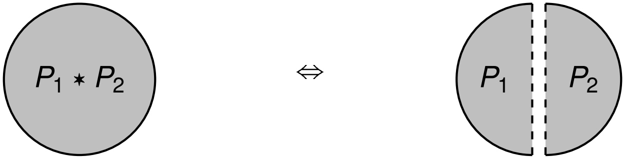
The Frame Rule of Separation Logic

Frame

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

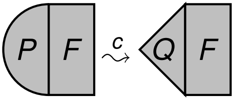


- $\sigma_1 \bullet \sigma_2$ is the disjoint union of σ_1 and σ_2
- $\sigma \models P_1 \star P_2$ iff $\exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \bullet \sigma_2\ \&\ \sigma_1 \models P_1\ \&\ \sigma_2 \models P_2$

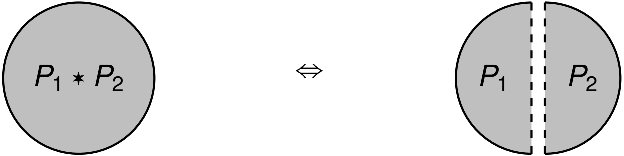


Frame

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



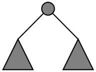
- $\sigma_1 \bullet \sigma_2$ is the disjoint union of σ_1 and σ_2
- $\sigma \vDash P_1 \star P_2$ iff $\exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \bullet \sigma_2 \ \&\ \sigma_1 \vDash P_1 \ \&\ \sigma_2 \vDash P_2$



Binary Trees in SL

$$\mathsf{tree}(x,\tau) \overset{\text{def}}{=}$$
$$(x = 0 \wedge \mathsf{emp} \wedge \tau = \emptyset)$$
$$\vee\ \exists L, R, M, \tau_L, \tau_R.$$
$$x \mapsto m : M, \ell : L, r : R \star$$
$$\mathsf{tree}(L,\tau_L) \star \mathsf{tree}(R,\tau_R) \wedge \tau = \mathit{node}(x, M, \tau_L, \tau_R)$$

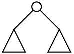
$\emptyset$



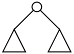
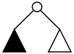
Marking a Tree

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

void mark_tree(struct node *t) { // {tree(t, τ)}        △
  if (!t || t->m) return;
  struct node *l = t->l, *r = t->r;
//
  mark_tree(l);
//
  mark_tree(r);
//
  t->m = 1;
//
} // {tree(t, m(τ))}                                      ▲
```

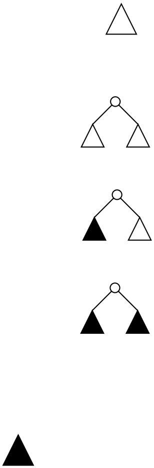
```
struct node {short m; struct node *l,*r;};

void mark_tree(struct node *t) { // {tree(t, τ)}
  if (!t || t->m) return;
  struct node *l = t->l, *r = t->r;
// { t ↦ m : 0, ℓ : l, r : r ∗ tree(l, τ_ℓ) ∗ tree(r, τ_r)
//   ∧ τ = node(0, τ_ℓ, τ_r) }
  mark_tree(l);
//
  mark_tree(r);
//
  t->m = 1;
//
} // {tree(t, m(τ))}
```



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

void mark_tree(struct node *t) { // {tree(t, τ)}
  if (!t || t->m) return;
  struct node *l = t->l, *r = t->r;
// { t ↦ m : 0, ℓ : l, r : r ∗ tree(l, τ_ℓ) ∗ tree(r, τ_r)
//   ∧ τ = node(0, τ_ℓ, τ_r) }
  mark_tree(l);
// { t ↦ m : 0, ℓ : l, r : r ∗ tree(l, m(τ_ℓ)) ∗ tree(r, τ_r)
//   ∧ τ = node(0, τ_ℓ, τ_r) }
  mark_tree(r);

//

  t->m = 1;

//

} // {tree(t, m(τ))}
```


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

void mark_tree(struct node *t) { // {tree(t, τ)}
  if (!t || t->m) return;
  struct node *l = t->l, *r = t->r;
// { t ↦ m: 0, ℓ: l, r: r ∗ tree(l, τ_ℓ) ∗ tree(r, τ_r)
//   ∧ τ = node(0, τ_ℓ, τ_r) }
  mark_tree(l);
// { t ↦ m: 0, ℓ: l, r: r ∗ tree(l, m(τ_ℓ)) ∗ tree(r, τ_r)
//   ∧ τ = node(0, τ_ℓ, τ_r) }
  mark_tree(r);
// { t ↦ m: 0, ℓ: l, r: r ∗ tree(l, m(τ_ℓ)) ∗ tree(r, m(τ_r))
//   ∧ τ = node(0, τ_ℓ, τ_r) }
  t->m = 1;

//

} // {tree(t, m(τ))}
```



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

void mark_tree(struct node *t) { // {tree(t, τ)}
  if (!t || t->m) return;
  struct node *l = t->l, *r = t->r;
```

$$// \left\{ \begin{array}{l} \mathtt{t} \mapsto \mathtt{m}: 0, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * \mathsf{tree}(\mathtt{l}, \tau_\ell) * \mathsf{tree}(\mathtt{r}, \tau_r) \\ \wedge\ \tau = \mathit{node}(0, \tau_\ell, \tau_r) \end{array} \right\}$$

```c
  mark_tree(l);
```

$$// \left\{ \begin{array}{l} \mathtt{t} \mapsto \mathtt{m}: 0, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * \mathsf{tree}(\mathtt{l}, m(\tau_\ell)) * \mathsf{tree}(\mathtt{r}, \tau_r) \\ \wedge\ \tau = \mathit{node}(0, \tau_\ell, \tau_r) \end{array} \right\}$$

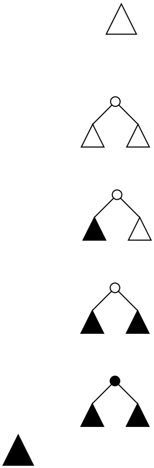
```c
  mark_tree(r);
```

$$// \left\{ \begin{array}{l} \mathtt{t} \mapsto \mathtt{m}: 0, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * \mathsf{tree}(\mathtt{l}, m(\tau_\ell)) * \mathsf{tree}(\mathtt{r}, m(\tau_r)) \\ \wedge\ \tau = \mathit{node}(0, \tau_\ell, \tau_r) \end{array} \right\}$$

```c
  t->m = 1;
```

$$// \left\{ \begin{array}{l} \mathtt{t} \mapsto \mathtt{m}: 1, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * \mathsf{tree}(\mathtt{l}, m(\tau_\ell)) * \mathsf{tree}(\mathtt{r}, m(\tau_r)) \\ \wedge\ \tau = \mathit{node}(0, \tau_\ell, \tau_r) \end{array} \right\}$$

```c
} // {tree(t, m(τ))}
```



Program Proofs without Sharing

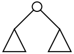
1. Define inductive predicates for recursive data structures
2. Express pre- and post-conditions of the program
3. Apply logic rules to the program

Overlap, Ramification, and DAGs

- DAG predicate:

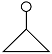
$$\begin{aligned}
&\mathsf{dag}(x,\delta) \stackrel{\text{def}}{=} \\
&(x = 0 \wedge \texttt{emp} \wedge \delta = \emptyset) \\
&\ \vee \exists \ell, r, m, \delta_\ell, \delta_r.\, x \mapsto \ell : \ell, r : r, m : m \ast \\
&\quad (\mathsf{dag}(\ell, \delta_\ell) \mathbin{?} \mathsf{dag}(r, \delta_r)) \wedge \\
&\quad \delta = \mathit{node}(x, m, \delta_\ell, \delta_r)
\end{aligned}$$

- DAG predicate:



$$
\begin{aligned}
&\mathsf{dag}(x,\delta) \stackrel{\text{def}}{=} \\
&(x = 0 \land \mathtt{emp} \land \delta = \emptyset) \\
&\lor \exists \ell, r, m, \delta_\ell, \delta_r.\, x \mapsto \ell : \ell, r : r, m : m \star \\
&\quad (\mathsf{dag}(\ell,\delta_\ell) \star \mathsf{dag}(r,\delta_r)) \land \\
&\quad \delta = \mathit{node}(x, m, \delta_\ell, \delta_r)
\end{aligned}
$$

- With "$\star$": a tree



- DAG predicate:

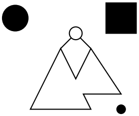
$$
\begin{aligned}
&\mathsf{dag}(x, \delta) \stackrel{\text{def}}{=} \\
&(x = 0 \land \mathtt{emp} \land \delta = \emptyset) \\
&\lor \exists \ell, r, m, \delta_\ell, \delta_r .\, x \mapsto \ell : \ell, r : r, m : m \ast \\
&\quad (\mathsf{dag}(\ell, \delta_\ell) \land \mathsf{dag}(r, \delta_r)) \land \\
&\quad \delta = \mathit{node}(x, m, \delta_\ell, \delta_r)
\end{aligned}
$$

- With "$\ast$": a tree
- With "$\land$":

- DAG predicate:

$$
\begin{aligned}
&\mathsf{dag}(x,\delta) \stackrel{\text{def}}{=} \\
&(x = 0 \wedge \mathtt{emp} \wedge \delta = \emptyset) \\
&\vee \exists \ell, r, m, \delta_\ell, \delta_r.\, x \mapsto \ell : \ell, r : r, m : m \ast \\
&\quad (\mathsf{dag}(\ell, \delta_\ell) \wedge \mathsf{dag}(r, \delta_r)) \wedge \\
&\quad \delta = \mathit{node}(x, m, \delta_\ell, \delta_r)
\end{aligned}
$$

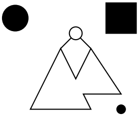
- With "$\ast$": a tree
- With "$\wedge$": a list



- DAG predicate:

$$\mathsf{dag}(x,\delta) \stackrel{\text{def}}{=} (x = 0 \wedge \texttt{emp} \wedge \delta = \varnothing) \vee \exists \ell, r, m, \delta_\ell, \delta_r.\, x \mapsto \ell : \ell, r : r, m : m \star ((\mathsf{dag}(\ell,\delta_\ell) \star \mathsf{true}) \wedge (\mathsf{dag}(r,\delta_r) \star \mathsf{true})) \wedge \delta = node(x, m, \delta_\ell, \delta_r)$$

- With "$\star$": a tree
- With "$\wedge$": a list
- With "$\wedge$" and "$\star$ true": a DAG + anything

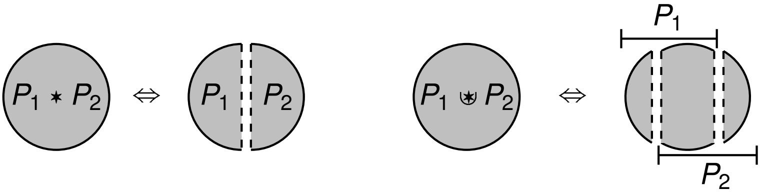


- DAG predicate:

$$\mathsf{dag}(x,\delta) \stackrel{\text{def}}{=} (x = 0 \wedge \mathtt{emp} \wedge \delta = \emptyset) \vee \exists \ell, r, m, \delta_\ell, \delta_r.\, x \mapsto \ell : \ell, r : r, m : m \ast ((\mathsf{dag}(\ell,\delta_\ell) \ast \mathsf{true}) \wedge (\mathsf{dag}(r,\delta_r) \ast \mathsf{true})) \wedge \delta = node(x, m, \delta_\ell, \delta_r)$$

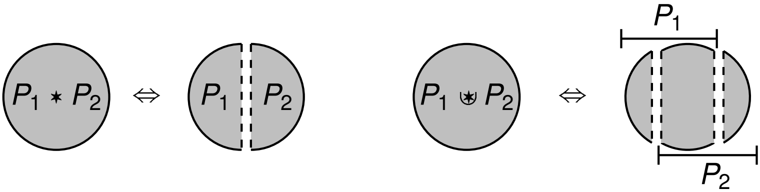
- With "$\ast$": a tree
- With "$\wedge$": a list
- With "$\wedge$" and "$\ast$ true": a DAG + anything
- We need something else…

- Separating vs Overlapping conjunction:



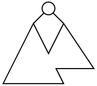
- $\sigma \models P_1 \circledast P_2$ iff $\exists \sigma_1, \sigma_2, \sigma_3.\, \sigma = \sigma_1 \bullet \sigma_2 \bullet \sigma_3$ & $\sigma_1 \bullet \sigma_2 \models P_1$ & $\sigma_2 \bullet \sigma_3 \models P_2$

- Separating vs Overlapping conjunction:



- DAG predicate:

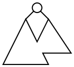
$$\begin{aligned}
&\mathsf{dag}(x,\delta) \stackrel{\text{def}}{=} \\
&(x = 0 \wedge \texttt{emp} \wedge \delta = \emptyset) \\
&\vee \exists \ell, r, m, \delta_\ell, \delta_r . \\
&\quad x \mapsto \ell : \ell, r : r, m : m * (\mathsf{dag}(\ell,\delta_\ell) \cup\!\!\!\!* \; \mathsf{dag}(r,\delta_r)) \wedge \\
&\quad \delta = node(x, m, \delta_\ell, \delta_r)
\end{aligned}$$



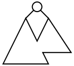
```
struct node {short m; struct node *l,*r;};

void mark_dag(struct node *d) { // {dag(d, δ)}                △
  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(δ))}                                        ▲
```

A Failed Attempt at Framing

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

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

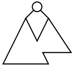


```
struct node {short m; struct node *l,*r;};

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



A Failed Attempt at Framing

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

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



The Ramify Rule of Separation Logic

Ramify

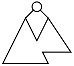
$$\frac{\{P\}\ c\ \{Q\} \qquad \mathit{ramify}(P \leadsto Q, R) = R'}{\{R\}\ c\ \{R'\}}$$

- $\mathit{ramify}(P \leadsto Q, R) = R' \quad \stackrel{\text{def}}{=} \quad R \vdash P \star (Q \mathrel{-\!\!\star} R')$
- $\sigma \models P_1 \mathrel{-\!\!\star} P_2$ iff $\forall \sigma' \models P_1 .\ \sigma \bullet \sigma' \models P_2$

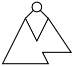
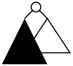
1. Define inductive predicates for recursive data structures
2. Express pre- and post-conditions of the program
3. Apply logic rules to the program
4. **Prove ramification conditions**

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

void mark_dag(struct node *d) { // {dag(d, δ)}
  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(δ))}
```

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

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



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

void mark_dag(struct node *d) { // {dag(d, δ)}
  if (!d || d->m) return;
  struct node *l = d->l, *r = d->r;
// { d ↦ m : 0, ℓ : l, r : r ∗ (dag(l, δ_ℓ) ⊛ dag(r, δ_r))
//   ∧ δ = node(0, δ_ℓ, δ_r) }
  mark_dag(l);
// { d ↦ m : 0, ℓ : l, r : r ∗ (dag(l, m(δ_ℓ)) ⊛ dag(r, δ'_r))
//   ∧ m(δ_r) = m(δ'_r) ∧ δ = node(0, δ_ℓ, δ_r) }
  mark_dag(r);

//

  d->m = 1;

//

} // {dag(d, m(δ))}
```


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

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

// $\left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 0, \ell : \mathtt{l}, \mathtt{r} : \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \delta_\ell) \circledast \mathsf{dag}(\mathtt{r}, \delta_r)) \\ \wedge\ \delta = \mathit{node}(0, \delta_\ell, \delta_r) \end{array} \right\}$

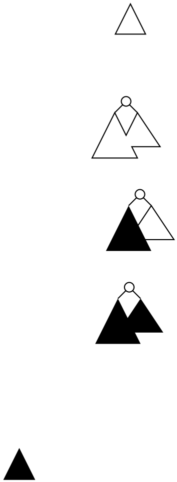
```
  mark_dag(l);
```

// $\left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 0, \ell : \mathtt{l}, \mathtt{r} : \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\delta_\ell)) \circledast \mathsf{dag}(\mathtt{r}, \delta'_r)) \\ \wedge\ m(\delta_r) = m(\delta'_r) \wedge \delta = \mathit{node}(0, \delta_\ell, \delta_r) \end{array} \right\}$

```
  mark_dag(r);
```

// $\left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 0, \ell : \mathtt{l}, \mathtt{r} : \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\delta_\ell)) \circledast \mathsf{dag}(\mathtt{r}, m(\delta'_r))) \\ \wedge\ m(\delta_r) = m(\delta'_r) \wedge \delta = \mathit{node}(0, \delta_\ell, \delta_r) \end{array} \right\}$

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



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

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

$$// \left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 0, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * (\mathsf{dag}(\mathtt{l}, \delta_\ell) \circledast \mathsf{dag}(\mathtt{r}, \delta_r)) \\ \wedge\ \delta = node(0, \delta_\ell, \delta_r) \end{array} \right\}$$

```
  mark_dag(l);
```

$$// \left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 0, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\delta_\ell)) \circledast \mathsf{dag}(\mathtt{r}, \delta'_r)) \\ \wedge\ m(\delta_r) = m(\delta'_r) \wedge \delta = node(0, \delta_\ell, \delta_r) \end{array} \right\}$$

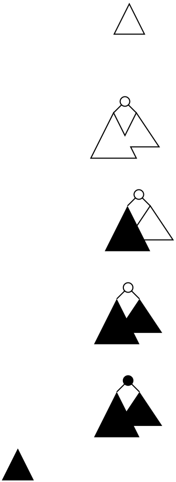
```
  mark_dag(r);
```

$$// \left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 0, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\delta_\ell)) \circledast \mathsf{dag}(\mathtt{r}, m(\delta'_r))) \\ \wedge\ m(\delta_r) = m(\delta'_r) \wedge \delta = node(0, \delta_\ell, \delta_r) \end{array} \right\}$$

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

$$// \left\{ \begin{array}{l} \mathtt{d} \mapsto \mathtt{m}: 1, \ell: \mathtt{l}, \mathtt{r}: \mathtt{r} * (\mathsf{dag}(\mathtt{l}, m(\delta_\ell)) \circledast \mathsf{dag}(\mathtt{r}, m(\delta'_r))) \\ \wedge\ m(\delta_r) = m(\delta'_r) \wedge \delta = node(0, \delta_\ell, \delta_r) \end{array} \right\}$$

```
} // {dag(d, m(δ))}
```

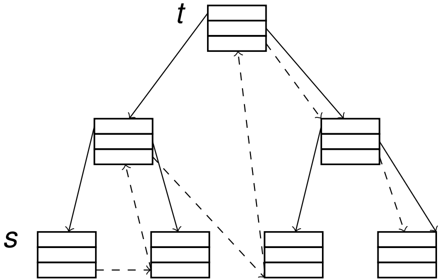


$$
\begin{aligned}
& \mathsf{dag}(\ell, \delta_\ell) \circledast \mathsf{dag}(r, \delta_r) \\
\vdash \quad & \mathsf{dag}(\ell, \delta_\ell) * (\mathsf{dag}(\ell, m(\delta_\ell)) -\!\!* \\
& \qquad \mathsf{dag}(\ell, m(\delta_\ell)) \circledast \mathsf{dag}(r, \delta'_r) \wedge m(\delta_r) = m(\delta'_r))
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
& \mathsf{dag}(\ell, \delta'_\ell) \circledast \mathsf{dag}(r, \delta'_r) \\
\vdash \quad & \mathsf{dag}(r, \delta'_r) * (\mathsf{dag}(r, m(\delta'_r)) -\!\!* \\
& \qquad \mathsf{dag}(\ell, \delta''_\ell) \circledast \mathsf{dag}(r, m(\delta'_r)) \wedge m(\delta'_\ell) = m(\delta''_\ell))
\end{aligned}
\tag{2}
$$

Overlaid Data Structures

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



Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
//
  s = s->next;
//
  t = tree_remove(t,c);
//
  return c;
} // {(list(s)) ∧ tree(t)) * ret ↦ –, –, –}
```

With

$$\{\mathrm{tree}(t)\}\ \text{tree\_remove(t,c)}\ \{\mathrm{tree}(\mathrm{ret}) * c \mapsto -, -, -\}$$

Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
//
  t = tree_remove(t,c);
//
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\mathsf{tree}(t)\}\ \text{tree\_remove(t,c)}\ \{\mathsf{tree}(\mathsf{ret}) \ast c \mapsto -,-,-\}$$

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
  t = tree_remove(t,c);
//
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\text{tree}(t)\}\ \text{tree\_remove(t,c)}\ \{\text{tree}(\text{ret}) * c \mapsto -, -, -\}$$

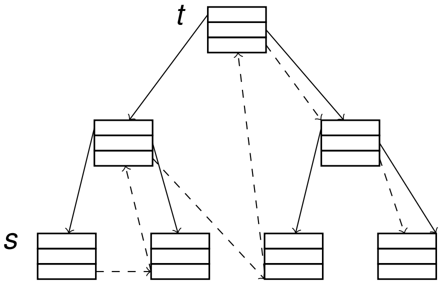
Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
  t = tree_remove(t,c);
// {??? ∧ (tree(t) ∗ c ↦ −, −, −)}
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

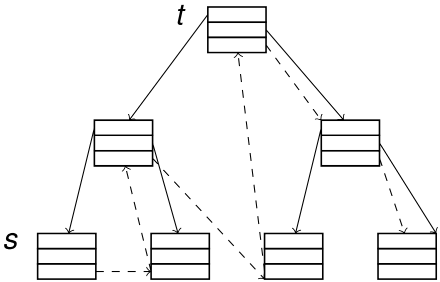
With

$$\{\mathsf{tree}(t)\}\ \mathsf{tree\_remove(t,c)}\ \{\mathsf{tree}(\mathsf{ret}) * c \mapsto -, -, -\}$$

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

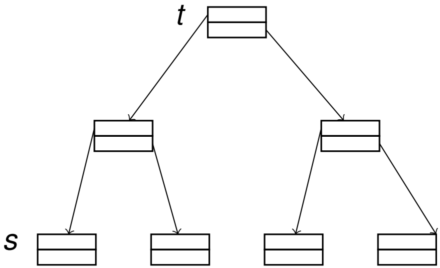


tree(t)



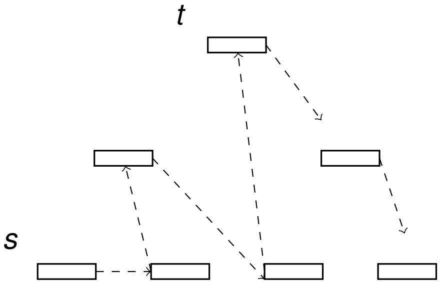
$$\text{tree}(x) \stackrel{\text{def}}{=} (x = 0 \land \text{emp}) \lor \exists L, R, N.\ x \mapsto L, R, N \star \text{tree}(L) \star \text{tree}(R)$$

sktree(t)



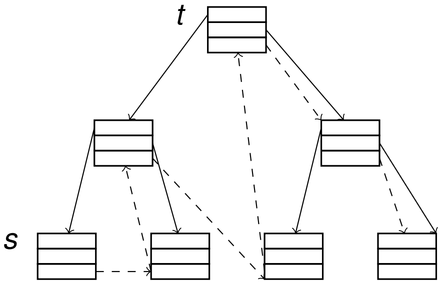
$$\text{sktree}(x) \stackrel{\text{def}}{=} (x = 0 \wedge \text{emp}) \vee \exists L, R, N.\ x \mapsto L, R \star \text{sktree}(L) \star \text{sktree}(R)$$

sklist(s)



$$\mathrm{sklist}(x) \stackrel{\mathrm{def}}{=} (x = 0 \wedge \mathrm{emp}) \vee \exists N.\, x + 2 \mapsto N \star \mathrm{sklist}(N)$$

$\mathsf{tree}(t) \Leftrightarrow \mathsf{sktree}(t, \pi) \star \mathsf{sklist}(t, \pi)$



Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
//
  t = tree_remove(t,c);
//
//
//
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\mathsf{sktree}(t, \pi \uplus \{c\})\}\ \text{tree\_remove(t,c)}\ \{\mathsf{sktree}(\mathrm{ret}, \pi) * c \mapsto -, -\}$$

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ (sktree(t, π ⊎ {c}) ∗ sklist(c, π ⊎ {c}))}
  t = tree_remove(t,c);
//
//
//
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\mathsf{sktree}(t, \pi \uplus \{c\})\}\ \text{tree\_remove(t,c)}\ \{\mathsf{sktree}(\mathsf{ret}, \pi) * c \mapsto -, -\}$$

Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ (sktree(t, π ⊎ {c}) ∗ sklist(c, π ⊎ {c}))}
  t = tree_remove(t,c);
// {(c ↦ −, −, s ∗ list(s)) ∧ (sktree(t, π) ∗ c ↦ −, − ∗ sklist(s, π ⊎ {c}))}
//
//
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\mathsf{sktree}(t, \pi \uplus \{c\})\}\ \text{tree\_remove(t,c)}\ \{\mathsf{sktree}(\text{ret}, \pi) * c \mapsto -, -\}$$

Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
//  {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
//  {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
//  {(c ↦ ℓ, r, s ∗ list(s)) ∧ (sktree(t, π ⊎ {c}) ∗ sklist(c, π ⊎ {c}))}
  t = tree_remove(t,c);
//  {(c ↦ −, −, s ∗ list(s)) ∧ (sktree(t, π) ∗ c ↦ −, − ∗ sklist(s, π ⊎ {c}))}
//  {(c ↦ −, −, s ∗ list(s)) ∧ (sktree(t, π) ∗ c ↦ −, − ∗ sklist(s, π) ∗ c + 2 ↦ −)}
//
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\mathsf{sktree}(t, \pi \uplus \{c\})\}\ \text{tree\_remove(t,c)}\ \{\mathsf{sktree}(\mathrm{ret}, \pi) * c \mapsto -, -\}$$

Removal from a Threaded Tree

```
struct node { struct node *l,*r;
              struct node *next; };
struct node * pop(void) { // {list(s) ∧ tree(t)}
  if (!s) return 0;
  struct node * c = s;
// {(∃n. s ↦ ℓ, r, n ∧ s = c ∗ list(n)) ∧ tree(t)}
  s = s->next;
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ tree(t)}
// {(c ↦ ℓ, r, s ∗ list(s)) ∧ (sktree(t, π ⊎ {c}) ∗ sklist(c, π ⊎ {c}))}
  t = tree_remove(t,c);
// {(c ↦ −, −, s ∗ list(s)) ∧ (sktree(t, π) ∗ c ↦ −, − ∗ sklist(s, π ⊎ {c}))}
// {(c ↦ −, −, s ∗ list(s)) ∧ (sktree(t, π) ∗ c ↦ −, − ∗ sklist(s, π) ∗ c + 2 ↦ −)}
// {(c ↦ −, −, − ∗ list(s)) ∧ (tree(t) ∗ c ↦ −, −, −)}
  return c;
} // {(list(s)) ∧ tree(t)) ∗ ret ↦ −, −, −}
```

With

$$\{\mathsf{sktree}(t, \pi \uplus \{c\})\}\ \text{tree\_remove(t,c)}\ \{\mathsf{sktree}(\mathsf{ret}, \pi) * c \mapsto -, -\}$$

$$
\begin{aligned}
& (c \mapsto \ell, r, n * \mathsf{list}(s)) \wedge (\mathsf{sktree}(t, \pi \uplus \{c\}) * \mathsf{ptrs}(\pi \uplus \{c\})) \\
\vdash \quad & \mathsf{sktree}(t, \pi \uplus \{c\}) * (\mathsf{sktree}(t', \pi) * c \mapsto -, - \mathrel{-\!\!*} \\
& \quad (c \mapsto -, -, n * \mathsf{list}(s)) \wedge (\mathsf{sktree}(t', \pi) * c \mapsto -, - * \mathsf{ptrs}(\pi))
\end{aligned}
$$

Towards Tool Support

Program Proofs with Ramification

- Meta-theory validated in Coq
- Programs proved by hand
- Ramification conditions proved in Coq (work in progress)

Collection of lemmas to simplify ramification conditions, *e.g.*

- $\forall P, Q, R, R', F$

$$\frac{\begin{array}{c} R \vdash P * (Q \mathrel{-\!*} R') \\ R \vdash P * F * \mathrm{true} \qquad F \mathbin{-\!\circledast} R' \vdash F \mathrel{-\!*} R' \end{array}}{R \vdash P * F * (Q * F \mathrel{-\!*} R')}$$

- $\forall P, Q, R, R', F$

$$\frac{\mathit{precise}(P) \qquad \mathit{precise}(Q) \qquad P \mathbin{\cup\!\!\!\!*} R \vdash P * (Q \mathrel{-\!*} Q \mathbin{\cup\!\!\!\!*} R')}{(P * F) \mathbin{\cup\!\!\!\!*} R \vdash P * (Q \mathrel{-\!*} (Q * F) \mathbin{\cup\!\!\!\!*} R')}$$

- $\forall P, Q, R, R', F$

$$\frac{P \mathbin{\cup\!\!\!\!*} R \vdash P * (Q \mathrel{-\!*} Q \mathbin{\cup\!\!\!\!*} R')}{P \mathbin{\cup\!\!\!\!*} (R * F) \vdash P * (Q \mathrel{-\!*} Q \mathbin{\cup\!\!\!\!*} (R' * F))}$$

Benchmark: 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_ref((void**)scan);
    copy_ref((void**)(scan + 4));
    scan = scan + 8;
  }
}
```

```c
void copy_ref(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;
  } } }
```

Loop Invariant

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

Loop Invariant

$$\mathit{to} \leqslant \mathit{scan} \leqslant \mathit{free} < \mathit{to} + \mathit{size} \wedge$$
$$\mathit{cheney}(*\mathit{to}, \mathit{scan}, \mathit{free}) \circledast \mathit{cheney}(*\mathit{scan}, \mathit{scan}, \mathit{free})$$

In-Copy Graph Predicate

$$\mathrm{cheney}(g, \mathit{scan}, \mathit{free}) \stackrel{\mathrm{def}}{=}$$
$$(g = 0 \wedge \mathtt{emp}) \vee (g \mapsto a, b \wedge$$
$$(\mathit{to} \leqslant g \leqslant \mathit{scan} \Rightarrow \mathit{to} \leqslant a, b \leqslant \mathit{to} + \mathit{size}) \wedge$$
$$(\mathit{scan} \leqslant g \leqslant \mathit{free} \Rightarrow \mathit{from} \leqslant a, b \leqslant \mathit{from} + \mathit{size}))$$
$$\circledast \mathrm{cheney}(a, \mathit{scan}, \mathit{free}) \circledast \mathrm{cheney}(b, \mathit{scan}, \mathit{free})$$

Conclusion

Ramify Rule

- Small and intricate programs with sharing
- Exposes the essence of the proofs
- Concise and compositional proofs
- Valid in any separation logic

Ramification Conditions $R \vdash P \star (Q \mathrel{-\!\!\star} R')$

- Beyond the reach of today's automatic theorem provers
- Simplification lemmas provided for Coq
- Expressed as SL entailments
- $\circledast$ is a useful connective**!**

Prospects for Automation

Current Tools

- Automatic shape analysis tools cannot deal with sharing
- Have separation baked in

Automatic Proofs of Programs with Sharing

- Extend classic shape domains to express sharing
- Automate checks of ramification conditions
- More proved programs to come**!**

The Ramify Rule of Separation Logic

Compositional Reasoning for Sharing

Jules Villard[1]

Joint work with Aquinas Hobor[2]

[1]University College London
[2]National University of Singapore