Commutativity Reasoning for the Heap

Jared Pincus

Stevens Institute of Technology

April 27, 2022

Commutativity in Context

Used in increasingly many tools and theories!

- Scalable concurrency
- Smart contracts
- Transactional memory
- Automatic parallelization

… What’s commutativity?

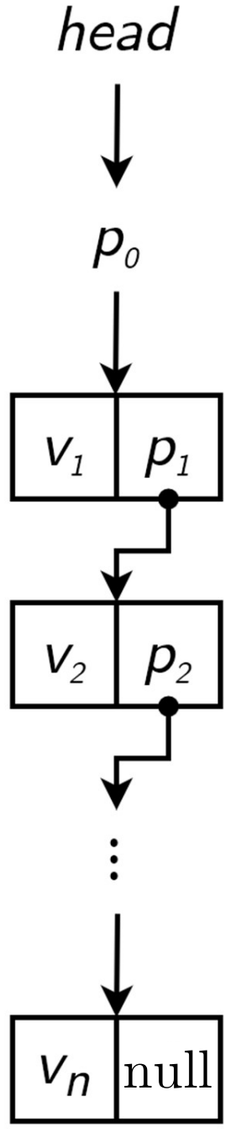
Motivating Example: Linked Stack

Consider a **linked stack** with `push`, `pop`, and `peek` operations.

```
push(v):
  h = ! head
  p = (v, h)
  r = ref p
  h <- r
```

```
pop():
  h = ! head
  if h <> null
  then
    (v, p) = ! h
    head <- p
    free h
```

```
peek():
  h = ! head
  (v, _) = ! h
  return v
```



Given two operations on the stack, when can we **swap** their **order of execution** and maintain the **same outcome**?

Standard Definition of Commutativity

Define **programs** to have type: $\text{input vector} \to \text{state} \to \text{state}$.

Assume **determinism** and **termination**, so programs are well-defined.

Consider a predicate P on a state and two vectors of values,
and an **observational equivalence** relation $\sim$.

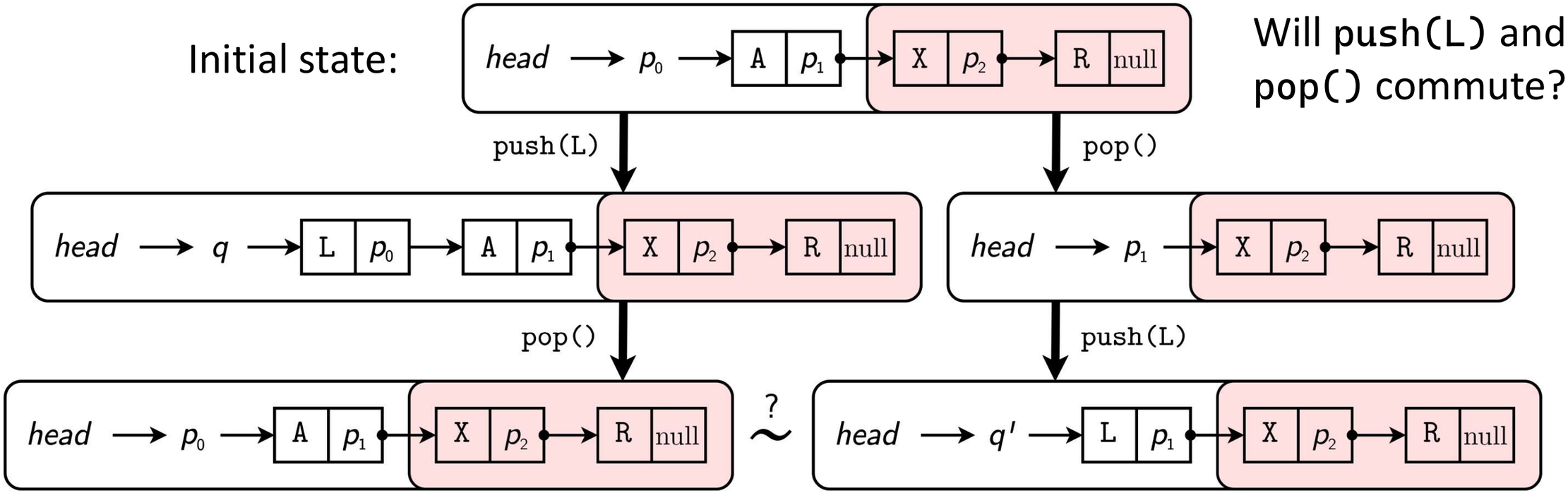
Equivalence modulo *observation*, not exact equality

Two programs C_1 and C_2 **commute** under P, denoted $C_1 \bowtie_P C_2$, if

$$\forall \sigma\ \bar{x}\ \bar{y}\ .\ P\ \sigma\ \bar{x}\ \bar{y}\ \Longrightarrow\ C_2\ \bar{y}\ (C_1\ \bar{x}\ \sigma) \sim C_1\ \bar{x}\ (C_2\ \bar{y}\ \sigma).$$

When $C_1 \bowtie_P C_2$ holds, we call P a correct **commutativity condition** for C_1 and C_2.

Motivating Example: Linked Stack



The bottom cells of the stack remain unread and unwritten.
But we still check both entire states for equivalence!

How can we **frame away** untouched pieces of state?

Key Contributions

Commutativity with Heaps

Separation Logic

Abstraction Predicates

Framed Commutativity

Implementation in Coq

Software Foundations Volume 6

Automation Tactics

Correctness and Completeness Proofs

Veracity Interpreter

Commute Blocks

Runtime with Parallelization

Heap Integration

Key Contributions

Commutativity with Heaps

Separation Logic

Abstraction Predicates

Framed Commutativity

Implementation in Coq

Software Foundations Volume 6

Automation Tactics

Correctness and Completeness Proofs

Veracity Interpreter

Commute Blocks

Runtime with Parallelization

Heap Integration

Separation Logic: Heap Predicates

Separation logic concerns **pieces of state** called **heaps**:
finite maps from **addresses** to **values**.

Heaps are characterized using **heap predicates**.

$$\frac{}{\mathbf{emp}\ \emptyset}\ \text{Empty} \qquad \frac{}{(p \mapsto v)\ (p{:}\, v)}\ \text{Single} \qquad \frac{P}{\forall h\ .\ [P]\ h}\ \text{Pure}$$

$$\frac{P\ h_1 \quad Q\ h_2 \quad h_1 \cap h_2 = \emptyset}{(P * Q)\ (h_1 \cdot h_2)}\ \text{Separating Conjunction}$$

Separation Logic: Heap Triples

Heap triples are the Hoare triples of separation logic.

Given program C and heap predicates P and Q, $\{P\}\ C\ \{Q\}$ holds if $\forall h\,.\,P\ h \Rightarrow Q\ (C\ h)$.

$$\frac{P' \Rightarrow P \quad \{P\}\ C\ \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$ Rule of Consequence

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$ Frame Rule

Ex: $\{x \mapsto 5\}\ x := 2\ \{x \mapsto 2\}$

$\{x \mapsto 5 * y \mapsto 7\}\ x := 2\ \{x \mapsto 2 * y \mapsto 7\}$

$\{\mathbf{emp}\}\ \text{ref}\ 5\ \{\exists p\,.\,p \mapsto 5\}$

Framed Commutativity: Attempt 1

$$\{P\ \bar{x}\ \bar{y} * \mathbf{H}\}$$

$$C_1\ \bar{x}; C_2\ \bar{y} \ \Big|\ C_2\ \bar{y}; C_1\ \bar{x}$$

$$\{Q * \mathbf{H}\}$$

P is the **commutativity condition** on the **program inputs** and **initial heap**.

$\mathbf{H}$ is easily framed away.

Both post-heaps must satisfy Q ~~– equivalence!~~

This doesn't work! Q isn't precise enough.

If Q **lacks existentials**, h_1 and h_2 need to be **identical**.

If Q **includes existentials**, the **relationship** between h_1 and h_2 is severed.

Framed Commutativity: Attempt 2

$\{P\ \bar{x}\ \bar{y} * \mathbf{H}\}$	$\{P\ \bar{x}\ \bar{y} * \mathbf{H}\}$
$C_1\ \bar{x}; C_2\ \bar{y}$	$C_2\ \bar{y}; C_1\ \bar{x}$
$\{Q_1 * \mathbf{H}\}$	$\{Q_2 * \mathbf{H}\}$

$\sim$

Now h_1 and h_2 satisfy distinct postconditions.

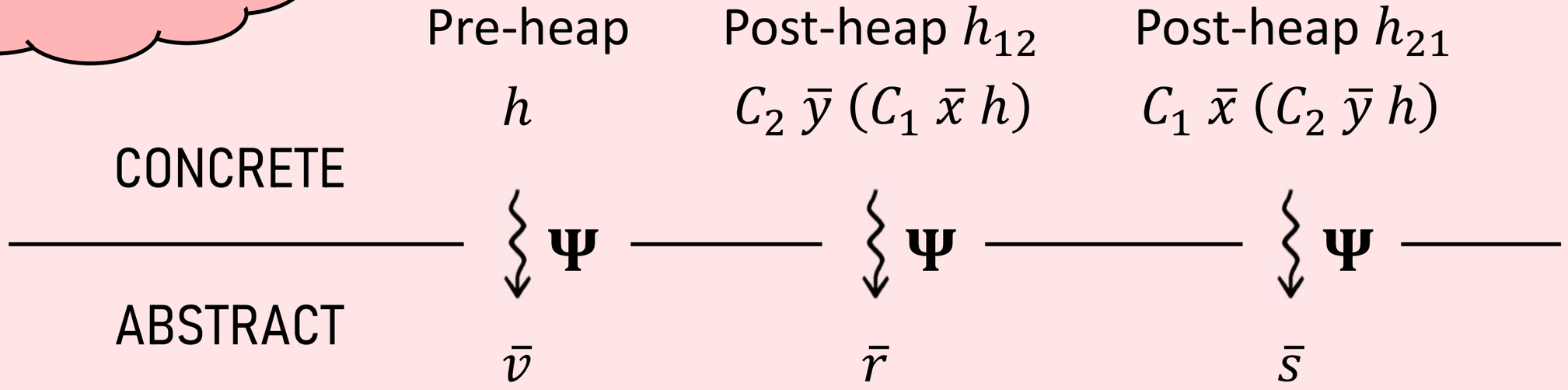
But we still haven't related h_1 and h_2!

Derive relatable values from postconditions?

Bottom line: heap triples aren't cutting it…

Abstraction Predicates

$P\ \bar{v} \Longrightarrow \bar{r} \sim \bar{s}$



$\mathbf{\Psi}$ is a problem-specific **abstraction predicate** parameterized on an **abstraction vector**.

Ex: Non-negative counter $\quad \mathbf{\Psi}\ \langle v \rangle \coloneqq c \mapsto v * [v \geq 0]$

Two-set $\quad \mathbf{\Psi}\ \langle u, v \rangle \coloneqq p \mapsto u * q \mapsto v * [u \neq v]$

Framed Commutativity

$$\begin{array}{lll} & \boldsymbol{\Psi} & \\ h & \rightsquigarrow & \bar{v} \\ h_{12} & \rightsquigarrow & \bar{r} \\ h_{21} & \rightsquigarrow & \bar{s} \end{array}$$

$$h_{12} \coloneqq C_2\ \bar{y}\ (C_1\ \bar{x}\ h)$$
$$h_{21} \coloneqq C_1\ \bar{x}\ (C_2\ \bar{y}\ h)$$

$P\ \bar{v} \Longrightarrow \bar{r} \sim \bar{s}$

Ingredients:

C_1, C_2	Programs
$\boldsymbol{\Psi}$	Abstraction predicate
$\sim$	Eq. relation on abstraction vectors
$\mathcal{P}$	Predicate on 3 vectors

C_1 and C_2 **commute** under $\mathcal{P}$, denoted $C_1 \bowtie_{\mathcal{P}} C_2$, if

$$\forall \mathbf{H}\ \bar{v}\ \bar{x}\ \bar{y}\ h\ .\ \underbrace{(\boldsymbol{\Psi}\ \bar{v} * \mathbf{H})\ h\ \wedge\ \mathcal{P}\ \bar{v}\ \bar{x}\ \bar{y}}_{\text{PRECONDITION}} \Longrightarrow \underbrace{\exists \bar{r}\ \bar{s}\ .\ (\boldsymbol{\Psi}\ \bar{r} * \mathbf{H})\ h_{12} \wedge (\boldsymbol{\Psi}\ \bar{s} * \mathbf{H})\ h_{21} \wedge \bar{r} \sim \bar{s}}_{\text{POST-RELATION}}$$

We say that $\mathcal{P}$ is a **correct commutativity condition** for C_1 and C_2.

Motivating Example: Linked Stack

Consider again our linked stack with **push**, **pop**, and **peek** operations.

$\boldsymbol{\Psi}$ should encode a **partial stack** to some depth,
by observing the stack contents V and last pointer f before framed portion.

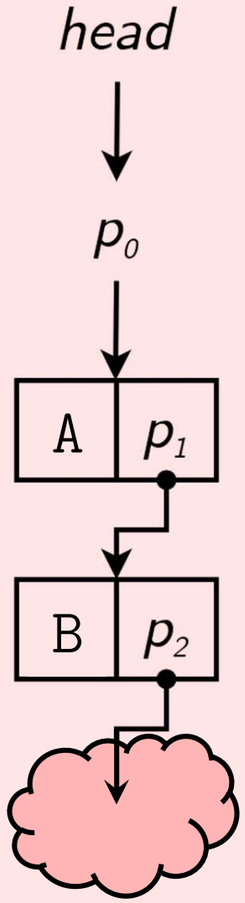
Abstraction for partial stack (top n values):

$$\boldsymbol{\Psi}_n\ \langle f, V\rangle \quad \coloneqq [|V| = n]\ *\ \exists p\ .\ head \mapsto p\ *\ \psi\ f\ p\ V$$
$$\psi\ f\ p\ (v :: t) \ \coloneqq\ \exists q\ .\ p \mapsto (v, q)\ *\ \psi\ f\ q\ t$$
$$\psi\ f\ p\ \mathrm{nil} \quad \coloneqq\ [p = f]$$

Abstraction for total stack (all values):

$$\boldsymbol{\Psi}_\star\ V \coloneqq \boldsymbol{\Psi}_{|V|}\ \langle \mathrm{null}, V\rangle$$

$V = [\mathtt{A}, \mathtt{B}]$
$f = p_2$



Motivating Example: Linked Stack

For each pair of stack operations, select particular depths for Ψ_n in the precondition and post-relation.

Operation pair	Pre depth	Post depth	V	$\mathcal{P}$
`push(x)` ⋈ `push(y)`	0	2	`[]`	`x = y`
`push(x)` ⋈ `peek`	1	2	`[v]`	`v = x`
`push(x)` ⋈ `pop`	1	1	`[v]`	`v = x`
`peek` ⋈ `peek`	1	1	`[_]`	`T`
`peek` ⋈ `pop`	2	1	`[v1,v2]`	`v1 = v2`
`pop` ⋈ `pop`	2	0	`[_,_]`	`T`

We will only ever need a depth of 2 for all stack reasoning!

Complete Conditions

Correct condition formula:

$$\forall \mathbf{H}\ \bar{v}\ \bar{x}\ \bar{y}\ h\,.\ (\mathbf{\Psi}\ \bar{v} * \mathbf{H})\ h\ \wedge\ \mathcal{P}\ \bar{v}\ \bar{x}\ \bar{y} \implies \exists \bar{r}\ \bar{s}\,.\,(\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12} \wedge (\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21} \wedge \bar{r} \sim \bar{s}$$

Conversely, $\mathcal{P}$ is **complete** if

$$\forall \mathbf{H}\ \bar{v}\ \bar{x}\ \bar{y}\ h\,.\ (\mathbf{\Psi}\ \bar{v} * \mathbf{H})\ h \implies \exists \bar{r}\ \bar{s}\,.\,(\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12} \wedge (\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21} \wedge \bar{r} \sim \bar{s} \implies \mathcal{P}\ \bar{v}\ \bar{x}\ \bar{y}$$

Thus, $\mathcal{P}$ is **exact** (correct and complete) if

$$\forall \mathbf{H}\ \bar{v}\ \bar{x}\ \bar{y}\ h\,.\ (\mathbf{\Psi}\ \bar{v} * \mathbf{H})\ h \implies \left(\exists \bar{r}\ \bar{s}\,.\,(\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12} \wedge (\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21} \wedge \bar{r} \sim \bar{s} \iff \mathcal{P}\ \bar{v}\ \bar{x}\ \bar{y} \right)$$

Evaluation

As framed commutativity is partly **relational**, it does not play nicely with standard heap triples out of the box.

Framed commutativity is not easily composable,
e.g. deriving $a \bowtie b; c$ from $a \bowtie b$ and $a \bowtie c$ is not straightforward.

Relational separation logic may offer some solutions.

Key Contributions

Commutativity with Heaps

Separation Logic

Abstraction Predicates

Framed Commutativity

Implementation in Coq

Software Foundations Volume 6

Automation Tactics

Correctness and Completeness Proofs

Veracity Interpreter

Commute Blocks

Runtime with Parallelization

Heap Integration

Infrastructure

Coq implementation is built atop the separation logic library in Volume 6 of *Software Foundations*.

Library features a toy language with big-step semantics.

```
Example push v :=
  <{ let 'h = ! head in
     let 'p = `pair 'h v in
     let 'r = ref 'p in
     head := 'r;
     () }>.
```

```
Example pop :=
  <{ let 'h = ! head in
     let 'g = 'h = null in
     if 'g then () else
     let 'p = ! 'h in
     let 'q = fst 'p in
     head := 'q;
     free 'h;
     () }>.
```

```
Example peek (r : loc) :=
  <{ let 'h = ! head in
     let 'p = ! 'h in
     let 'v = snd 'p in
     r := 'v; () }>.
```

Developed Utilities

Objective: automate commutativity proofs as much as possible.

We developed a suite of tactics and algorithms, all mostly automated:

- Dispatch heap-based goals
- Fact derivation from heap-based hypotheses
- Forward big-step symbolic execution of program goals
- Inverted big-step symbolic execution of program hypotheses

Correctness Proof Structure

```
Example comm_c1_c2 : forall v x y,
  P v x y ->
  comm (c1 x) (c2 y) (psi v) R.
Proof.
```

Setup ⟶ `introv HP phi. unfolds R, psi, c1, c2.`

Derive facts about h from $(\mathbf{\Psi}\ \bar{v} * \mathbf{H})\ h$ ⟶ `destruct_heap phi.`

`(* Other simplifications here *)`

Forward symbolic execution 4 times ⟶ `eexists*; splits; apply_evals.`

Instantiate existentials of post-relation ⟶ `eexists*. splits;`

Dispatch $(\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12}$ and $(\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21}$ ⟶ `solve_hprop.`

Problem-specific dispatch of $\bar{r} \sim \bar{s}$ ⟶ `(* Solve r ~ s here *)`

```
Qed.
```

Completeness Proof Structure

```
Example comm_c1_c2_inv: forall v x y,
  comm (c1 x) (c2 y) (psi v) R ->
  P v x y.
Proof.
```

Setup	⟶	`introv C.`
Construct a heap h that satisfies $\mathbf{\Psi}\ \bar{v}$	⟶	`assert ((psi v) h) as H. { solve_hprop. }`
Instantiate `comm` with h and $\mathbf{H} \coloneqq \mathbf{emp}$	⟶	`apply_comm C H. unfolds R, psi, c1, c2.`
Inverted symbolic execution 4 times	⟶	`invert_evals.`
Derive facts about h_{12} and h_{21}	⟶	`destruct_heap HR1, HR2.`
Problem-specific facts from $\bar{r} \sim \bar{s}$	⟶	`(* Derive facts from r ~ s here *)`
Problem-specific dispatch of $\mathcal{P}$	⟶	`(* Dispatch P here *)`

```
Qed.
```

Evaluation

Functionality:

- Largely automated verification of $\mathcal{P}$ **correctness** and **completeness**
- Interactive $\mathcal{P}$ **inference** in some scenarios

Useful contexts:

- Reason about programs with multiple data structures for free
- Simple reasoning about inductive structures

Challenges:

- Requires full big-step symbolic execution
- Not compositional – every commutativity proof stands in isolation

Key Contributions

Commutativity with Heaps

Separation Logic

Abstraction Predicates

Framed Commutativity

Implementation in Coq

Software Foundations Volume 6

Automation Tactics

Correctness and Completeness Proofs

Veracity Interpreter

Commute Blocks

Runtime with Parallelization

Heap Integration

Commute Blocks

Veracity is a C-like language with the novel construct of the **commute block**.

```
commute (guard) {
  { f1 }
  { f2 }
}
```

```
commute (c >= 0) {
  { c = c + 1; }
  { if (c > 0) c = c - 1; }
}
```

`commute` has two equivalent semantics:

1. Sequential (`guard` is `false`): `f1; f2`
2. Concurrent (`guard` is `true`): Run `f1` and `f2` in parallel (with Multicore OCaml)

Look out for our presentation at NJPLS!

Heap Integration

Toolchain uses abstraction refinement to infer and verify `commute` block guards.

Interpreter (OCaml) encodes state as `(string * (value ref)) list`.

- Similar in principle to separation logic heaps

Possible heap integrations:

1. Encode separation logic in infer/verify toolchain
2. Coq theories ⟼ VCY interpreter
3. VCY programs ⟼ Coq AST

Summary

Framed Commutativity

- Novel abstraction for heap-based commutativity reasoning
- Simplified modeling/reasoning via framing

Coq Implementation

- Automation of framed commutativity proofs
- Room to develop more capabilities

Veracity Interpreter

- Commute block semantics
- Opportunities for heap integration

Thank you!

All comments and questions are welcome.

jpincus@stevens.edu

Related Work

Papers that verify/discover/prove commutativity:

- Bansal, Koskinen, Tripp. “Automatic Generation of Precise and Useful Commutativity conditions.” TACAS’18
- Kim, Rinard. “Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures.” PLDI’11
- Pîrlea, Kumar, Sergey. “Practical Smart Contract Sharding with Ownership and Commutativity Analysis.” PLDI’21
- Chen, Fathololumi, Koskinen, Pincus. “Veracity: Declarative Multicore Programming with Commutativity.” *Under review*

Papers that use commutativity:

- Diniz, Rinard. “Dynamic Feedback: An Effective Technique for Adaptive Computing.” PLDI’97
- Herlihy, Koskinen. “Transactional boosting: a methodology for highly-concurrent transactional objects.” PPoPP’08
- Clements et al. “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors.” SOSP’13

Return Value Lifting

How do we reason about functions with return values?

Augment $\mathbf{\Psi}$ with addresses to which programs a and b write any returns.

$$\mathbf{\Psi}'\ \langle v_a, v_b, \bar{v}\rangle \ :=\ \rho_a \mapsto v_a * \rho_b \mapsto v_b * \mathbf{\Psi}\ \bar{v}$$

$$\langle r_a, r_b, \bar{r}\rangle \sim'\ \langle s_a, s_b, \bar{s}\rangle \ :=\ r_a = s_a \wedge r_b = s_b \wedge \bar{r} \sim \bar{s}$$

Path Sensitivity

Recall the example of a two-element set, that stores non-negative integers at p and q.

```
Example mem (r : loc) (v : int) :=
  <{ let 'vp = ! p in
     let 'vq = ! q in
     let 'ep = ('vp = v) in
     let 'eq = ('vq = v) in
     if 'ep then
       (r := true; ())
     else if 'eq then
       (r := true; ())
     else
       (r := false; ()) }>.
```

```
Definition psi (u v : int) (t1 t2 : bool) :=
  p ~~> u \* q ~~> v \*
  \[u >= 0 \/ v >= 0 -> u <> v] \*
  r1 ~~> t1 \* r2 ~~> t2.

Definition R H h1 h2 :=
  exists u1 v1 u2 v2 t1 t2,
  (psi u1 v1 t1 t2 \* H) h1 /\
  (psi u2 v2 t1 t2 \* H) h2 /\
  ((u1 = u2 /\ v1 = v2) \/ (u1 = v2 /\ v1 = v2)).
```

To prove `mem` $\bowtie_{\top}$ `mem`, we must account for ITE branching!

Preemptive Guard Destruction

```
Example mem (r : loc) (v : int) :=
  <{ let 'vp = ! p in
     let 'vq = ! q in
     let 'ep = ('vp = v) in
     let 'eq = ('vq = v) in
     if 'ep then
       (r := true; ())
     else if 'eq then
       (r := true; ())
     else (r := false; ()) }>.
```

Naïve solution:

Destruct on every guard *before* existential instantiation

Kill any paths which contradict $\mathcal{P}$ hypothesis

Perform up to $2^{\#\,\text{guards}}$ proofs in parallel

```
Example comm_mem_mem :
  forall (u v x y : int) (t1 t2 : bool),
  x >= 0  ->  y >= 0 ->
  comm (mem r1 x) (mem r2 y)
       (psi u v t1 t2) R.
Proof.

  introv Zx Zy S. unfolds mem, R, psi.
  destruct_heap S. simpl_disjoint.

  either (u = x); either (v = x);
  either (u = y); either (v = y); subst.

  all: try contradiction.
  all: try (exfalso; apply~ L; fail).

  all: solve_comm.
  all: eexists*; splits.
  all: solve_hprop.
  all: auto.

Qed.
```