# Steel:
# Scaling up verification in $\mathbf{F}^\star$

AYMERIC FROMHERZ, DENIS MERIGOUX

# Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

# Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

- Verified low-level programming is now viable, but requires lots of effort

# Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

- Verified low-level programming is now viable, but requires lots of effort

EVERCRYPT

everparse

# Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

- Verified low-level programming is now viable, but requires lots of effort

- F*: Classical Hoare logic and select/update reasoning. How to scale?

EVERCRYPT

everparse

# Type-based Ownership to the Rescue?

# Type-based Ownership to the Rescue?

The Rust example:

- Memory safety
- Data-race freedom

By virtue of typing!

# Type-based Ownership to the Rescue?

The Rust example:

- Memory safety
- Data-race freedom

By virtue of typing!

What about verification?

- Rust programs aren't proven correct
- Rust programs have unsafe blocks

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*

- Targets general-purpose concurrent, systems programming

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F*'s logic

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F*'s logic
- Extensible with new constructs, expressed as verified libraries

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into $F^\star$

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within $F^\star$'s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into $F^\star$

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within $F^\star$'s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:

1. Separated resources and framing for interference control

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into $\mathrm{F}^\star$

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within $\mathrm{F}^\star$'s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:

1. Separated resources and framing for interference control
2. Permissions: Exclusive mutable or shared immutable access to resources

# Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into $F^\star$

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within $F^\star$'s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:

1. Separated resources and framing for interference control
2. Permissions: Exclusive mutable or shared immutable access to resources
3. Fork/join concurrency with locks

# Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

# Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

# Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

We can already write Steel programs, but…

# Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

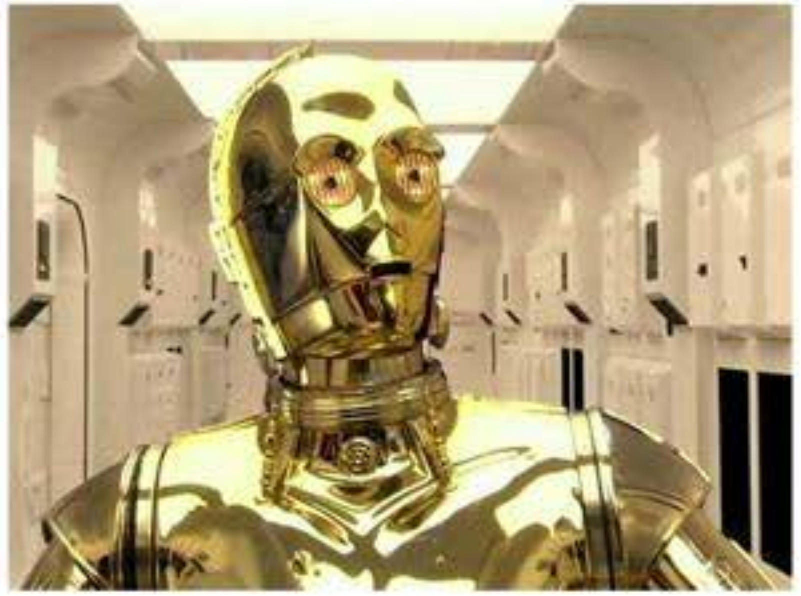
We can already write Steel programs, but…

# Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

We can already write Steel programs, but…



# Main idea 1:
# Resources

# Main idea 1:
# Resources

1. Footprint

# Main idea 1: Resources

1. Footprint

2. Invariant

# Main idea 1: Resources

1. Footprint

2. Invariant

```
emp: resource
```

# Main idea 1: Resources

1. Footprint — `emp: resource`

2. Invariant — `ptr` $p$ `arr` $b$

# Main idea 1: Resources

1. Footprint

`emp: resource`

2. Invariant

`ptr` $p$ `arr` $b$

`ptr` $p \mapsto v$ `arr` $b \mapsto [> 0; \_]$

# Main idea 1: Resources

1. Footprint — `emp: resource`

2. Invariant — `ptr` $p$     `arr` $b$

3. View — `ptr` $p \mapsto v$     `arr` $b \mapsto [> 0; \_\,]$

# Resources : separation

# Resources : separation

$$R_1 \star R_2$$

# Resources : separation

$$\boxed{R_1 \star R_2}$$

$$(\mathtt{ptr}\ p_1 \mapsto v_1) \star (\mathtt{ptr}\ p_2 \mapsto v_2)$$

# Resources : separation

$$R_1 \star R_2$$

$(\mathtt{ptr}\ p_1 \mapsto v_1) \star (\mathtt{ptr}\ p_2 \mapsto v_2)$

$\mathtt{slist}\ x \mapsto \ell =$

# Resources : separation

$$\boxed{R_1 \star R_2}$$

$$(\mathtt{ptr}\ p_1 \mapsto v_1) \star (\mathtt{ptr}\ p_2 \mapsto v_2)$$

$$\begin{aligned} \mathtt{slist}\, x \mapsto \ell = \ & \mathtt{match}\ \ell\ \mathtt{with} \\ & \mid\ [\,] \to \\ & \mid\ \mathrm{hd} :: \mathrm{tl} \to \end{aligned}$$

# Resources : separation

$$R_1 \star R_2$$

$$(\texttt{ptr}\ p_1 \mapsto v_1) \star (\texttt{ptr}\ p_2 \mapsto v_2)$$

$$\begin{aligned} \texttt{slist}\ x \mapsto \ell = \texttt{match}\ \ell\ \texttt{with} \\ \mid\ [\,] \to\ \texttt{emp} \\ \mid\ \text{hd} :: \text{tl} \to \end{aligned}$$

# Resources : separation

$$R_1 \star R_2$$

$$(\texttt{ptr}\ p_1 \mapsto v_1) \star (\texttt{ptr}\ p_2 \mapsto v_2)$$

$$\begin{aligned}\texttt{slist}\ x \mapsto \ell = \ & \texttt{match}\ \ell\ \texttt{with} \\ & \mid [\,] \rightarrow \texttt{emp} \\ & \mid \mathrm{hd} :: \mathrm{tl} \rightarrow (\texttt{ptr}\ x \mapsto \mathrm{hd}) \star (\texttt{slist}\ \mathrm{hd.next} \mapsto \mathrm{tl})\end{aligned}$$

# Resources: typing

# Resources: typing

Computations on resources :

`RST` $(\alpha$: `Type`)

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

# Resources: typing

<u>Allocating a pointer:</u>

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
      (provides (λ p → (ptr p ↦ v)))
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
      (provides (λ p → (ptr p ↦ v)))
```

Updating a pointer:

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
      (provides (λ p → (ptr p ↦ v)))
```

Updating a pointer:

```
val (:=) : (p: pointer α) → (v: α) →
  RST unit
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
      (provides (λ p → (ptr p ↦ v)))
```

Updating a pointer:

```
val (:=) : (p: pointer α) → (v: α) →
  RST unit
      (expects (ptr p))
```

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
      (provides (λ p → (ptr p ↦ v)))
```

Updating a pointer:

```
val (:=) : (p: pointer α) → (v: α) →
  RST unit
      (expects (ptr p))
      (provides (λ _ → (ptr p ↦ v)))
```

# Main idea 1: Resources

1. Footprint — `emp: resource`

2. Invariant — `ptr` $p$ — `arr` $b$

3. View — `ptr` $p \mapsto v$ — `arr` $b \mapsto [> 0; \_\,]$

# Resources: typing

Computations on resources :

```
RST (α: Type)
    (expects resource)
    (provides (α → resource))
```

Allocating a pointer:

```
val alloc : (v: α) →
  RST (pointer α)
      (expects emp)
      (provides (λ p → (ptr p ↦ v)))
```

Updating a pointer:

```
val (:=) : (p: pointer α) → (v: α) →
  RST unit
      (expects (ptr p))
      (provides (λ _ → (ptr p ↦ v)))
```

# Resources: view specification

# Resources: view specification

Separation-logic style specification:

# Resources: view specification

Separation-logic style specification:

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
```

# Resources: view specification

<u>Separation-logic style specification:</u>

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

# Resources: view specification

Separation-logic style specification:

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

More stateful specification:

# Resources: view specification

Separation-logic style specification:

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

More stateful specification:

```
val inc : (p: pointer int) →
  RST unit
```

# Resources: view specification

<u>Separation-logic style specification:</u>

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

<u>More stateful specification:</u>

```
val inc : (p: pointer int) →
  RST unit
      (expects (ptr p))
```

# Resources: view specification

<u>Separation-logic style specification:</u>

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

<u>More stateful specification:</u>

```
val inc : (p: pointer int) →
  RST unit
      (expects (ptr p))
      (provides (λ _ → ptr p))
```

# Resources: view specification

<u>Separation-logic style specification:</u>

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

<u>More stateful specification:</u>

```
val inc : (p: pointer int) →
  RST unit
      (expects (ptr p))
      (provides (λ _ → ptr p))
      (ensures (λ old _ new →
```

# Resources: view specification

<u>Separation-logic style specification:</u>

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
      (expects (ptr p ↦ v))
      (provides (λ _ → (ptr p ↦ v + 1)))
```

<u>More stateful specification:</u>

```
val inc : (p: pointer int) →
  RST unit
      (expects (ptr p))
      (provides (λ _ → ptr p))
      (ensures (λ old _ new →
        new (ptr p) = old (ptr p) + 1
```

# A frame rule for Steel

# Resources: view specification

Separation-logic style specification:

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
    (expects (ptr p ↦ v))
    (provides (λ _ → (ptr p ↦ v + 1)))
```

More stateful specification:

```
val inc : (p: pointer int) →
  RST unit
    (expects (ptr p))
    (provides (λ _ → ptr p))
    (ensures (λ old _ new →
      new (ptr p) = old (ptr p) + 1
    ))
```

# Resources: view specification

Separation-logic style specification:

```
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
    (expects (ptr p ↦ v))
    (provides (λ _ → (ptr p ↦ v + 1)))
```

More stateful specification:

```
val inc : (p: pointer int) →
  RST unit
    (expects (ptr p))
    (provides (λ _ → ptr p))
    (ensures (λ old _ new →
      new (ptr p) = old (ptr p) + 1
    ))
```

# A frame rule for Steel

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

In Steel:

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

In Steel:

```
val frame :
  (P ⋆ Q: resource) →
```

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

In Steel:

```
val frame :
  (P ⋆ Q: resource) →
  (P ⋆ R: resource) →
```

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

In Steel:

```
val frame :
  (P ⋆ Q: resource) →
  (P ⋆ R: resource) →
  (f: unit → RST α (expects Q) (provides R)) →
```

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

In Steel:

```
val frame :
  (P ⋆ Q: resource) →
  (P ⋆ R: resource) →
  (f: unit → RST α (expects Q) (provides R)) →
  RST α
      (expects (P ⋆ Q))
      (provides (P ⋆ R))
```

# Challenge: efficient $\mathbf{F}^{\star}$ embedding

# Challenge: efficient $\mathbf{F}^\star$ embedding

<u>Associative/Commutative rewriting:</u>

# Challenge: efficient $\mathbf{F}^\star$ embedding

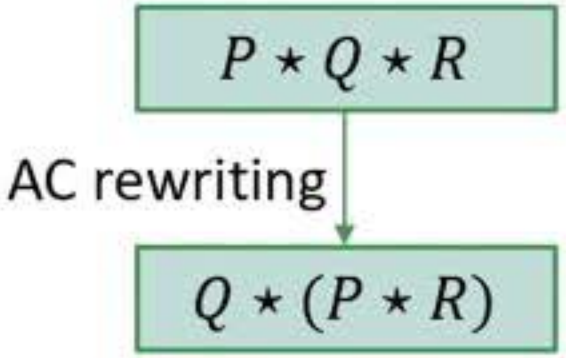
Associative/Commutative rewriting:

$P \star Q \star R$

```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

# Challenge: efficient $\mathbf{F}^\star$ embedding

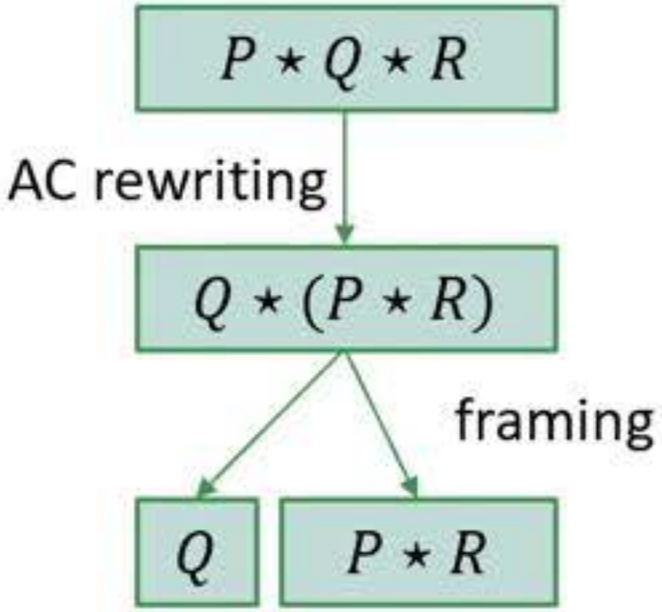
<u>Associative/Commutative rewriting:</u>




```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

# Challenge: efficient $\mathrm{F}^\star$ embedding

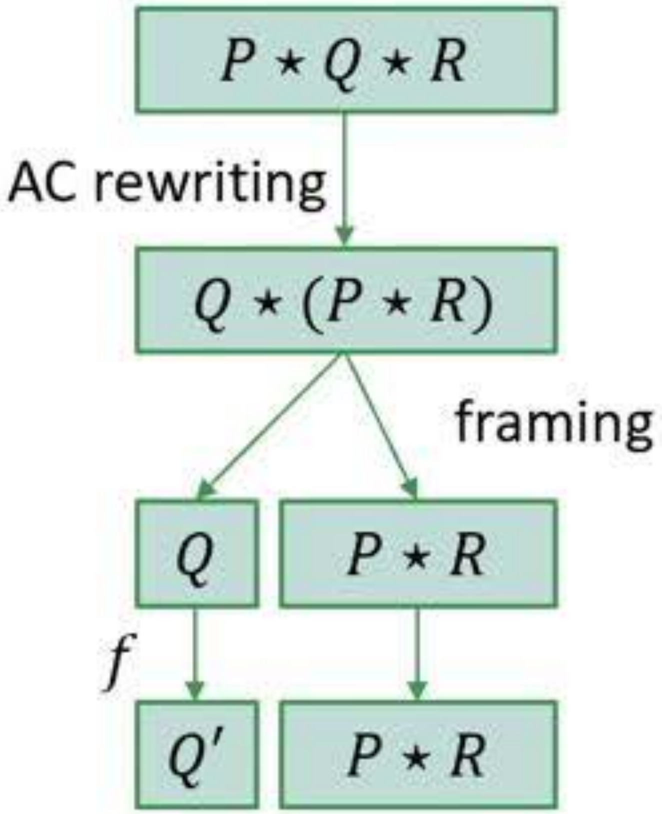
<u>Associative/Commutative rewriting:</u>




```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

# Challenge: efficient $\mathbf{F}^\star$ embedding

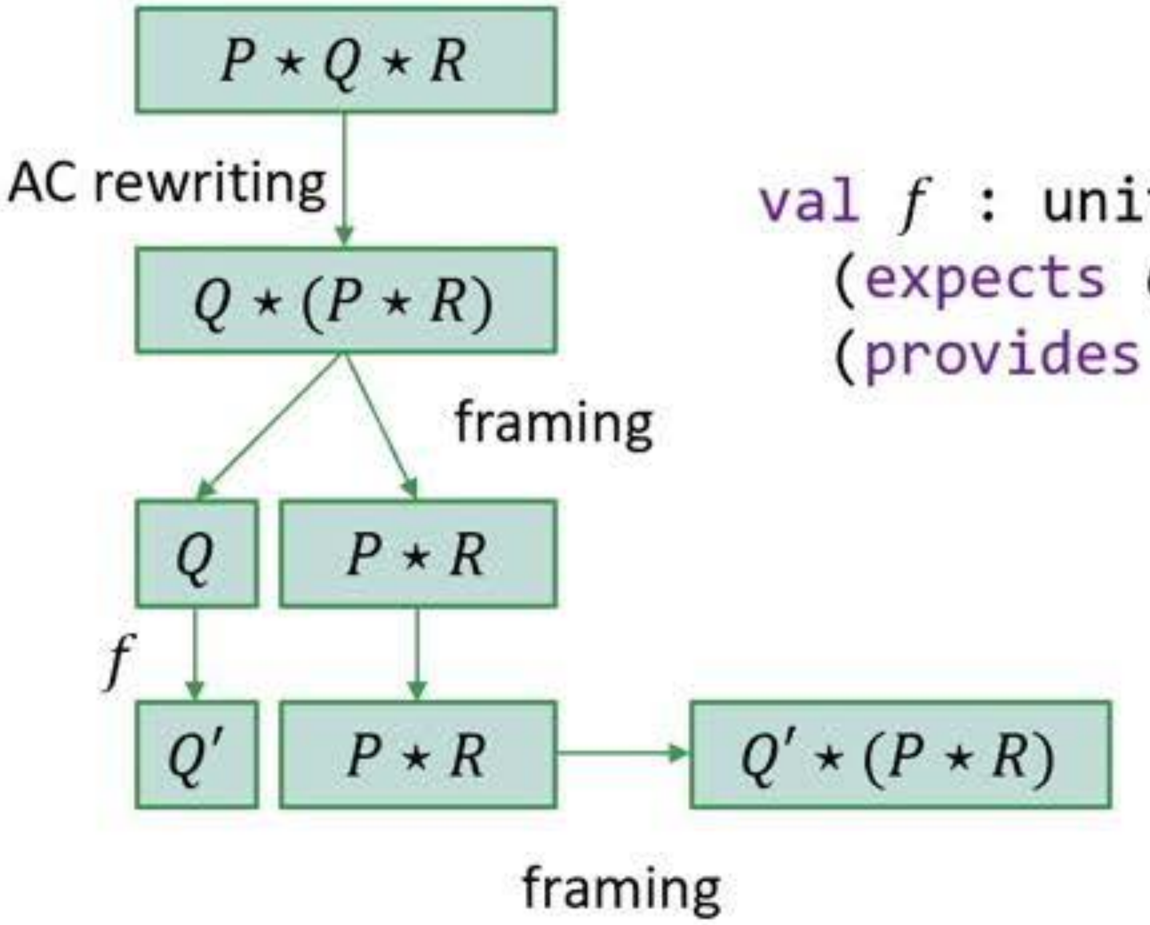
Associative/Commutative rewriting:




```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

# Challenge: efficient $\mathbf{F}^\star$ embedding

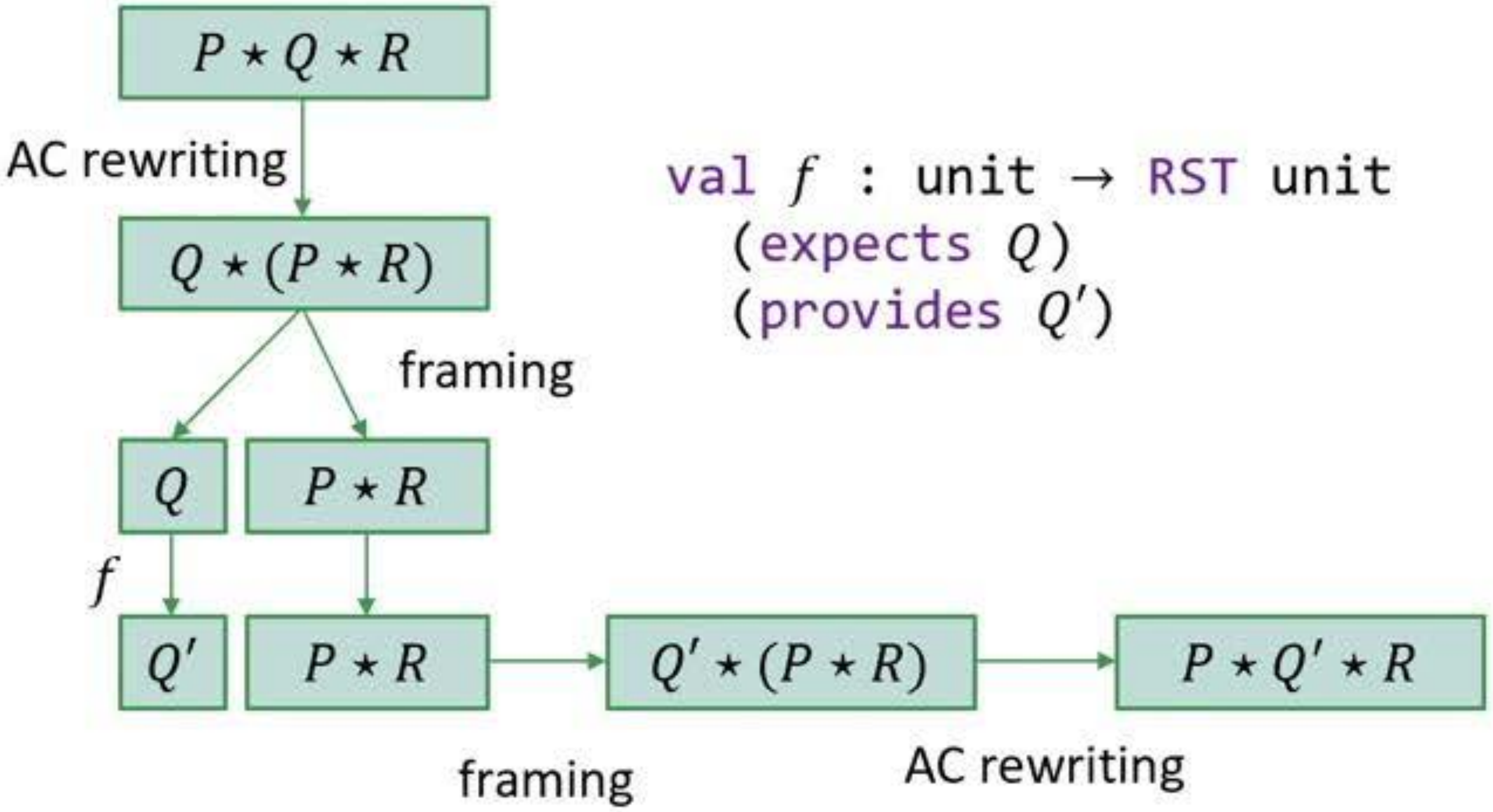
Associative/Commutative rewriting:




```
val f : unit → RST unit
  (expects Q)
  (provides Q')
```

# Challenge: efficient F* embedding

<u>Associative/Commutative rewriting:</u>

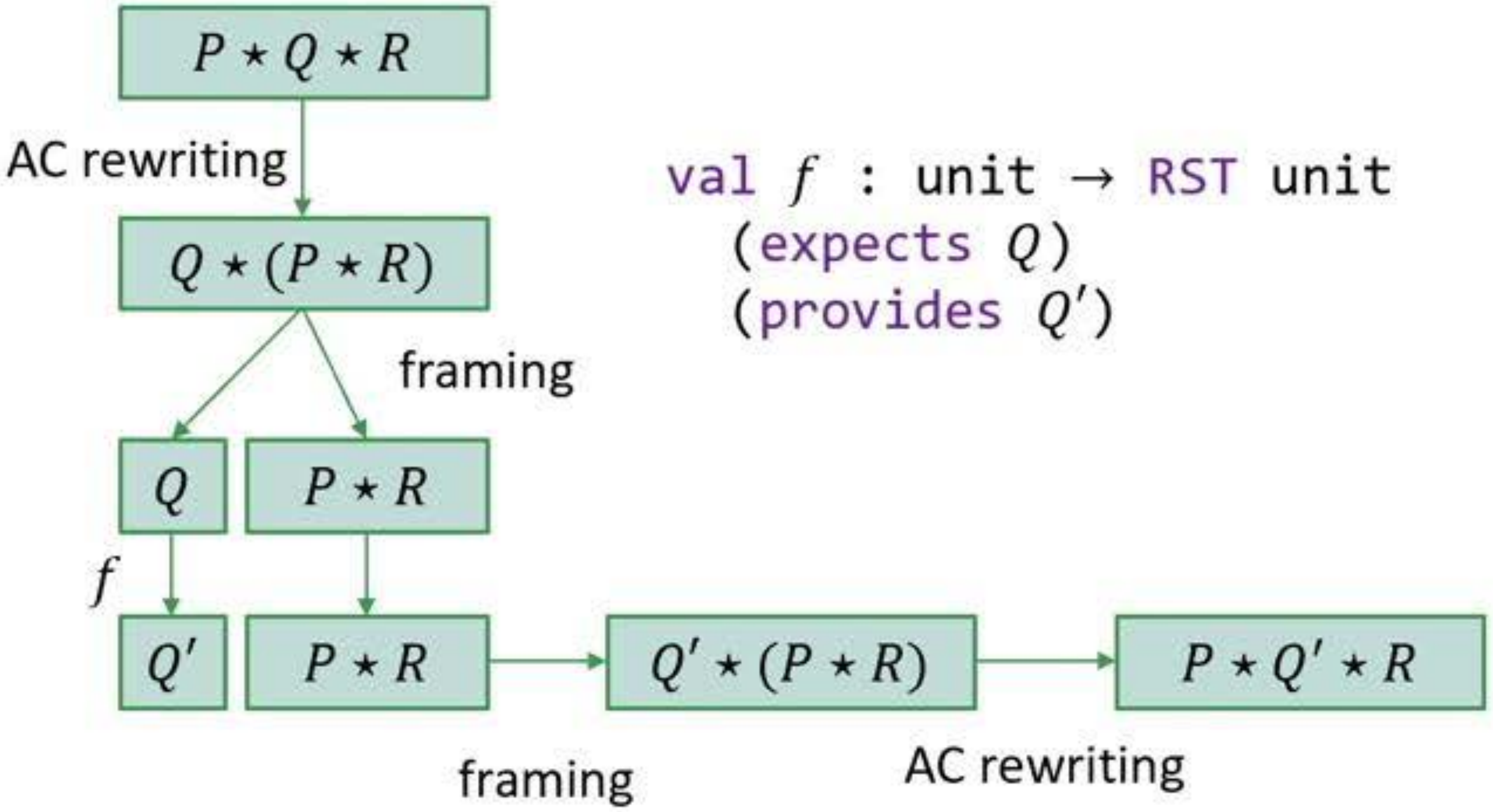



```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

# Challenge: efficient F* embedding

Associative/Commutative rewriting:

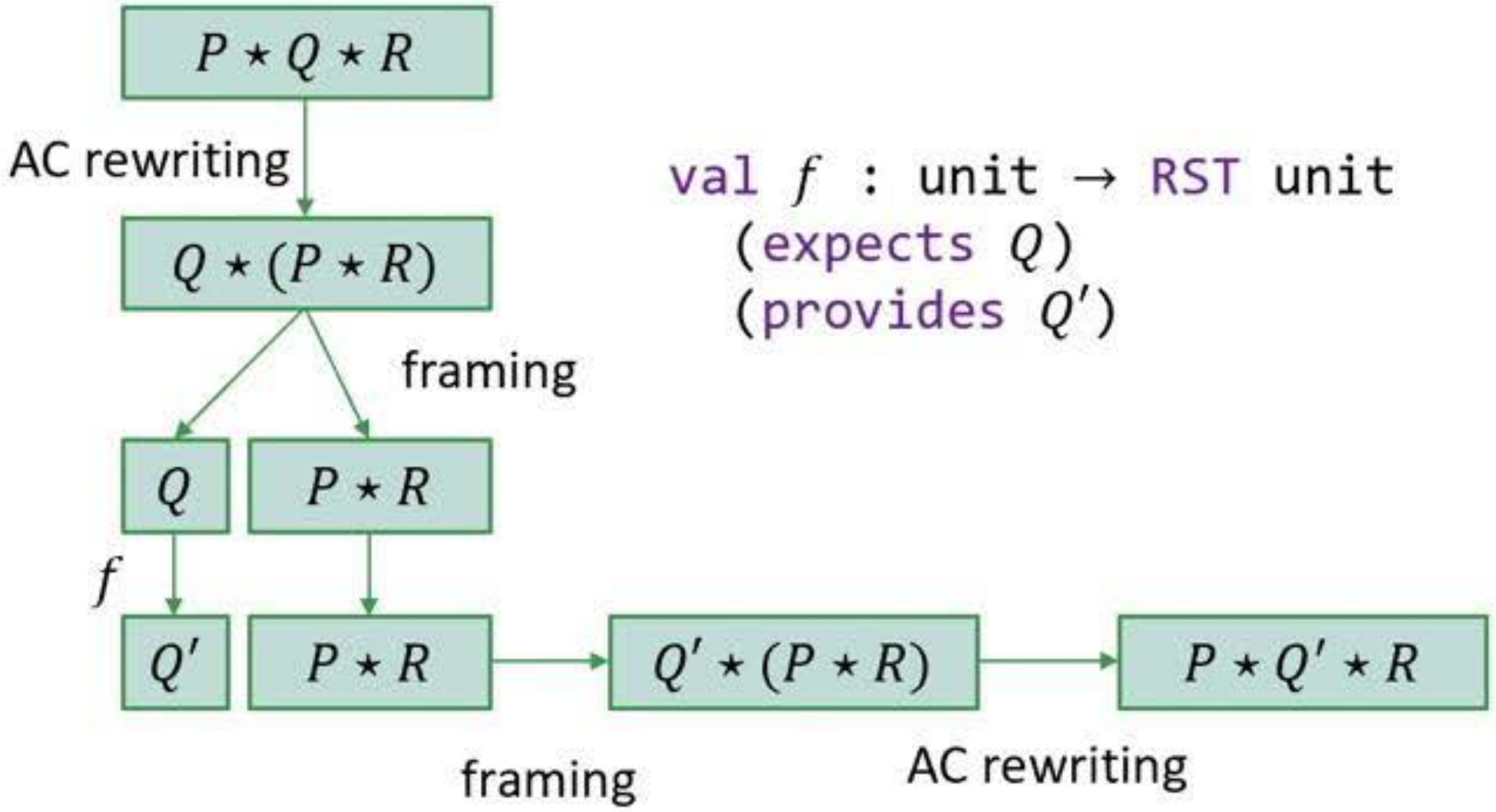
```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```




Going higher order:

# Challenge: efficient $\mathbf{F}^\star$ embedding

<u>Associative/Commutative rewriting:</u>



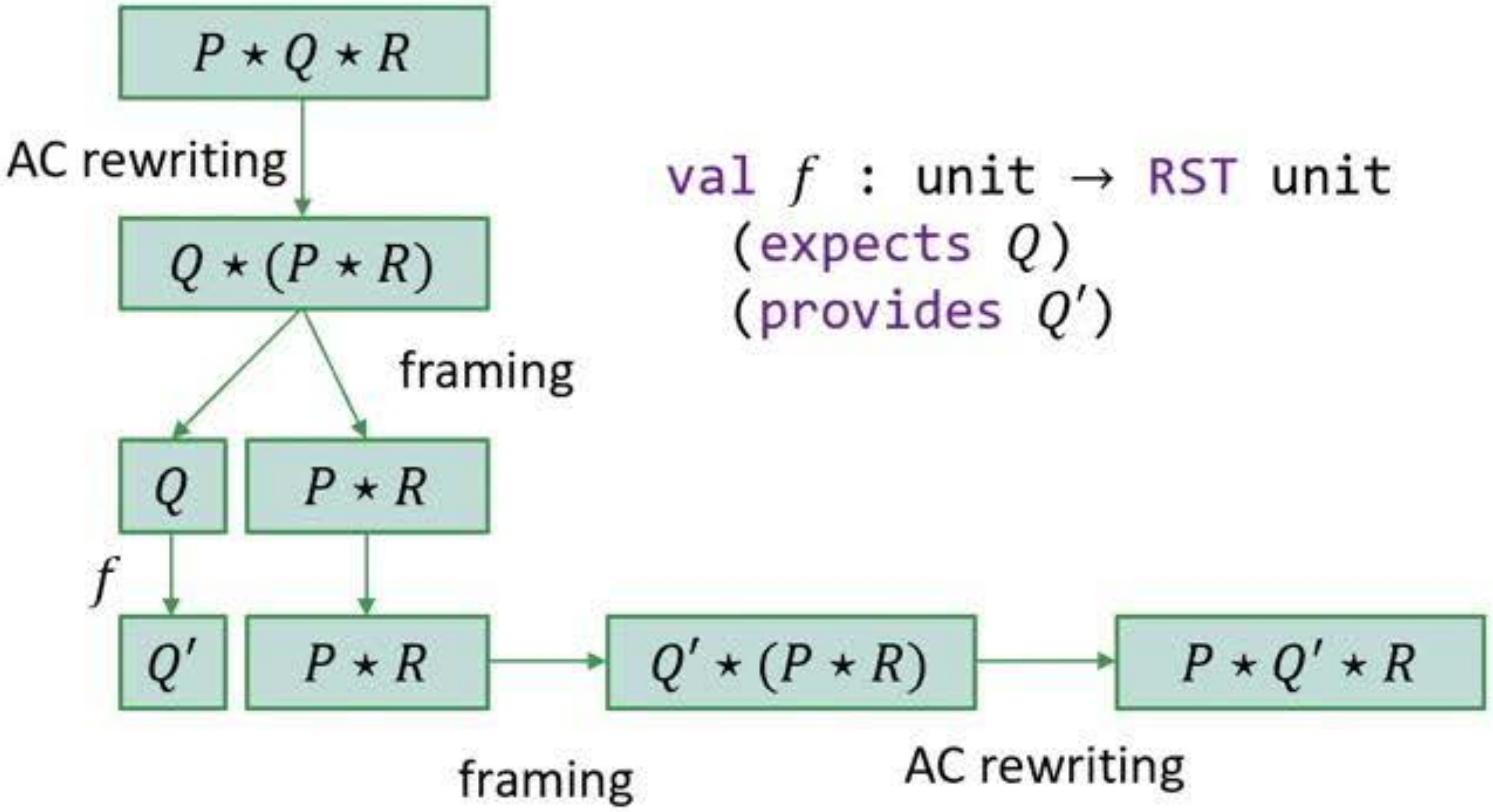

```
val f : unit → RST unit
  (expects Q)
  (provides Q')
```

<u>Going higher order:</u>

- Current heap model: only first-order logic

# Challenge: efficient F$^\star$ embedding

<u>Associative/Commutative rewriting:</u>



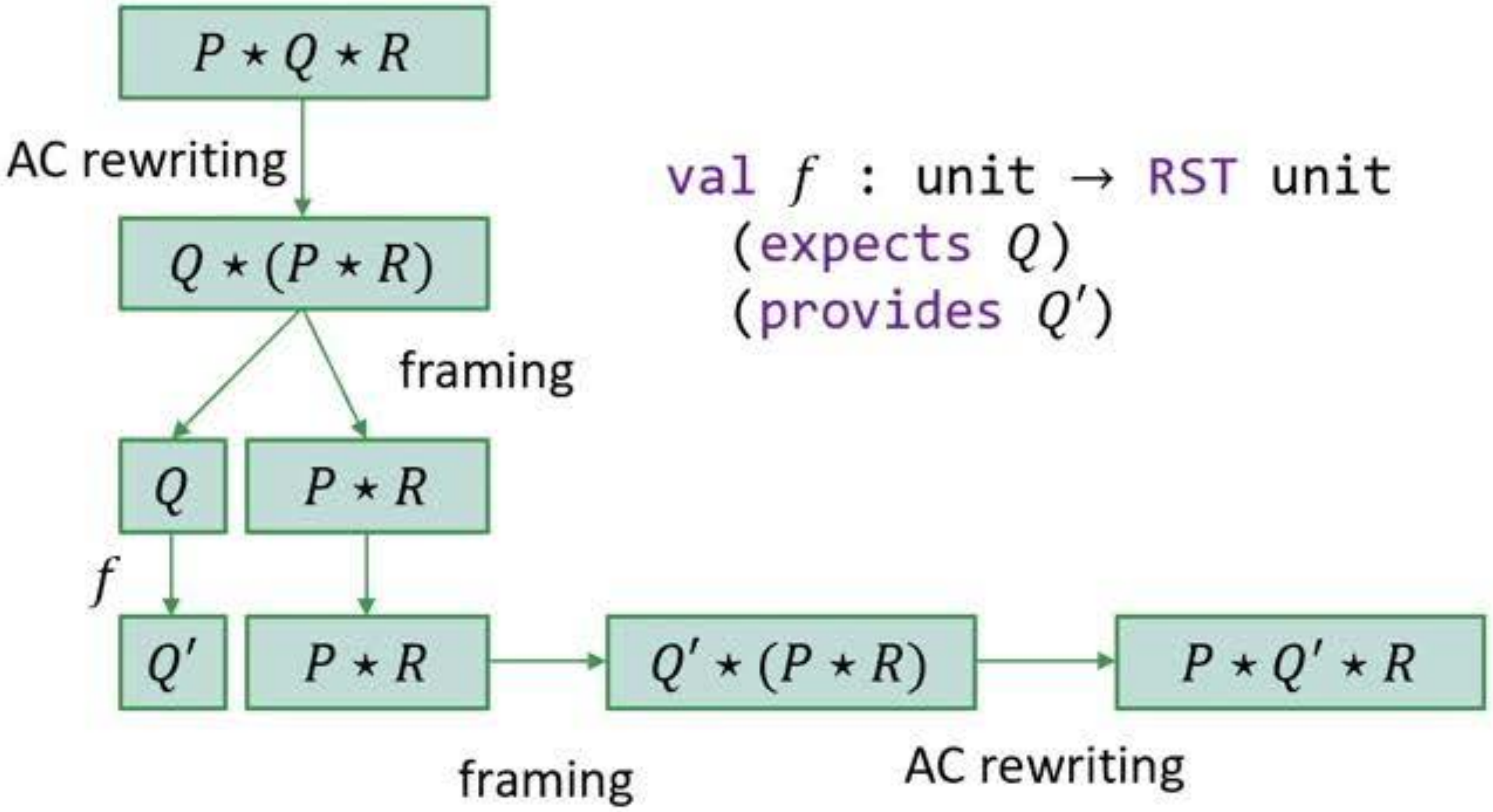

```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

<u>Going higher order:</u>

- Current heap model: only first-order logic
- `resource ≈ heap → prop`

# Challenge: efficient $\mathbf{F}^\star$ embedding

Associative/Commutative rewriting:




```
val f : unit → RST unit
  (expects Q)
  (provides Q′)
```

Going higher order:

- Current heap model: only first-order logic
- `resource ≈ heap → prop`
- `⋆ : resource → resource → resource` is higher-order

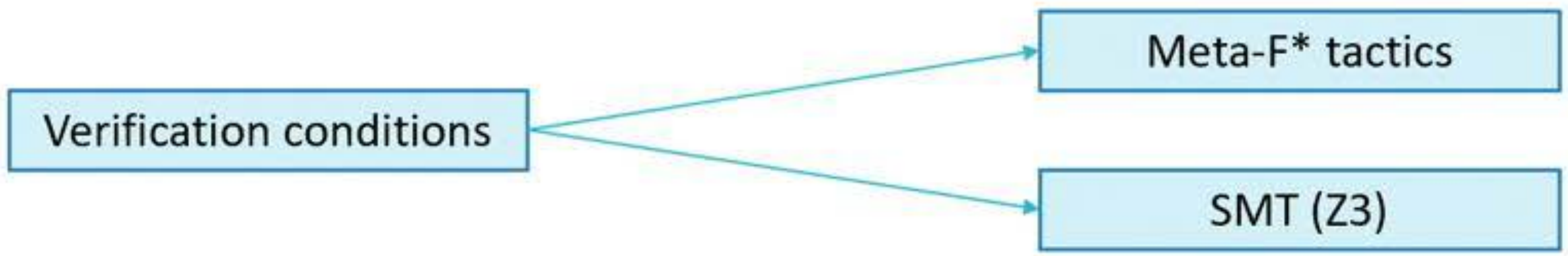
# Mixing tactics and SMT

# Mixing tactics and SMT

Verification conditions

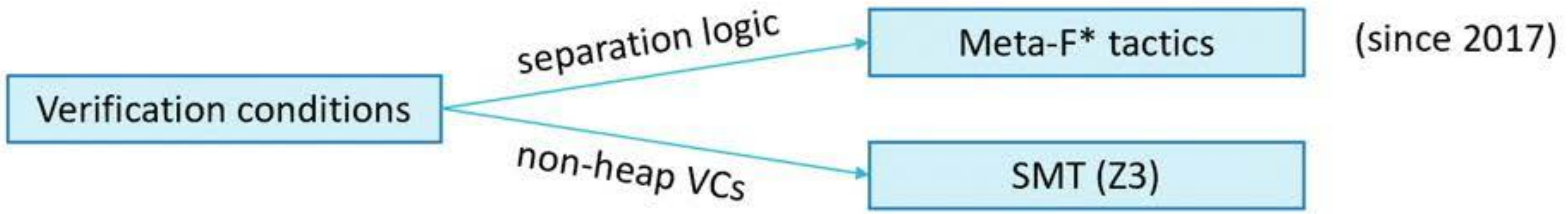
# Mixing tactics and SMT

# Mixing tactics and SMT




(since 2017)

# Mixing tactics and SMT



# Main idea 2:
# Permissions over shared resources

# Main idea 2:
# Permissions over shared resources

Updating requires exclusive ownership:

# Main idea 2:
# Permissions over shared resources

Updating requires exclusive ownership:

```
val (:=) : (p: pointer α) → (v: α) →
  RST (pointer α)
      (expects (ptr p))
      (provides (λ p → (ptr p ↦ v)))
```

# Main idea 2:
# Permissions over shared resources

Updating requires exclusive ownership:

```
val (:=) : (p: pointer α) → (v: α) →
  RST (pointer α)
      (expects (ptr p))
      (provides (λ p → (ptr p ↦ v)))
```

What if we only want to read $p$ ?

# Main idea 2:
# Permissions over shared resources

<u>Updating requires exclusive ownership:</u>

```
val (:=) : (p: pointer α) → (v: α) →
  RST (pointer α)
      (expects (ptr p))
      (provides (λ p → (ptr p ↦ v)))
```

What if we only want to read $p$ ?

```
read_only p:

…
p := 0;
…
```

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

<u>Pointer dereference:</u>

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

```
val (!) : (p: pointer α) →
  RST α
```

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

```
val (!) : (p: pointer α) →
  RST α
      (expects (RO (ptr p)))
```

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

<u>Pointer dereference:</u>

```
val (!) : (p: pointer α) →
  RST α
      (expects (RO (ptr p)))
      (provides (λ v → RO (ptr p ↦ v)))
```

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

```
val (!) : (p: pointer α) →
  RST α
      (expects (RO (ptr p)))
      (provides (λ v → RO (ptr p ↦ v)))
```

Immutable sharing:

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

```
val (!) : (p: pointer α) →
  RST α
      (expects (RO (ptr p)))
      (provides (λ v → RO (ptr p ↦ v)))
```

Immutable sharing:

```
val share : (p: pointer α) →
  RST (pointer α)
```

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

```
val (!) : (p: pointer α) →
  RST α
      (expects (RO (ptr p)))
      (provides (λ v → RO (ptr p ↦ v)))
```

Immutable sharing:

```
val share : (p: pointer α) →
  RST (pointer α)
      (expects (RW (ptr p)))
```

# Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

<u>Pointer dereference:</u>

```
val (!) : (p: pointer α) →
  RST α
      (expects (RO (ptr p)))
      (provides (λ v → RO (ptr p ↦ v)))
```

<u>Immutable sharing:</u>

```
val share : (p: pointer α) →
  RST (pointer α)
      (expects (RW (ptr p)))
      (provides (λ p′ → RO (ptr p) ⋆ RO (ptr p′)))
```

# Managing fractional permissions

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\mathrm{Perm}=f},\ 0 < f < 1$

RW $R = R_{\mathrm{Perm}=1}$

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f},\ 0 < f < 1$

RW $R = R_{\text{Perm}=1}$

Gathering back permissions:

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f},\ 0 < f < 1$

RW $R = R_{\text{Perm}=1}$

Gathering back permissions:

```
val gather : (p: pointer α) → (p′: pointer α) →
  RST unit
```

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f}$, $0 < f < 1$

RW $R = R_{\text{Perm}=1}$

<u>Gathering back permissions:</u>

val gather : ($p$: pointer $\alpha$) → ($p'$: pointer $\alpha$) →
  RST unit
    (expects ((ptr $p$)$_{\text{Perm}=f}$ ⋆ (ptr $p'$)$_{\text{Perm}=f'}$)))

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f}$, $0 < f < 1$

RW $R = R_{\text{Perm}=1}$

<u>Gathering back permissions:</u>

val gather : $(p$: pointer $\alpha) \rightarrow (p'$: pointer $\alpha) \rightarrow$
  RST unit
    (expects ((ptr $p)_{\text{Perm}=f} \star$ (ptr $p')_{\text{Perm}=f'}$)))
    (provides ($\lambda\_ \rightarrow$ (ptr $p)_{\text{Perm}=f+f'}$)))

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f},\ 0 < f < 1$

RW $R = R_{\text{Perm}=1}$

Gathering back permissions:

```
val gather : (p: pointer α) → (p′: pointer α) →
  RST unit
      (expects ((ptr p)_{Perm=f} ⋆ (ptr p′)_{Perm=f′}))
      (provides (λ _ → (ptr p)_{Perm=f+f′}))
      (requires (gatherable p p′))
```

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f}$, $0 < f < 1$

RW $R = R_{\text{Perm}=1}$

Gathering back permissions:

```
val gather : (p: pointer α) → (p′: pointer α) →
  RST unit
      (expects ((ptr p)Perm=f ⋆ (ptr p′)Perm=f′)))
      (provides (λ _ → (ptr p)Perm=f+f′)))
      (requires (gatherable p p′))
```

- Within TCB and memory model

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f}$, $0 < f < 1$

RW $R = R_{\text{Perm}=1}$

- Within TCB and memory model
- Statically checked with SMT

Gathering back permissions:

```
val gather : (p: pointer α) → (p′: pointer α) →
  RST unit
      (expects ((ptr p)_{Perm=f} ⋆ (ptr p′)_{Perm=f′}))
      (provides (λ _ → (ptr p)_{Perm=f+f′}))
      (requires (gatherable p p′))
```

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f}$, $0 < f < 1$

RW $R = R_{\text{Perm}=1}$

Gathering back permissions:

```
val gather : (p: pointer α) → (p′: pointer α) →
  RST unit
      (expects ((ptr p)_{Perm=f} ⋆ (ptr p′)_{Perm=f′}))
      (provides (λ _ → (ptr p)_{Perm=f+f′}))
      (requires (gatherable p p′))
```

- Within TCB and memory model
- Statically checked with SMT
- Users can define scoped sharing, etc.

# Main idea 3: Concurrency in Steel

Currently in scope:

- Data-race freedom
- Sequential consistency
- Scoped fork-join model (par combinator)
- Mutable memory shared through locks

# Managing fractional permissions

Permissions are fractions:

RO $R = R_{\text{Perm}=f}$, $0 < f < 1$

RW $R = R_{\text{Perm}=1}$

Gathering back permissions:

$\texttt{val gather : } (p\texttt{: pointer } \alpha) \rightarrow (p'\texttt{: pointer } \alpha) \rightarrow$
$\quad \texttt{RST unit}$
$\quad\quad (\texttt{expects } ((\texttt{ptr } p)_{\text{Perm}=f} \star (\texttt{ptr } p')_{\text{Perm}=f'}))$
$\quad\quad (\texttt{provides } (\lambda\_ \rightarrow (\texttt{ptr } p)_{\text{Perm}=f+f'}))$
$\quad\quad (\texttt{requires } (\texttt{gatherable } p\ p'))$

- Within TCB and memory model
- Statically checked with SMT
- Users can define scoped sharing, etc.

# Main idea 3: Concurrency in Steel

Currently in scope:

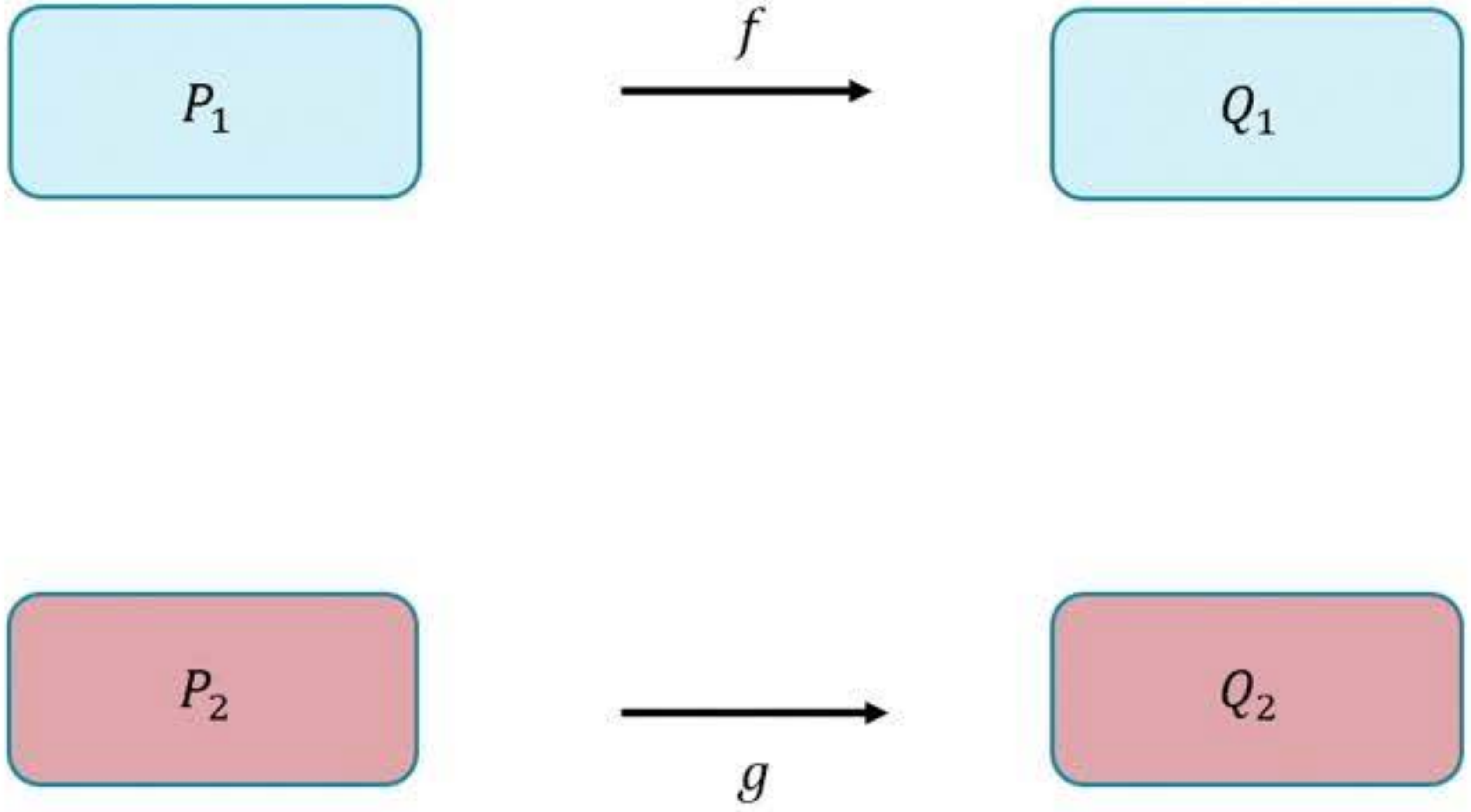
- Data-race freedom
- Sequential consistency
- Scoped fork-join model (par combinator)
- Mutable memory shared through locks

# The “par” combinator

# The "par" combinator

Concurrent separation logic:

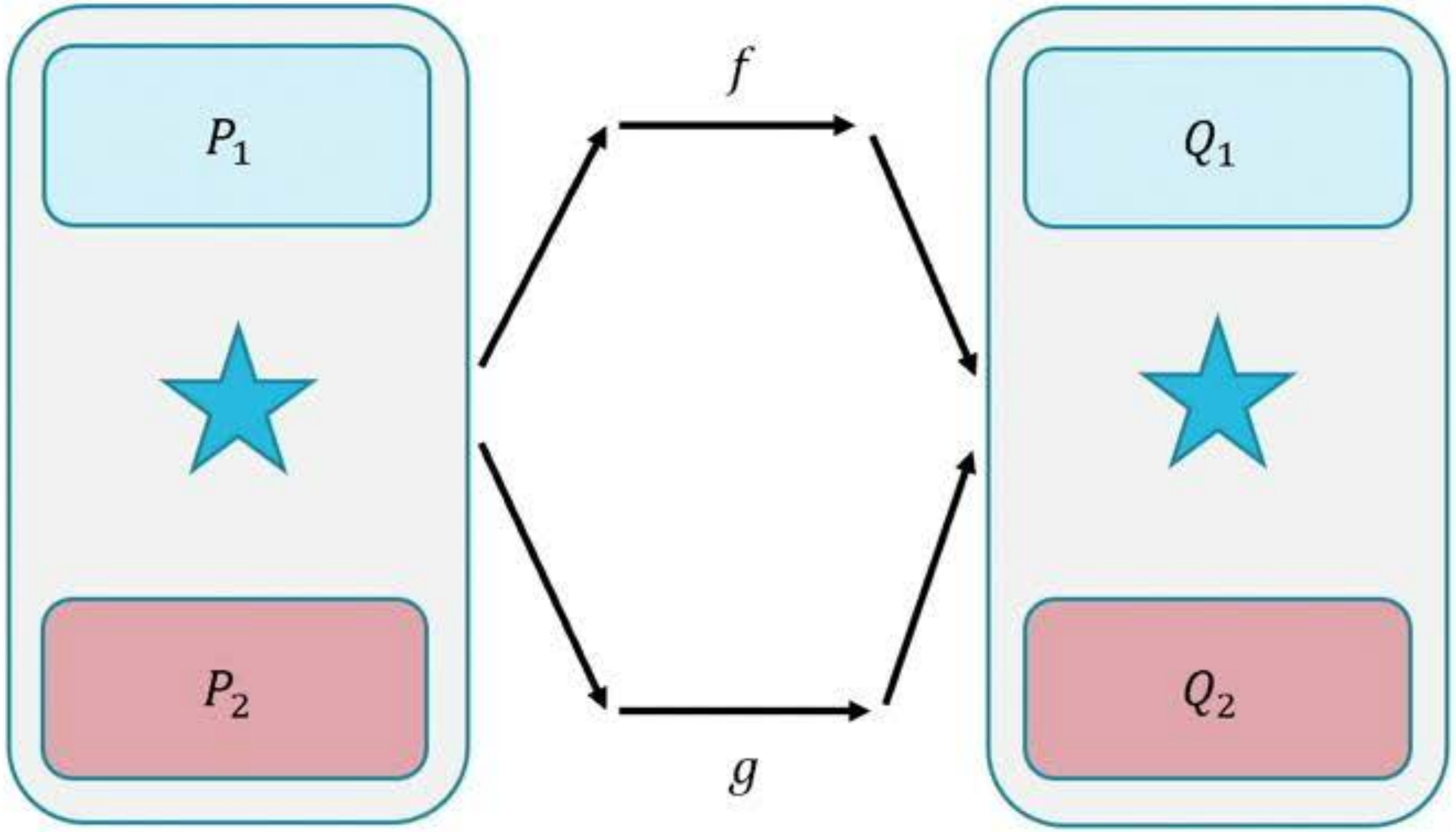
$$\frac{\{P_1\}\, f\, \{Q_1\} \qquad \{P_2\}\, g\, \{Q_2\}}{}$$



# The "par" combinator

Concurrent separation logic:

$$\frac{\{\, P_1 \,\}\, f \,\{\, Q_1 \,\} \qquad \{\, P_2 \,\}\, g \,\{\, Q_2 \,\}}{\{\, P_1 \star P_2 \,\}\, f \,||\, g \,\{\, Q_1 \star Q_2 \,\}}$$



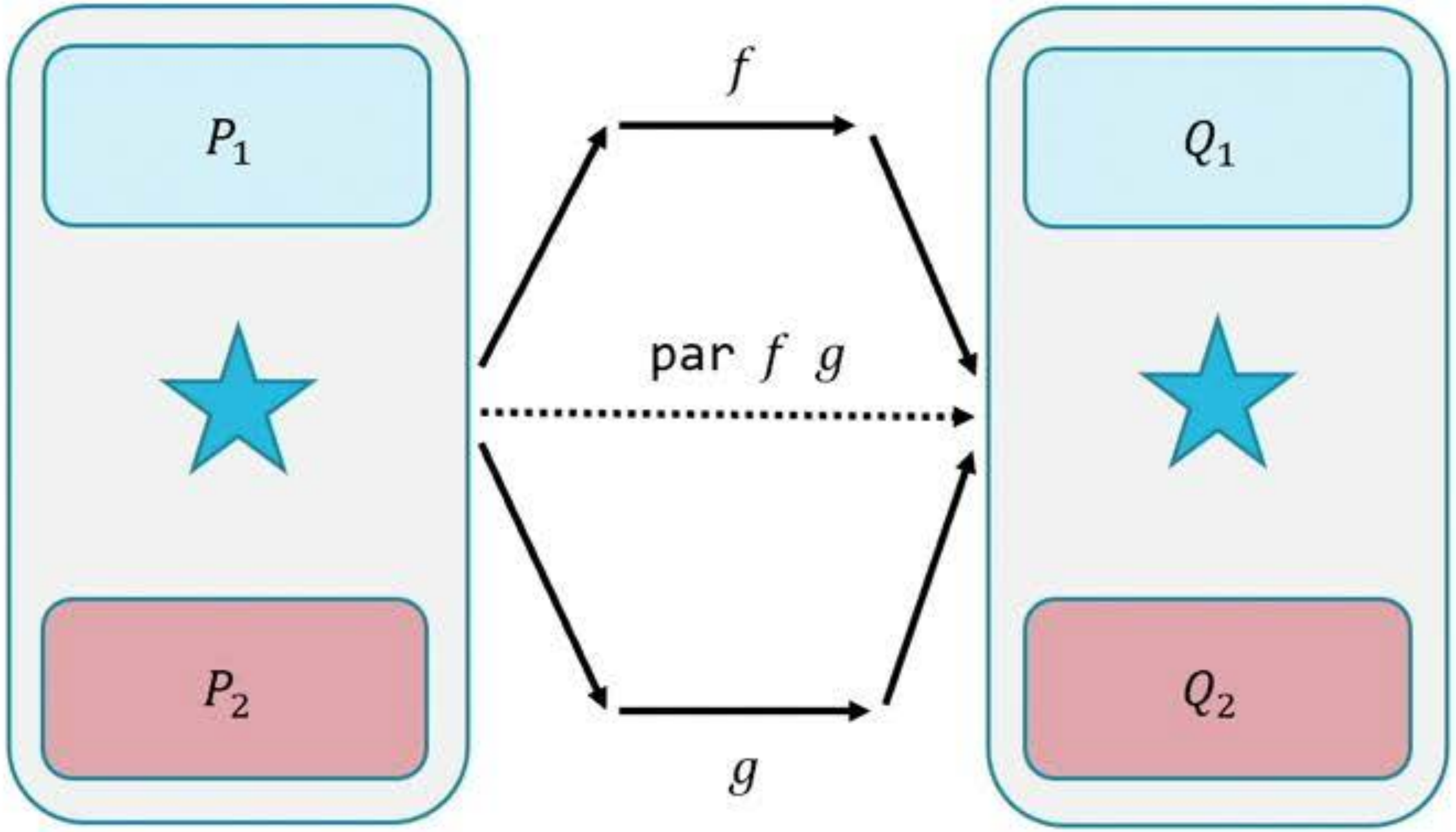
# The “par” combinator

Concurrent separation logic:

$$\frac{\{P_1\}\, f\, \{Q_1\} \qquad \{P_2\}\, g\, \{Q_2\}}{\{P_1 \star P_2\}\, f \,||\, g\, \{Q_1 \star Q_2\}}$$

Steel:

```
val par:
  (f: unit → RST α P₁ Q₁) →
  (g: unit → RST β P₂ Q₂) →
  RST (α * β)
     (expects P₁ ⋆ P₂)
     (provides Q₁ ⋆ Q₂)
```



# Shared mutable access using locks

Concurrent separation logic:

# Shared mutable access using locks

Concurrent separation logic:

$\{P\}$ `new_lock: lock` $P$ $\{\text{emp}\}$

# Shared mutable access using locks

Concurrent separation logic:

| | | |
|---|---|---|
| $\{ P \}$ | `new_lock: lock` $P$ | $\{ \text{emp} \}$ |
| $\{ \text{emp} \}$ | `acquire (`$\ell$`: lock` $P$`)` | $\{ P \}$ |
| $\{ P \}$ | `release (`$\ell$`: lock P)` | $\{ \text{emp} \}$ |

# Shared mutable access using locks

Concurrent separation logic:

Stable invariant

$\{ P \}$ `new_lock: lock` $P$ $\{ \text{emp} \}$

$\{ \text{emp} \}$ `acquire (`$\ell$`: lock` $P$`)` $\{ P \}$

$\{ P \}$ `release (`$\ell$`: lock P)` $\{ \text{emp} \}$

# Shared mutable access using locks

Concurrent separation logic:

Stable invariant

| | | |
|---|---|---|
| $\{P\}$ | `new_lock: lock` $P$ | $\{\text{emp}\}$ |
| $\{\text{emp}\}$ | `acquire (`$\ell$`: lock` $P$`)` | $\{P\}$ |
| $\{P\}$ | `release (`$\ell$`: lock P)` | $\{\text{emp}\}$ |

# Shared mutable access using locks

Concurrent separation logic:

Stable invariant

$\{P\}$ `new_lock: lock` $P$ $\{\text{emp}\}$

$\{\text{emp}\}$ `acquire (`$\ell$`: lock` $P$`)` $\{P\}$

$\{P\}$ `release (`$\ell$`: lock P)` $\{\text{emp}\}$

Steel:

```
val acquire (ℓ:lock R) →
  RST unit
     (expects emp)
     (provides R)
```

# Shared mutable access using locks

Concurrent separation logic:

Stable invariant

$\{ P \}$ `new_lock: lock P` $\{ \text{emp} \}$

$\{ \text{emp} \}$ `acquire (ℓ: lock P)` $\{ P \}$

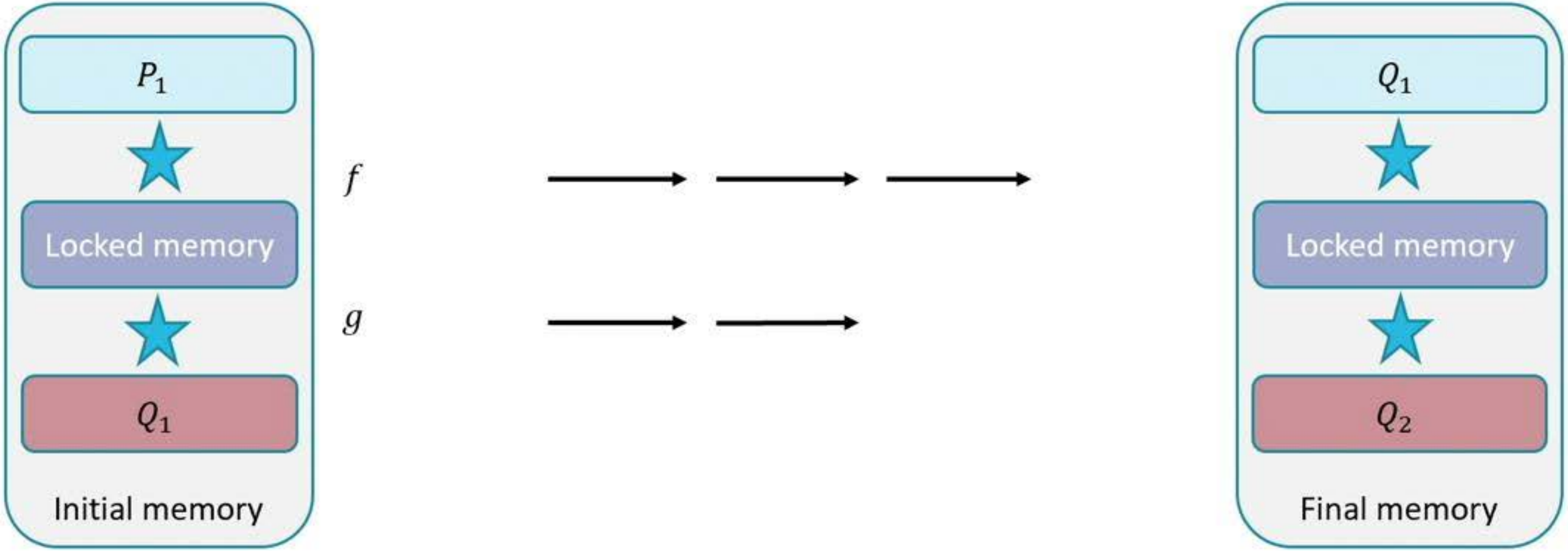
$\{ P \}$ `release (ℓ: lock P)` $\{ \text{emp} \}$

Steel:

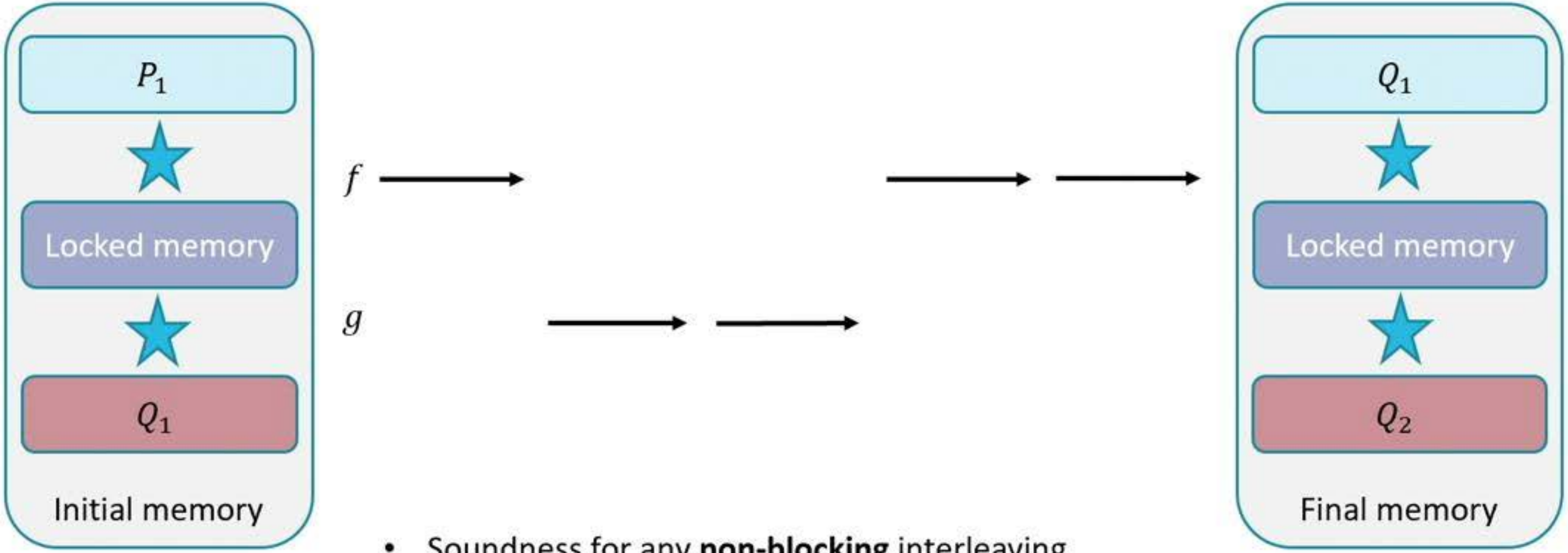
```
val acquire (ℓ:lock R) →
  RST unit
      (expects emp)
      (provides R)
```

- Lock predicates checked statically
- Lock availability checked at runtime

# Soundness of the concurrency model
## (Work in progress)

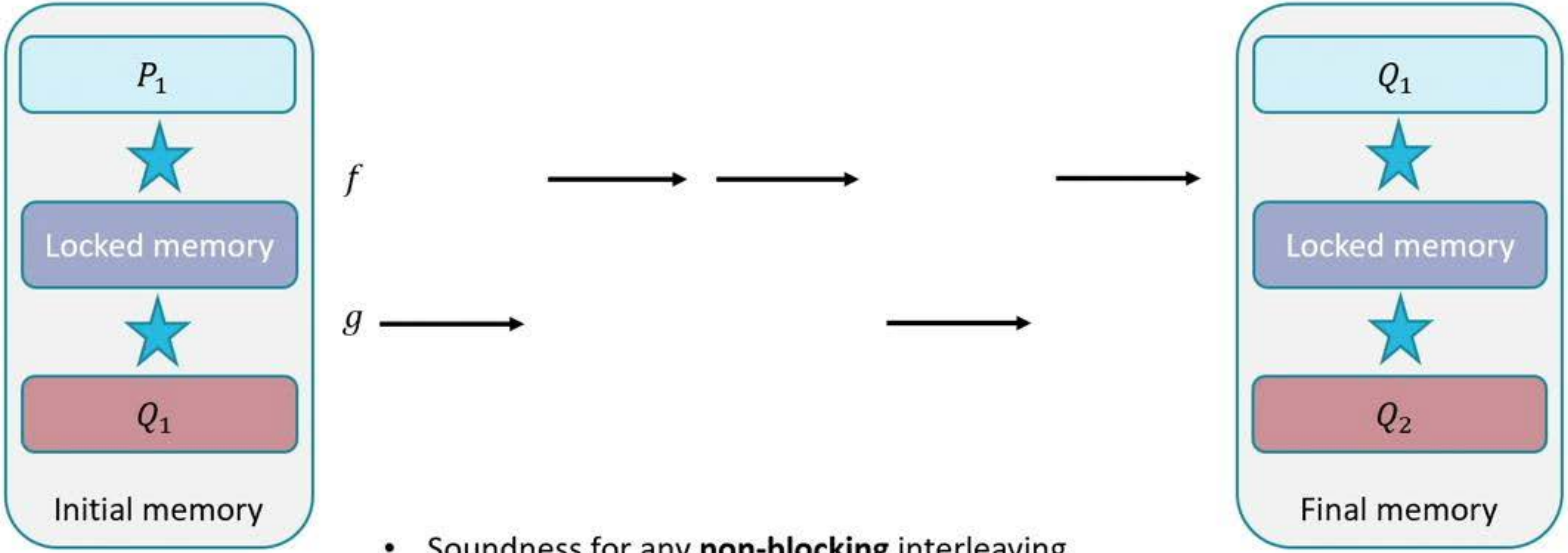


# Soundness of the concurrency model
## (Work in progress)




- Soundness for any **non-blocking** interleaving
- Machine checked in F* !

# Soundness of the concurrency model
## (Work in progress)




- Soundness for any **non-blocking** interleaving
- Machine checked in F* !

# Case study: Linked lists specification

# Case study: Linked lists specification

<u>Steel:</u>

```
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x ↦ hd) ⋆ (slist hd.next tl)
```

# Case study: Linked lists specification

Steel:

```
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x ↦ hd) ⋆ (slist hd.next tl)

val cons (p: pointer) v x ℓ : RST unit
  (expects ((p ↦ v ) ⋆ (slist x ℓ )))
  (provides (slist p (v :: ℓ)))
```

# Case study: Linked lists specification

Steel:

```
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x ↦ hd) ⋆ (slist hd.next tl)

val cons (p: pointer) v x ℓ : RST unit
  (expects ((p ↦ v ) ⋆ (slist x ℓ )))
  (provides (slist p (v :: ℓ)))

val map f p ℓ : RST unit
  (expects (slist p ℓ))
  (provides (slist p (map_cell f ℓ)))
```

# Case study: Linked lists specification

Steel:

```
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x ↦ hd) ⋆ (slist hd.next tl)

val cons (p: pointer) v x ℓ : RST unit
  (expects ((p ↦ v ) ⋆ (slist x ℓ )))
  (provides (slist p (v :: ℓ)))

val map f p ℓ : RST unit
  (expects (slist p ℓ))
  (provides (slist p (map_cell f ℓ)))
```

Low*:

```
let well_formed x ℓ = … (10 lines)
let footprint x ℓ = … (8 lines)

val cons p v x ℓ : ST unit
  (requires
     well_formed x ℓ &&
     disjoint (loc p) (footprint x ℓ) &&
     live p &&
     get p == v)
  (ensures
     well_formed p (v :: ℓ) &&
     modifies (footprint p (v :: ℓ)))
```

cons, head, tail, map specification

- Steel: 30 LOC
- Low*: 100 LOC

# Case study: Linked lists implementation

<u>Steel:</u>

```
let rec map f x ℓ =
  …
  frame
    ((x ↦ hd ℓ) ⋆ slist (hd ℓ).next (tl ℓ))
    (λ _ → (x ↦ f (hd ℓ)) ⋆ slist (hd ℓ).next (tl ℓ))
    (update_cell f x);
  frame
    ((x ↦ f (hd ℓ)) ⋆ slist (hd ℓ).next (tl ℓ))
    (λ _ → (x ↦ f (hd ℓ)) ⋆ slist (hd ℓ).next (map_cell f (tl ℓ)))
    (map f (hd ℓ).next (tl ℓ))
```

# Case study: Linked lists implementation

Steel:

```
let rec map f x ℓ =
  …
  frame
    ((x ↦ hd ℓ) ⋆ slist (hd ℓ).next (tl ℓ))
    (λ _ → (x ↦ f (hd ℓ)) ⋆ slist (hd ℓ).next (tl ℓ))
    (update_cell f x);
  frame
    ((x ↦ f (hd ℓ)) ⋆ slist (hd ℓ).next (tl ℓ))
    (λ _ → (x ↦ f (hd ℓ)) ⋆ slist (hd ℓ).next (map_cell f (tl ℓ)))
    (map f (hd ℓ).next (tl ℓ))
```

Ideally:

```
let rec map f x ℓ =
  …
  update_cell f x;
  map f (hd ℓ).next (tl ℓ)
```

Work in progress: Better frame inference

# Case study: Doubly-linked lists

- Doubly-linked lists in Steel: 400 LoCs
- In $\mathrm{Low}^\star$: 4000 LoCs!

# Case study: Doubly-linked lists

- Doubly-linked lists in Steel: 400 LoCs
- In Low*: 4000 LoCs!

- Doubly-linked lists are not expressible in Rust without unsafe blocks due to aliasing restrictions
- Steel is expressive enough to capture complex aliasing patterns

# Future work

- Improve usability of the framework (3 – 6 months)
  - Frontend syntax
  - More fine-tuning of SMT queries
  - Additional libraries
  - Complete interoperation with Low*

# Future work

- Improve usability of the framework (3 – 6 months)
  - Frontend syntax
  - More fine-tuning of SMT queries
  - Additional libraries
  - Complete interoperation with Low*
- Concurrency (1 – 2 months)
  - Deadlock prevention
  - Complete proof of soundness

# At last, a $\star$ for $\mathbf{F}^\star$!

# At last, a $\star$ for $F^\star$!

- Separation logic in $F^\star$: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-$F^\star$: Tactics + SMT make it possible

# At last, a ⋆ for F$^\star$!

- Separation logic in F$^\star$: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-F$^\star$: Tactics + SMT make it possible
- **This summer:** The right abstractions with resource typing to make it scale

# At last, a ⋆ for $F^\star$!

- Separation logic in $F^\star$: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-$F^\star$: Tactics + SMT make it possible
- **This summer:** The right abstractions with resource typing to make it scale
- Many applications targeted: Beyond crypto verification
  - Concurrent networking protocols, e.g. Quic
  - Critical systems components in Azure: Parts of Hyper-V? Azure CCF?

# At last, a ⋆ for $F^\star$!

- Separation logic in $F^\star$: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-$F^\star$: Tactics + SMT make it possible
- **This summer:** The right abstractions with resource typing to make it scale
- Many applications targeted: Beyond crypto verification
  - Concurrent networking protocols, e.g. Quic
  - Critical systems components in Azure: Parts of Hyper-V? Azure CCF?
- Many possible synergies:
  - Verifying Rust programs inside of Steel?
  - Verifying Verona components (ownership-based systems language from MSR Cambridge)?

# At last, a ⋆ for $F^\star$!

- Separation logic in $F^\star$: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-$F^\star$: Tactics + SMT make it possible
- **This summer:** The right abstractions with resource typing to make it scale
- Many applications targeted: Beyond crypto verification
  - Concurrent networking protocols, e.g. Quic
  - Critical systems components in Azure: Parts of Hyper-V? Azure CCF?
- Many possible synergies:
  - Verifying Rust programs inside of Steel?
  - Verifying Verona components (ownership-based systems language from MSR Cambridge)?

**fromherz@cmu.edu** **denis.merigoux@inria.fr**

# A frame rule for Steel

Classic frame rule:

$$\frac{\{Q\}\ f\ \{R\}}{\{P \star Q\}\ f\ \{P \star R\}}$$

In Steel:

```
val frame :
  (P ⋆ Q: resource) →
  (P ⋆ R: resource) →
  (f: unit → RST α (expects Q) (provides R)) →
  RST α
      (expects (P ⋆ Q))
      (provides (P ⋆ R))
```