COMMUTATIVITY REASONING FOR THE HEAP

by

Jared Pincus

A THESIS

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE — COMPUTER SCIENCE

Jared Pincus, Candidate

ADVISORY COMMITTEE

Eric Koskinen, Chairman Date

Michael Greenberg Date

STEVENS INSTITUTE OF TECHNOLOGY
Castle Point on Hudson
Hoboken, NJ 07030
2022

COMMUTATIVITY REASONING FOR THE HEAP

ABSTRACT

The program property of commutativity is analyzed or utilized in a growing body of theories and tools, including efforts in scalable concurrency, data structure design, and program parallelization. However, current work in commutativity lacks useful abstractions for reasoning about programs that modify the heap.

We introduce a theoretic framework for reasoning about the commutativity of programs with a heap-based memory model. We construct a specialized form of separation-logic-style framing, that incorporates a commutativity precondition and an observational equivalence relation on the post-states. This form permits the framing away of unimportant pieces of the heap, dramatically simplifying commutativity proofs.

We show an implementation of this theory in Coq, featuring tactics that enable largely automated proofs of the correctness and completeness of commutativity conditions, as well as interactive inference of conditions in some scenarios. Finally, we consider how to integrate this progress in heap-based reasoning with our implementation of automatic commutativity reasoning in the Veracity programming language.

Author: Jared Pincus
Advisor: Eric Koskinen
Date: May 4, 2022
Department: Computer Science
Degree: Master of Science — Computer Science

To PF, VC, FH, and most of all JPP,
without whom I would not be where and who I am today.

Acknowledgments

I must begin by expressing my gratitude to my advisor, Eric Koskinen, for his invaluable mentorship over this past year. My current academic trajectory would genuinely be completely different were it not for his guidance.

I must also thank Michael Greenberg for his insightfulness during the thesis process, and for his remarkable impact on the programming languages community at Stevens.

I am also grateful to Parisa Fathololumi and Adam Chen, for continuing to be excellent team members on the Veracity project.

To my professors, who have offered such remarkable opportunities in the form of teaching assistantships, research experiences, fascinating conversations, and painful exams — you have been instrumental in making my time at Stevens so valuable.

To my friends of every sort, thank you for sticking with me at my worst and best, my busiest and freest.

And of course, I owe my life in the most literal sense to my parents. Words cannot capture the extent to which you have guided and supported me throughout my existence. I'll make it up to you someday.

Table of Contents

Chapter 1
Introduction

Imagine a stack data structure, featuring the usual operations of `push(E)`, `pop()`, and `peek()`. Suppose we perform two of these operations in sequence, say, `push(E)` followed by `pop()`. In this case, we will obviously end up with the same stack as before.

The question we will explore is, what happens when we reverse the order to be `pop()` followed by `push(E)`? Will the outcome be the same as before? Or to ask question differently, under what initial conditions will `push(E); pop()` produce the same result as `pop(); push(E)`? With these two operations in particular, the result will be the same when the initial stack has `E` as its top value. But said condition will differ based on the two operations under consideration. In general, the ability for two operations to swap order without consequence is called *commutativity*, and it is the topic of this thesis.

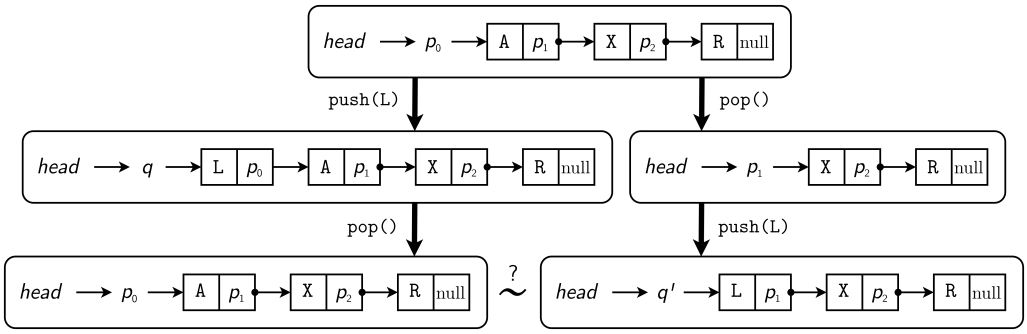
1.1 Related Work

Commutativity as a program property is being used in increasingly many tools and theories in the domain of programming languages research. Areas utilizing commutativity include scalable concurrency [Clements et al. (2013)], smart contracts [Pîrlea et al. (2021)], transactional memory [Herlihy and Koskinen (2008)], and automatic parallelization [Chen et al. (2022)].

Supporting these use cases are a variety of efforts to analyze commutativity in theoretical and applied settings. Bansal et al. (2018) use an abstraction refinement loop to synthesize useful commutativity conditions, a process realized in the Servois analysis tool and used by the Veracity language, as discussed in § 5.1. Kim and Rinard (2011) verify commutativity conditions for linked data structures by analyzing structure specifications. Pîrlea et al. (2021) perform abstract interpretation to assess absolute commutativity. Chen et al. (2022) synthesize and verify commutativity conditions for declarative programs. As it stands, the commutativity space lacks useful abstractions for reasoning about programs with a heap-based memory model. To appreciate the value of such a model, let's return to our stack example.

1.2 Motivating Example

Using a linked-list-style implementation for our stack, we'll begin with a stack containing `[A,X,R]`. Observe what happens when we apply `push(L)` and `pop()` in either order.



To assess equivalence between the final states, we consider the values on the stack, and importantly, we disregard (non)equivalence between pointers. Obviously, the final states are not equivalent — left contains `[A,X,R]`, right contains `[L,X,R]`. Indeed, if we had pushed `A` instead of `L`, we would have equivalence.

Notice that in every step of the operations, the bottom portion of the stack remains unread and unwritten. Yet, we needlessly keep track of it the entire time, and assess it at the end for equivalence. In this work, we will leverage the compositionality of separation logic heaps, to *frame away* superfluous pieces of state, in an effort to simplify commutativity reasoning.

There are two key contexts in which a heap-based representation of commutativity with framing can simplify reasoning. The first is in program settings with multiple data structures, in which certain operations only interact with individual structures. The second, which we will focus on, is settings in which we only operate on *part* of a data structure in a predictable way, permitting us to frame away the rest of the structure.

1.3 Overview

Below, we summarize the three main components of contribution in this work.

Framed Commutativity

We introduce a theoretic framework for reasoning about the commutativity of programs with a heap-based memory model. We construct a specialized form of separation-logic-style framing, that incorporates a commutativity precondition and an observational equivalence relation on the post-states. This form permits the framing away of unimportant pieces of the heap, dramatically simplifying commutativity proofs.

Implementation in Coq

We show an implementation of this theory in Coq, featuring tactics that enable largely automated proofs of the correctness and completeness of commutativity conditions, as well as interactive inference of conditions in some scenarios.

Veracity Interpreter

We consider how to integrate this progress in heap-based reasoning with our implementation of automatic commutativity reasoning in the Veracity programming language.

Chapter 2
Preliminaries

2.1 Separation Logic

Separation logic, first laid out by Reynolds (2002), is a model for reasoning about *heaps* — finite maps from abstracted *addresses* to *values*. Heaps are similar to the general idea of a "state"; we distinguish them with the notion that a heap may represent only a portion of a larger state. We might choose to compose smaller heaps into a larger heap, or decompose larger into smaller.

Here, we will summarize the aspects of separation logic requisite for our work.

Heaps

The smallest heap is the empty heap $\varnothing$. We denote a singleton heap as $p : v$, where address p points to value v. A heap with multiple addresses is $p_1 : v_1, p_2 : v_2, \ldots, p_3 : v_3$. Addresses shadow in order, so reading p from $p : v_1, p : v_2$ returns v_1.

Heaps may be unioned with $h_1 \cdot h_2$. Union is asymmetric in that addresses from h_1 will shadow those in h_2.

Heap Predicates

Heaps are characterized using heap predicates. First, observe the rules of the empty, single, pure, and existential predicates.

$$\frac{}{\mathbf{emp}\ \varnothing}\ \text{Empty} \qquad \frac{}{(p \mapsto v)\ (p : v)}\ \text{Single} \qquad \frac{P}{[P]\ h}\ \text{Pure} \qquad \frac{\exists x\ .\ (F\ x)\ h}{(\exists x\ .\ F\ x)\ h}\ \text{Exists}$$

emp holds for the empty heap; $(p \mapsto v)$ holds for a singleton heap with address p and value v; $[P]$ holds for any heap so long as the proposition P is true; $(\exists x\ .\ F\ x)$ enables the "injection" of an existential variable into a heap predicate.

These predicates are not too useful in isolation. To combine them, we use the separating conjunction (written $*$), which characterizes the union of disjoint heaps.

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

Heap Triples

Heap triples are the Hoare triples of separation logic. Given a program C which modifies the heap, and heap predicates P and Q, $\{P\}\ C\ \{Q\}$ holds if
$\forall h\ .\ P\ h \Longrightarrow Q\ (C\ h)$.

Many rules for Hoare triples also apply to heap triples. One example is the rule of consequence:

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

We also get an additional rule, one very critical to our work — the frame rule:

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

Intuitively, the frame rule permits local reasoning in state. We can tack additional pieces of state onto a program's pre-heap, with the knowledge that this added piece must remain unchanged in the post-heap. And, if a program leaves a piece of state unread and unwritten, we may *frame away* that piece from the pre- and post-heaps.

Some simple examples of valid heap triples include:

$$\{x \mapsto 5 * y \mapsto 7\}\ x \coloneqq 2\ \{x \mapsto 2 * y \mapsto 7\}$$
$$\{x \mapsto 5\}\ x \coloneqq 2\ \{x \mapsto 2\}$$
$$\{\mathbf{emp}\}\ \text{ref}\ 5 \quad \{\exists p\ .\ p \mapsto 5\}$$

The first example shows that assigning x to 2 changes the value at address x from 5 to 2. The second example demonstrates how we can frame $y \mapsto 7$ away from the pre- and post-heaps in the prior example, because y is untouched by the program. The third example shows how, when we create a reference to the value 5, the post-heap must include some new address which points to 5.

2.2 Commutativity

Two operations *commute* when swapping their order of execution produces an equivalent final state. In practice, exact equality of final states is too restrictive, so we use *observational equivalence* (OE). Two states are observationally equivalent if, af-

ter any sequence of read/write operations permitted within the problem space, the results of any permitted observation (read) of each state will match. For instance, two unordered sets containing the same values are observationally equivalent, even if their values are arranged in memory in a different order.

Often, two operations will not *always* commute, but instead commute under a subset of initial conditions. In many settings that utilize commutativity, it's worth knowing this subset — all or none is less useful. Consider a state predicate P and equivalence relation on states $\sim$. We say two operations a and b commute up to $\sim$ under P, denoted $a \bowtie_P b$, if

$$\forall \sigma \ . \ P \ \sigma \Longrightarrow (a; b) \ \sigma \sim (b; a) \ \sigma.$$

If $a \bowtie_P b$ holds, we call P a *commutativity condition* for a and b.

2.3 Programs

For this work, we define *programs* to be functions of type `input vector -> heap -> heap`. We assume determinism and termination, so that programs are well-defined.

Under this definition, a standard formulation of commutativity would consider a commutativity condition on the initial heap and program inputs. That formulation might look like

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

for programs C_1 and C_2, and observational equivalence relation $\sim$ on the post-heaps. In the next section, we will develop a more nuanced definition of commutativity for heap programs. We will also use the following abbreviations for the post-heaps moving forward:

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

Chapter 3
Framed Commutativity

The goal of this section is to motivate and demonstrate our particular formula for commutativity of heap programs. Recall that we intend to introduce framing to commutativity reasoning; as such, our pre- and post-conditions for commutativity should be "frame-able".

A natural first attempt is to say that programs C_1 and C_2 commute under a precondition P if $\{P\}\ C_1\ \bar{x};\ C_2\ \bar{y}\ \{Q\}$, and $\{P\}\ C_2\ \bar{y};\ C_1\ \bar{x}\ \{Q\}$, and $\bar{x}$ and $\bar{y}$ satisfy some additional predicate. Under this formulation, we assert that post-heaps h_{12} and h_{21} are equivalent in that they both satisfy some carefully-chosen predicate Q.

However, this idea fails because it does not relate h_{12} and h_{21} precisely enough. If Q features existential quantification, then h_{12} and h_{21} will satisfy Q with independent sets of instantiated values, with no way to relate said values. If Q lacks existential quantification, then the two post-heaps must be identical, which is too restrictive.

Intuitively, the issue is that we need to reason about the pre- and post-heaps concretely, but heap triples abstract them away! So, our objective will be to construct a formula like

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

where P is the commutativity condition, $\sim$ is an equivalence relation, and $\bar{v}$, $\bar{r}$, and $\bar{s}$ are vectors of values which are somehow derived from the pre-heap and post-heaps respectively. The way we will bridge the gap between concrete heap reasoning and abstract vector reasoning is with a specially designed predicate called an abstraction predicate.

3.1 Abstraction Predicates

An *abstraction predicate* $\mathbf{\Psi}$ is a heap predicate parameterized on a vector of values called an *abstraction vector*. The goal when constructing $\mathbf{\Psi}$ for a particular problem space is to precisely encode abstract program state.

Abstraction predicates allow us to assess observational equivalence of post-heaps by designing an appropriate equivalence relation $\sim$ on abstraction vectors. If

h_{12} satisfies $\mathbf{\Psi}$ with vector $\bar{r}$, and h_{21} with $\bar{s}$, then the heaps are observationally equivalent if $\bar{r} \sim \bar{s}$.

The best way to understand the construction and use of abstraction predicates is through simple examples.

Non-Negative Counter

Consider the data structure of a non-negative counter, a sort of "hello world" problem in commutativity. Such a counter may be incremented, decremented (if we decrement while the counter is 0, the counter remains at 0), and read. If we store the counter value at address c, then $\mathbf{\Psi}$ and $\sim$ should be:

$$\mathbf{\Psi}\,\langle v\rangle \coloneqq c \mapsto v$$
$$\langle v_1\rangle \sim \langle v_2\rangle \coloneqq v_1 = v_2$$

Two-Set

Consider an unordered set that may contain at most two elements, to be stored at addresses p and q. These addresses contain null when unoccupied. We interact with the set through add, remove, and membership operations. This example demonstrates a nontrivial definition of observational equivalence through $\sim$.

$$\mathbf{\Psi}\,\langle u, v\rangle \coloneqq p \mapsto u * q \mapsto v * [u \neq v]$$
$$\langle u_1, v_1\rangle \sim \langle u_2, v_2\rangle \coloneqq \bigvee \begin{cases} u_1 = u_2 \wedge v_1 = v_2 \\ u_1 = v_2 \wedge v_1 = u_2 \end{cases}$$

With this construction, we capture the unordered equivalence of two post-heaps not at the heap predicate level, but at the abstraction vector level.

3.2 Precondition

We have now seen how to relate the post-heaps via abstraction predicates. How about the commutativity condition?

We will deliberately decompose the *precondition* for commutativity into two pieces: one which asserts executability of C_1 and C_2 on the concrete pre-heap, and

one which asserts commutativity at the abstraction vector level. The latter half, the *commutativity condition*, will be a predicate $\mathcal{P}$ on three vectors: the abstraction vector for the pre-heap, and the input vectors for C_1 and C_2.

3.3 Framed Commutativity Formula

Using abstraction predicates, observational equivalence on abstraction vectors, and preconditions, we can construct a formula for *framed commutativity*.

For some problem space, we consider an appropriately selected abstraction $\mathbf{\Psi}$ and equivalence relation $\sim$, two programs C_1 and C_2 operating in the space, and a candidate commutativity condition $\mathcal{P}$. We say C_1 and C_2 *commute* under $\mathcal{P}$, denoted $C_1 \bowtie_{\mathcal{P}} C_2$, if

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

Overall, this formula says: given any input vectors and an abstraction-compliant heap, if the commutativity condition is satisfied by said heap (via the abstraction predicate) and inputs, then the two post-heaps (from swapping the sequence of C_1 and C_2 with their appropriate inputs) are observationally equivalent via abstraction.

Additionally, notice the arbitrary heap predicate $\mathbf{H}$ starred onto the pre- and post-heap characterizing predicates. This construction allows for the framing away of superfluous pieces of state, i.e. local commutativity reasoning.

3.4 Complete Conditions

By taking the converse of the commutativity formula, we find the formula to describe a *complete* commutativity condition:

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

Intuitively, this formula says that $\mathcal{P}$ is complete if the observational equivalence of both post-heaps guarantees $\mathcal{P}$. Notice how the original precondition has been split here — we keep the abstraction on h in the hypothesis to ensure the proper execution

context, while moving the commutativity assertion to the conclusion.

An *exact* condition, one that is both correct and complete, will satisfy

$$\forall \mathbf{H}\ \bar{v}\ \bar{x}\ \bar{y}\ h\ .\ (\mathbf{\Psi}\ \bar{v} * \mathbf{H})\ h \Longrightarrow \left(\begin{array}{c} \mathcal{P}\ \bar{v}\ \bar{x}\ \bar{y} \\ \Updownarrow \\ \exists \bar{r}\ \bar{s}\ .\ (\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12} \wedge (\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21} \wedge \bar{r} \sim \bar{s} \end{array} \right)$$

As examples, consider the full suites of exact commutativity conditions for operations on the previously mentioned non-negative counter and two-set data structures. We specifically show the exact conditions, rather than correct or complete, because correct or complete conditions can be imprecise (for instance, $\bot$ is always correct and $\top$ is always complete). Note that the `read` and `mem` operations return values; these returns should also remain equivalent when execution order is swapped, for commutativity to hold. A formal treatment of this property is discussed in § 3.6.

Non-negative counter, with $\bar{v} \coloneqq \langle \mathtt{v} \rangle$

Commute Pair			$\mathcal{P}$
`incr()`	$\bowtie_{\mathcal{P}}$	`incr()`	$\top$
`incr()`	$\bowtie_{\mathcal{P}}$	`decr()`	`v > 0`
`incr()`	$\bowtie_{\mathcal{P}}$	`read()`	$\bot$
`decr()`	$\bowtie_{\mathcal{P}}$	`decr()`	$\top$
`decr()`	$\bowtie_{\mathcal{P}}$	`read()`	`v = 0`
`read()`	$\bowtie_{\mathcal{P}}$	`read()`	$\top$

Two-set, with $\bar{v} \coloneqq \langle \mathtt{u}, \mathtt{v} \rangle$

Commute Pair	$\mathcal{P}$
`add(x)` $\bowtie_{\mathcal{P}}$ `add(y)`	`x = y` $\vee$ `v = x` $\vee$ `v = y` $\vee$ (`u` = null $\wedge$ `v` = null) $\vee$ (`u` $\neq$ null $\wedge$ `v` $\neq$ null) $\vee$ (`v` $\neq$ null $\wedge$ (`u = x` $\vee$ `u = y`))
`add(x)` $\bowtie_{\mathcal{P}}$ `rem(y)`	(`x = y` $\wedge$ `u` $\neq$ null $\wedge$ `u` $\neq$ `x` $\wedge$ `v` $\neq$ `x`) $\vee$ (`x` $\neq$ `y` $\wedge$ (`u` = null $\vee$ `u = x` $\vee$ `v = x`)) $\vee$ (`x` $\neq$ `y` $\wedge$ `u` $\neq$ `y` $\wedge$ `v` $\neq$ `y`) $\vee$ (`u` = null $\wedge$ `v` $\neq$ null $\wedge$ `v` $\neq$ `x`) $\vee$ (`u` $\neq$ null $\wedge$ `v` = null $\wedge$ `u` $\neq$ `x`)
`add(x)` $\bowtie_{\mathcal{P}}$ `mem(y)`	`x` $\neq$ `y` $\vee$ `u = x` $\vee$ `v = x` $\vee$ (`u` $\neq$ null $\wedge$ `v` $\neq$ null)
`rem(x)` $\bowtie_{\mathcal{P}}$ `rem(y)`	$\top$
`rem(x)` $\bowtie_{\mathcal{P}}$ `mem(y)`	`x` $\neq$ `y` $\vee$ (`u` $\neq$ `x` $\wedge$ `v` $\neq$ `x`)
`mem(x)` $\bowtie_{\mathcal{P}}$ `mem(y)`	$\top$

3.5 The Linked Stack in Full

Let's return now to our introductory example of the linked stack, and build up our commutativity reasoning for it in full.

Recall that we intend to frame away untouched pieces of the stack, to simplify reasoning. To do so, we will create a *partial abstraction* for a *partial stack* (from the top down), as well as a *total* abstraction. The rule of thumb when creating a partial abstraction for a data structure is: frame away as much of the structure as possible, while maintaining that partial observational equivalence holds if and only if total observational equivalence holds.

Depending on the problem at hand, we want our abstraction to observe the stack to a desired depth. Specifically, the abstraction should only consider the values in the stack (top-down), in addition to the bottom-most pointer on the partial stack.

Thus, we will construct a recursive abstraction predicate $\mathbf{\Psi}_n$, which observes the top n values of the stack, and is parameterized on a list of values V and bottom-most pointer f.

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

$$\langle f_1, V_1 \rangle \sim \langle f_2, V_2 \rangle \coloneqq f_1 = f_2 \wedge (V_1 = V_2 \text{ element-wise})$$

There may be scenarios where reasoning about the entire stack is useful. In that case, we merely enforce that the bottom-most pointer on the partial stack is null, thereby making it a total stack:

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

Now, let's look at the full suite of exact commutativity conditions, accounting for the depths observed by $\boldsymbol{\Psi}$ in the pre-heap and post-heaps. Note that `peek` returns a value, which should remain equivalent when execution order is swapped for commutativity to hold. A formal treatment of this property is discussed in § 3.6.

Commute Pair	Pre-depth	Post-depth	V	$\mathcal{P}$
`push(x)` $\bowtie_{\mathcal{P}}$ `push(y)`	0	2	`[]`	`x = y`
`push(x)` $\bowtie_{\mathcal{P}}$ `peek()`	1	2	`[v]`	`v = x`
`push(x)` $\bowtie_{\mathcal{P}}$ `pop()`	1	1	`[v]`	`v = x`
`peek()` $\bowtie_{\mathcal{P}}$ `peek()`	1	1	`[_]`	$\top$
`peek()` $\bowtie_{\mathcal{P}}$ `pop()`	2	1	`[v1,v2]`	`v1 = v2`
`pop()` $\bowtie_{\mathcal{P}}$ `pop()`	2	0	`[_,_]`	$\top$

To understand the choices for pre- and post-depth more thoroughly, let's consider the pair `pop()` $\bowtie_{\mathcal{P}}$ `pop()` as an example. Because `pop` removes the top of the stack, our precondition must ensure that the stack contains at least two values, for removal to be possible twice; hence, our abstraction for the precondition uses a depth of 2. For the post-relation, we merely need to check that the two post-stacks share a common bottom pointer; hence, we use a depth of 0. Notice that the abstraction vector for the precondition contains a value list of length 2, but we disregard the values themselves as `pop` and `pop` always commute.

3.6 Return Value Lifting

In the above examples where some functions have return values, we've been imprecise with handling return value equivalence. Since our chosen model of programs lacks the notion of a return value, we need to hack our model a bit to allow for returns. Namely, the expected outcome of swapping execution order is that the return values of the individual functions remain the same, on top of the standard outcome that the final heaps are observationally equivalent.

We can reason about returns by *lifting* them to the abstraction vectors. We augment the abstraction $\mathbf{\Psi}$ and vector equivalence relation $\sim$ like so:

$$\mathbf{\Psi}' \ \langle v_1, v_2, \bar{v} \rangle \coloneqq \rho_1 \mapsto v_1 * \rho_2 \mapsto v_2 * \mathbf{\Psi} \ \bar{v}$$
$$\langle r_1, r_2, \bar{r} \rangle \sim' \langle s_1, s_2, \bar{s} \rangle \coloneqq r_1 = s_1 \wedge r_2 = s_2 \wedge \bar{r} \sim \bar{s}$$

where ρ_1 and ρ_2 are addresses not present in $\mathbf{\Psi}$. We then transform C_1 and C_2 as necessary to write their returns to ρ_1 and ρ_2 respectively.

3.7 Analysis

Framed commutativity is a simple and versatile method for reasoning about heap-modifying programs. Abstraction predicates offer a natural way to represent inductive data structures. In the next section, we will show how this framework translates neatly into the setting of automated reasoning.

We have come across challenges while developing framed commutativity. Because commutativity is partly a relational property, our model does not play nicely with standard heap triples out of the box. Similarly, framed commutativity assertions are not easily composable, which would be a desirable property for reasoning about complex programs. As this idea develops beyond a proof-of-concept, we believe relational separation logic may offer solutions to some of these challenges. Additional commentary is provided in § 6.1.

Chapter 4
Implementation in Coq

4.1 Infrastructure

Our implementation of heap-based commutativity reasoning is built atop the separation logic library in Volume 6 of *Software Foundations* [Charguéraud (2021)]. The library features a toy language with big-step semantics. Below are sample programs in the toy language, namely implementations of `push`, `pop`, and `peek` for the linked stack example.

```
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; () }>.
```

4.2 Developed Utilities

We have developed a suite of algorithms and tactics for automating commutativity proofs as much as possible.

Forward execution

A key step in the correctness proof process is big-step symbolic execution of programs. Symbolic execution is performed based on the inductively defined `eval` relation, which asserts that a piece of code transforms an initial heap into a final heap and returns a value.

Automating symbolic execution is generally straightforward: observe the type of `eval` expression in the goal, apply the appropriate inductive constructor, dispatch any side goals, and repeat.

The biggest challenge that appeared in the development of this procedure concerned the `ref` expression of the language. Allocating a reference requires that the

reference be fresh with respect to the initial heap. The language off-the-shelf lacks a procedure for *fresh* allocation. We introduced a stop-gap measure in the form of an abstract function $\texttt{frloc} : \texttt{heap} \to \texttt{loc}$, along with the axiom $\forall h \,.\, \texttt{frloc}(h) \notin h$. We use `frloc`, in an admittedly cumbersome manner, to permit the introduction of fresh addresses to the heap during forward execution.

Inverted execution

Inversion of program executions is necessary for proofs of completeness, as the `eval` statements describing the relationship between the initial and final heaps appear as hypotheses. But, because of how the semantics of the toy language are inductively defined in Coq, inverting an `eval` statement will sometimes produce multiple outcomes, where all but one are spurious. So we needed to be careful in the automated inversion procedure, and make sure that bad paths get killed as quickly as possible, thereby minimizing exponential blowup.

Goal dispatch

In the process of completing a commutativity proof, various goals pertaining to heaps pop up. We developed procedures and tactics for automating such goals as much as possible. Notable automated procedures include:

- Prove that a heap satisfies a heap predicate.
- Prove that two heaps are disjoint.
- Prove that an address is present (or absent) in a heap.

Hypothesis derivation

We must also derive non-trivial facts from various types of hypotheses throughout the proof process. Notable automated derivations include:

- Heaps are disjoint $\implies$ their addresses are unequal
- Heaps are equivalent $\implies$ values at the same address are equal
- Heap satisfies predicate $\implies$ heap contains particular addresses and values, etc.
- Heap contains some address $\implies$ heap contains some value at said address

4.3 Commutativity Formula

We encapsulate $C_1 \bowtie_{\mathcal{P}} C_2$ in Coq with the `comm` expression.

```
Definition hprop := heap -> Prop.
Definition hrel  := heap -> heap -> Prop.

Definition comm (c1 c2 : trm) (psi : hprop)
                (obseq : hprop -> hrel) :=
  forall H h,  (psi \* H) h ->
  exists h1 h2 v1 v2 h12 h21 v12 v21,
  eval h  c1 h1  v1   /\  eval h  c2 h2  v2   /\
  eval h1 c2 h12 v12  /\  eval h2 c1 h21 v21  /\
  obseq H h12 h21.
```

When constructing a commutativity assertion, the commutativity condition is placed outside of `comm` as either a hypothesis (for correctness) or conclusion (for completeness).

4.4 Correctness Proofs

Consider two programs `c1` and `c2`, a commutativity condition `P`, an abstraction predicate `psi`, and an equivalence relation on heaps `obseq`. The general structure for proving `P` correct is:

```
1  Example comm_c1_c2 :
2    forall v x y,
3    P v x y ->
4    comm (c1 x) (c2 y) (psi v) obseq.
5  Proof.

6    introv HP phi. unfolds R, psi, c1, c2.

7    destruct_heap phi.
8    (* Other derivations here *)

9    eexists*; splits; apply_evals.

10   eexists*.
11   splits; solve_hprop.
12   (* Solve r ~ s here *)

13 Qed.
```

The key pieces of this proof, in reference to their line numbers, are:

2: Universally quantify on all initial abstraction vectors and program inputs.

3: Assert the problem-specific commutativity condition as a hypothesis.

4: Assert commutativity as the conclusion, using the abstraction predicate and observational equivalence.

7: Derive facts about the initial heap h from the hypothesis $(\mathbf{\Psi}\ \bar{v} * \mathbf{H})\ h$.

8: Derive additional facts about the initial heap and program inputs from the commutativity condition.

9: Perform symbolic execution on each of the 4 `eval` statements within `comm`. This finds us the values of the post-heaps h_{12} and h_{21}.

10: Instantiate the abstraction vectors $\bar{r}$ and $\bar{s}$ in the post-relation.

11: Dispatch $(\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12}$ and $(\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21}$.

12: Dispatch the problem-specific equivalence of $\bar{r}$ and $\bar{s}$.

We will see concrete examples of correctness proofs in upcoming sections.

4.5 Completeness Proofs

Consider again the programs, abstraction predicate, commutativity condition, and equivalence relation on heaps. The general structure for proving `P` complete is:

```
1  Example comm_c1_c2_inv:
2    forall v x y,
3    comm (c1 x) (c2 y) (psi v) obseq ->
4    P v x y.
5  Proof.

6    introv C.

7    assert ((psi v) (* h *)) as H. { solve_hprop. }

8    apply_comm C H.
9    unfolds obseq, psi, c1, c2.

10   invert_evals.

11   destruct_heap HR1, HR2.
12   (* Derive facts from r ~ s here *)

13   (* Dispatch P here *)

14 Qed.
```

The key pieces of this proof, in reference to their line numbers, are:

2: Universally quantify on all initial abstraction vectors and program inputs.

3: Assert commutativity as the hypothesis, using the abstraction predicate and observational equivalence.

4: Assert the problem-specific commutativity condition in the conclusion.

7: Construct a problem-specific initial heap h that satisfies $\mathbf{\Psi}\ \bar{v}$.

8: Instantiate `comm` with h and the initial heap and **emp** as **H**.

10: Perform inverted symbolic execution on each of the 4 `eval` statements within `comm`. This finds us the values of the post-heaps h_{12} and h_{21}.

11: Derive facts about h_{12} and h_{21} from $(\mathbf{\Psi}\ \bar{r} * \mathbf{H})\ h_{12}$ and $(\mathbf{\Psi}\ \bar{s} * \mathbf{H})\ h_{21}$.

12: Derive problem-specific facts from the equivalence of $\bar{r}$ and $\bar{s}$.

13: Perform a problem-specific dispatch of the commutativity condition.

4.6 Condition Inference

In some scenarios, we can leverage the interactivity of Coq proofs to work towards inferring commutativity conditions. The procedure is simple: begin a correctness proof by assuming $\mathcal{P} = \top$, and see where and why the proof gets stuck!

Consider the example of two unary functions, $f(x) \coloneqq x^2+1$ and $g(x) \coloneqq 3x+2$. When does the application of these functions commute? In other words, for what inputs x does $(f \circ g)(x) = (g \circ f)(x)$?

```
Parameter p : loc.
Implicit Type x : int.

Example psi x :=
  p ~~> x.

Example obseq H h1 h2 :=
  exists x,
  (psi x \* H) h1 /\
  (psi x \* H) h2.
```

```
Example f :=
  <{ let 'x = ! p in
     let 'a = 'x * 'x in
     let 'b = 'a + 1 in
     p := 'b; () }>.

Example g :=
  <{ let 'x = ! p in
     let 'a = 3 * 'x in
     let 'b = 'a + 2 in
     p := 'b; () }>.
```

We try to prove commutativity with the assumption that $\mathcal{P} \coloneqq \top$. We'll fail when trying to prove equivalence of the post-heaps.

```
Lemma comm_f_g : forall x0,
  comm f g (psi x0) obseq.
Proof.
  introv. introv Psi.
  unfolds obseq, psi, f, g.
  destruct_heap Psi. simpl_disjoint.
  solve_comm.
  eexists*. split; solve_hprop.
  clean_vals.
Abort.
```

Before aborting the proof, we have the goal

```
9 * Z.pow_pos x0 2 + 12 * x0 + 4 + 1   =   3 * Z.pow_pos x0 2 + 3 + 2
```

The exact commutativity condition for f and g is indeed the solution to this equation: $x_0 = 0$ or $x_0 = -2$. The correctness proof for our new-found $\mathcal{P}$ looks like this:

```
Lemma comm_f_g : forall x0,
  x0 = 0 \/ x0 = -2 ->
  comm f g (psi x0) obseq.
Proof.
  introv P Psi. unfolds obseq, psi, f, g.
  destruct_heap Psi. simpl_disjoint.
  solve_comm.
  (* Destruct P before existential instantiation *)
  destruct P; subst.
  all: eexists*; split; solve_hprop.
Qed.
```

4.7 Preemptive Guard Destruction

If we attempt to conduct a commutativity proof with a program that contains an if-then-else branch, we will likely fail. This is due to the path sensitivity problem.

A naïve solution is to destruct on every branch guard *before* instantiating existentials. This strategy is inconvenient at best and intractable at worst, as it requires performing a manual symbolic execution to anticipate all possible guard evaluations.

Consider the example of proving that for a two-set, `mem` $\bowtie_\top$ `mem`.

```
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; () }>.
```

```
1  Example comm_mem_mem :
2    forall (u v x y : int) (t1 t2 : bool),
3    x >= 0  ->  y >= 0 ->
4    comm (mem rho1 x) (mem rho2 y)
5         (psi u v t1 t2) obseq.
6  Proof.

7    introv Zx Zy S. unfolds mem, obseq, psi.
8    destruct_heap S. simpl_disjoint.

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

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

13   all: solve_comm.
14   all: eexists*; splits; solve_hprop.
15   all: auto.

16 Qed.
```

Even though the programs always commute, we still need to account for branching, by destructing on the truth or falsity of the two guards in each program (lines 9-10). After this series of destructions, we first try to kill any paths which lead to immediate contradiction (11-12), then perform up to $2^{\#\text{guards}}$ proofs in parallel (13-15).

Additionally, note that line 3 is not a commutativity condition, but simply a constraint on the domains of the inputs passed to `mem`.

4.8 Evaluation

Our implementation of heap-based commutativity in Coq enables largely automated proofs of the correctness and completeness of commutativity conditions, in a setting natural for inductive structures. Interactive inference of conditions is also possible in some scenarios.

The current implementation features a few key challenges worth studying further. For one, proofs require full big-step symbolic execution. This is tractable for the simple examples we have attempted so far, but may cause problems in more complex programs. In particular, it will be ineffective for programs containing loops or

recursion, which are domains we intend to explore in the future.

Additionally, our naïve solution to the path sensitivity problem is impractical, as it produces exponential blowup in proof size, and requires the programmer to reason through what every branch guard expressions will be by hand.

Chapter 5
Veracity

Veracity is a programming language first introduced in Chen et al. (2022), which features the novel construct of the `commute` block. This block allows programmers to express fragments of code in a sequential manner, for them to be parallelized by the Veracity toolchain using commutativity analysis.

5.1 Commute Blocks

The general structure of the `commute` block is as follows:

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

The `guard` serves as the commutativity condition for blocks `f1` and `f2`. As an example, here are the increment and decrement operations on a non-negative counter, with an exact commutativity condition:

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

The `commute` block has two equivalent semantics, dependent on the evaluation of the guard at runtime:

1. If `guard` $\rightsquigarrow$ `false`, follow sequential semantics: `f1; f2`
2. If `guard` $\rightsquigarrow$ `true`, follow concurrent semantics: run `f1` and `f2` in parallel (with Multicore OCaml)

We have implemented an interpreter and runtime for Veracity in OCaml, with a toolchain involving the Servois analysis tool to infer and verify correct guards (commutativity conditions) for `commute` blocks.

5.2 Heap Integration

As we've seen, heap-based commutativity is a promising model for local commutativity reasoning, particularly for inductive data structures. Reasoning about inductive structures is a feature that Veracity presently lacks.

There are a few potential ways to integrate heap-based reasoning into the Veracity ecosystem. These possibilities are aided by the fact that the Veracity interpreter already encodes state as a mapping from variables to values, a representation easily translatable into a heap model. The possibilities under consideration are:

1. Embed separation logic in Veracity's condition inference/verification toolchain.
2. Export Coq theories into the Veracity interpreter.
3. Translate Veracity programs from OCaml into Coq.

Chapter 6
Conclusion

We have introduced a promising framework, based on separation logic, for simplified reasoning about the commutativity of heap-modifying programs. Namely, we use abstraction predicates to frame away unmodified portions of the heap, permitting local commutativity reasoning and simplified analysis of inductive structures.

We have also demonstrated a successful implementation of this framework in Coq, enabling largely automated proofs of the correctness and completeness of commutativity conditions. This implementation may also prove a valuable tool for integrating heap-based reasoning into the Veracity programming language.

6.1 Discussion

A few aspects of the theoretic framework established in this paper deserve further attention. Firstly, the determinism requirement for programs is too strict, especially because pointer allocation is generally nondeterministic. We expect that adjusting the theoretic framework and Coq implementation to account for this change will require little hassle.

Secondly, termination is also too strict a requirement, since program execution is undefined when addresses to be read/written are missing from the initial heap. A more accurate description of our current program model is that termination is guaranteed when the initial heap satisfies the abstraction predicate.

The framed commutativity formula may also benefit from a transformation into a framed commutativity *rule*, where the framed predicate **H** is introduced in the conclusion, mirroring the original frame rule.

Finally, it will be worth studying in more detail examples where the abstraction predicate in the pre-condition differs from that in the post-relation. Distinct pre- and post-abstractions were used in the linked stack example, and while it was applied carefully in that case, it technically disagrees with the framed commutativity formula. Another scenario justifying distinct abstractions would be one with programs which transform one data structure into another. The requirement that both abstractions match may indeed be needlessly strict.

6.2 Future Work

We have already discussed some future considerations throughout the paper, such as exploring relational separation logic (§ 3.7), attempting to reason with loops and recursion in Coq (§ 4.8), and integrating heap reasoning into Veracity (§ 5.2). Aside from these plans, and the ideas conveyed in the discussion above, there are a variety of other directions to take this work moving forward.

One objective is to try reasoning inductively, rather than finitely, about inductive structures. For example, we reasoned about linked stacks by observing a constant number of elements (§ 3.5). Imagine in contrast a case where we reason about a tree, and, for instance, frame away all but the left-most branch. Developing abstraction predicates and equivalence relations in both cases is similar; what may differ is the construction of proofs.

We also intend to explore interactive commutativity condition inference more thoroughly. Interactive proofs may offer new insights into condition inference, an important problem in the commutativity space.

Bibliography

Bansal, K., E. Koskinen, and O. Tripp (2018). Automatic generation of precise and useful commutativity conditions (extended version). *CoRR abs/1802.08748*.

Charguéraud, A. (2021). Separation logic foundations. In *Software Foundations*, Volume 6. University of Pennsylvania.

Chen, A., P. Fathololumi, E. Koskinen, and J. Pincus (2022). Veracity: Declarative multicore programming with commutativity.

Clements, A. T., M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler (2013). The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, pp. 1–17. Association for Computing Machinery.

Herlihy, M. and E. Koskinen (2008). Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, New York, NY, USA, pp. 207–216. Association for Computing Machinery.

Kim, D. and M. C. Rinard (2011). Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, New York, NY, USA, pp. 528–541. Association for Computing Machinery.

Pîrlea, G., A. Kumar, and I. Sergey (2021). *Practical Smart Contract Sharding with Ownership and Commutativity Analysis*, pp. 1327–1341. New York, NY, USA: Association for Computing Machinery.

Reynolds, J. (2002). Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74.