Enabling Modular Proofs of Correctness for Parallel Programs Using Annotated Abstract Data Types

Alan Weide[1], Paolo A. G. Sivilotti[1], and Murali Sitaraman[2]

[1] The Ohio State University, Columbus OH 43221, USA,
weide.3@osu.edu, paolo@cse.ohio-state.edu
[2] Clemson University, Clemson SC 29634, USA,
murali@clemson.edu

Abstract. Parallel programs using abstract data types are notoriously difficult to formally show correct in a modular manner because their behavior depends upon the implementations of those types in addition to the abstract specifications. A non-interference specification framework and related calculus is introduced to enable the modular verification of correctness of parallel programs using annotated abstract data types. Under the specification framework, non-interfering statements commute. This fact is proven and used to show the soundness of a proof rule for fork-join parallel programs.

1 Introduction

Parallel programs are notoriously difficult to formally show correct because of the possiblity of interference among threads. The formal verification of such programs typically involve reasoning directly about the implementation of the objects in use. However, complex software components in the real world are built in a layered fashion, reusing complex "high-level" components (which themselves are built using other high-level components). In a sequential context, verification of the correctness of these programs typically proceeds in a modular fashion, using the *behavioral specification*—and not the implementation!—of the underlying components. Unfortunately, such modular verification breaks down in a concurrent context because the underlying implementation matters somewhat. For example, it is easy to imagine two implementations of a queue—one that becomes corrupted when multiple processes attempt to concurrently enqueue and one that will handle it with grace—both of which respect some reasonable (sequential) behavioral specification. Of course, only the second implementation would be useful in a concurrent system.

A behavioral specification can be enhanced with a *non-interference specification* [**?**,**?**] to capture the conditions under which it is safe to use a component in a parallel context. The verification that an implementation meets its non-interference specification proceeds modularly and without sacrificing abstraction. Moreover, the non-interference specification does not require modification to the sequential behavioral specification.

In this paper, it is shown that the manner in which an implementation meets its non-interference specification and is considered "safe" implies that the parallel execution of non-interfering statements is equivalent to all possible sequential orderings of the statements.

1.1 Outline

In section 2, a caluclus for effects is defined. Effects are related to programs in section 3, and the semantics of a programming language with respect to effects is formalized in section 4. Finally, the various results achieved by this calculus and programming model are proved in section 5.

2 A/P Calculus

A non-interference specification identifies, for each operation on an object, under what conditions various pieces of the object's state are *affected* (*i.e.*, might have their value changed), *preserved* (*i.e.*, might have their value read), or *ignored* (*i.e.*, are neither read nor written to) by the execution of that operation. This section introduces a calculus to facilitate formal reasoning about these effects independent of their use in programs. Despite the apparent generality in the calculus, effects find their primary use in reasoning about programs involving partitioned objects, in which some pieces of each object are *affected* (corresponding to the 'A' set) and others are *preserved* (the 'P' set). (Pieces of an object that are *ignored* by an effect are not in either the A or P sets of that effect.)

For example, consider the C-language program fragment in listing 2.1. The effect of the method `modifyX` might be written in a non-interference specification as "*affects* `p.x`; *preserves* `p.y`, `p.z`". In the effects calculus, it would be summarized as the pair $(\{p.x\}, \{p.y, p.z\})$. That is, the method `modifyX` might change the value of `p.x` and might read the value of `p.y` and `p.z`. Of course, objects in real-world programs are typically far more complex that the `Point` struct in listing 2.1. The A/P calculus is general enough to serve as a useful reasoning tool both for simple structures such as `Point` and for complex objects with rich abstraction of the kind used in real-world programs. This generality is applied through the mechanisms described in sections 3 and 4.

2.1 Definitions

Definition 2.1 (Effect). *An* effect *is a pair of sets* $e = (A, P)$.

Definition 2.2 (Target of an effect). *The* target *of an effect* $e = (A, P)$ *is* $\mathcal{T}(e) = A \cup P$.

Definition 2.3 (Projection). *The* projection *of an effect* $e = (A, P)$ *to a set* S *is* $e|_S = (A \cap S, P \cap S)$.

Listing 2.1. C program fragment to illustrate a simple use of effects.

```
struct Point {
  int x;
  int y;
  int z;
};

void modifyX(struct Point *p) {
  p.x = p.x + p.y + p.z;
}
```

Definition 2.4 (Reduction). *The* reduction *of an effect $e = (A, P)$, denoted $\mathcal{R}(e)$, is defined as follows.*

$$\mathcal{R}(e) = (A, P \setminus A)$$

(When the need arises, we write $\mathcal{R}_S(e)$ when we mean $\mathcal{R}(e)|_S$.)

Definition 2.5 (Equivalence Modulo Reduction). *The relation $\equiv_{\mathcal{R}}$, defined as follows, is an equivalence relation:*

$$e_1 \equiv_{\mathcal{R}} e_2 \Leftrightarrow \mathcal{R}(e_1) = \mathcal{R}(e_2)$$

The equivalence class of effects with respect to $\equiv_{\mathcal{R}}$ for some effect e is denoted $[e]_{\mathcal{R}}$ (or just $[e]$ when $\mathcal{R}$ is understood). The canonical representation *of an equivalence class is the effect e within that class such that $\mathcal{R}(e) = e$.*

Definition 2.6 (Combined effect). *The* combined effect *of two effects $e_1 = (A_1, P_1)$ and $e_2 = (A_2, P_2)$, denoted $e_1 \sqcup e_2$, is defined as follows.*

$$e_1 \sqcup e_2 = \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big)$$

Definition 2.7 (Common effect). *The* common effect *of two effects $e_1 = (A_1, P_1)$ and $e_2 = (A_2, P_2)$, denoted $e_1 \sqcap e_2$, is defined as follows.*

$$e_1 \sqcap e_2 = \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big)$$

Definition 2.8 (The Is Covered By relation). *The* is covered by *relation $\sqsubseteq$ on effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$ is defined as follows.*

$$e_1 \sqsubseteq e_2 \Leftrightarrow (A_1 \subseteq A_2) \wedge \big(\mathcal{T}(e_1) \subseteq \mathcal{T}(e_2)\big)$$

Definition 2.9 (The Does Not Interfere With relation). *The* does not interfere with *relation $\ddagger$ on effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$ is defined as follows.*

$$e_1 \ddagger e_2 \Leftrightarrow \big(A_1 \cap \mathcal{T}(e_2) = \emptyset\big) \wedge \big(A_2 \cap \mathcal{T}(e_1) = \emptyset\big)$$

Definition 2.10 (Well-formedness). *An effect $e = (A, P)$ is* well-formed *if $A \cap P = \emptyset$. e is* well-formed with respect to *a set S if e is well-formed and $\mathcal{T}(e) \subseteq S$.*

2.2 Lemmas

Lemma 2.1. *For effect $e = (A, P)$, $\mathcal{T}\big(\mathcal{R}(e)\big) = \mathcal{T}(e)$.*

Proof.

$$\begin{aligned}
\mathcal{T}\big(\mathcal{R}(e)\big) &= \mathcal{T}\big((A, P \setminus A)\big) && \text{(def. of } \mathcal{R}) \\
&= A \cup (P \setminus A) && \text{(def. of } \mathcal{T}) \\
&= A \cup P && \text{(appl. of } \cup, \setminus) \\
&= \mathcal{T}(e) && \text{(def. of } \mathcal{T})
\end{aligned}$$

□

Lemma 2.2. *For effects e, $\mathcal{R}(e) \sqsubseteq e$.*

Proof. Let $\mathcal{R}(e) = (A_{\mathcal{R}}, P_{\mathcal{R}})$ and $e = (A, P)$.

$$\begin{aligned}
& A_{\mathcal{R}} = A \wedge \mathcal{T}\big(\mathcal{R}(e)\big) = \mathcal{T}(e) && \text{(def. of } \mathcal{R}\text{, lemma 2.1)} \\
\Rightarrow\ & A_{\mathcal{R}} \subseteq A \wedge \mathcal{T}\big(\mathcal{R}(e)\big) \subseteq \mathcal{T}(e) && \text{(appl. of } \subseteq) \\
\Rightarrow\ & \mathcal{R}(e) \sqsubseteq e && \text{(def. of } \sqsubseteq)
\end{aligned}$$

□

Lemma 2.3. *For effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$,*

$$\big(e_1 \sqcup \mathcal{R}(e_2) = e_1 \sqcup e_2\big)$$

and

$$\big(e_1 \sqcap \mathcal{R}(e_2) = e_1 \sqcap e_2\big).$$

Proof.

$$\begin{aligned}
e_1 \sqcup \mathcal{R}(e_2) &= \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(\mathcal{R}(e_2)))\big) && \text{(def. of } \sqcup) \\
&= \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(lemma 2.1)} \\
&= e_1 \sqcup e_2 && \text{(def. of } \sqcup)
\end{aligned}$$

$$\begin{aligned}
e_1 \sqcap \mathcal{R}(e_2) &= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(\mathcal{R}(e_2)))\big) && \text{(def. of } \sqcap) \\
&= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) && \text{(lemma 2.1)} \\
&= e_1 \sqcap e_2 && \text{(def. of } \sqcap)
\end{aligned}$$

□

Lemma 2.4. *For effect $e = (A, P)$, $\mathcal{R}(e)$ is well-formed.*

Proof.

$$\begin{aligned} & A \cap (P \setminus A) = \emptyset && \text{(appl. of } \setminus, \cap) \\ & \quad \Rightarrow (A, P \setminus A) \text{ is well-formed} && \text{(def. of well-formedness)} \\ & \quad \Rightarrow \mathcal{R}(e) \text{ is well-formed} && \text{(def. of } \mathcal{R}) \end{aligned}$$

□

Lemma 2.5. *Well-formedness is closed under* $|$, $\mathcal{R}$, $\sqcup$, *and* $\sqcap$.

Proof. We show each individually.

Claim. Well-formedness is closed under $|$. That is, for any well-formed effect $e = (A, P)$ and set S, $e|_S$ is well-formed.

Proof.

$$\begin{aligned} & e|_S = (A \cap S, P \cap S) && \text{(def. of } |) \\ & \quad \Rightarrow (A \cap S) \cap (P \cap S) = (A \cap P) \cap (S \cap S) && \text{(assoc., commut. of } \cap) \\ & \quad \Rightarrow (A \cap S) \cap (P \cap S) = \emptyset \cap (S \cap S) && \text{(well-formedness of } e) \\ & \quad \Rightarrow (A \cap S) \cap (P \cap S) = \emptyset && \text{(appl. of } \cap) \\ & \quad \Rightarrow (A \cap S, P \cap S) \text{ is well-formed} && \text{(def. of well-formedness)} \\ & \quad \Rightarrow e|_S \text{ is well-formed} && \text{(def. of } |) \end{aligned}$$

■

Claim. Well-formedness is closed under $\mathcal{R}$. That is, for any well-formed effect e, $\mathcal{R}(e)$ is well-formed.

Proof. By lemma 2.4, the reduction of any effect (including well-formed ones) is well-formed. Therefore, well-formedness is closed under $\mathcal{R}$. ■

Claim. Well-formedness is closed under $\sqcup$ and $\sqcap$. That is, for any well-formed effects e_1, e_2, $e_1 \sqcup e_1$ and $e_1 \sqcap e_2$ are both well-formed.

Proof. By definitions 2.6 and 2.7, $e_1 \sqcup e_2$ and $e_1 \sqcap e_2$ are each the result of applying $\mathcal{R}$ to some effect. By lemma 2.4, the reduction of any effect is well-formed. Therefore well-formedness is closed under $\sqcup$ and $\sqcap$. ■

□

Lemma 2.6. *The* $\ddagger$ *relation is symmetric. That is, for effects* $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$, $e_1 \ddagger e_2 \Leftrightarrow e_2 \ddagger e_1$.

Proof.

$$\begin{aligned} & e_1 \ddagger e_2 \\ & \quad \Leftrightarrow \big(A_1 \cap \mathcal{T}(e_2) = \emptyset\big) \wedge \big(A_2 \cap \mathcal{T}(e_1) = \emptyset\big) && \text{(def. of } \ddagger) \\ & \quad \Leftrightarrow \big(A_2 \cap \mathcal{T}(e_1) = \emptyset\big) \wedge \big(A_1 \cap \mathcal{T}(e_2) = \emptyset\big) && \text{(commut. of } \wedge) \\ & \quad \Leftrightarrow e_2 \ddagger e_1 && \text{(def. of } \ddagger) \end{aligned}$$

□

Lemma 2.7. *For effects* $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$,

$$e_1 \ddagger e_2 \wedge e_3 \sqsubseteq e_1 \Rightarrow e_3 \ddagger e_2$$

Proof.

$$\begin{aligned}
&e_1 \ddagger e_2 \wedge e_3 \sqsubseteq e_1 \\
&\quad \Rightarrow A_1 \cap \mathcal{T}(e_2) = \emptyset \wedge A_2 \cap \mathcal{T}(e_1) = \emptyset \wedge e_3 \sqsubseteq e_1 && \text{(def. of } \ddagger\text{)} \\
&\quad \Rightarrow \begin{pmatrix} A_1 \cap \mathcal{T}(e_2) = \emptyset \wedge A_2 \cap \mathcal{T}(e_1) = \emptyset \wedge \\ A_3 \subseteq A_{e_1} \wedge \mathcal{T}(e_3) \subseteq \mathcal{T}(e_1) \end{pmatrix} && \text{(def. of } \sqsubseteq\text{)} \\
&\quad \Rightarrow A_3 \cap \mathcal{T}(e_2) = \emptyset \wedge A_2 \cap \mathcal{T}(e_3) = \emptyset && \text{(appl. of } \subseteq\text{, substitution)} \\
&\quad \Rightarrow e_3 \ddagger e_2 && \text{(def. of } \ddagger\text{)}
\end{aligned}$$

□

Lemma 2.8. *For effects* $e_1 = (A_1, P_1)$, $e_1 = (A_2, P_2)$,

$$e_1 \sqsubseteq e_2 \Leftrightarrow \mathcal{R}(e_1) \sqsubseteq \mathcal{R}(e_2)$$

Proof. Let $\mathcal{R}(e_1) = (A_{\mathcal{R}_1}, P_{\mathcal{R}_1})$ and $\mathcal{R}(e_2) = (A_{\mathcal{R}_2}, P_{\mathcal{R}_2})$. Then by definition 2.4 and lemma 2.1, we have:

$$(A_{\mathcal{R}_1} = A_1) \wedge (A_{\mathcal{R}_2} = A_2) \wedge \big(\mathcal{T}(\mathcal{R}(e_1)) = \mathcal{T}(e_1)\big) \wedge \big(\mathcal{T}(\mathcal{R}(e_2)) = \mathcal{T}(e_2)\big).$$

$$\begin{aligned}
&e_1 \sqsubseteq e_2 \\
&\quad \Leftrightarrow A_1 \subseteq A_2 \wedge \mathcal{T}(e_1) \subseteq \mathcal{T}(e_2) && \text{(def. of } \sqsubseteq\text{)} \\
&\quad \Leftrightarrow A_{\mathcal{R}_1} \subseteq A_{\mathcal{R}_2} \wedge \mathcal{T}(e_1) \subseteq \mathcal{T}(e_2) && \text{(substitution)} \\
&\quad \Leftrightarrow A_{\mathcal{R}_1} \subseteq A_{\mathcal{R}_2} \wedge \mathcal{T}\big(\mathcal{R}(e_1)\big) \subseteq \mathcal{T}\big(\mathcal{R}(e_2)\big) && \text{(substitution)} \\
&\quad \Leftrightarrow \mathcal{R}(e_1) \sqsubseteq \mathcal{R}(e_2) && \text{(def. of } \mathcal{R}\text{)}
\end{aligned}$$

□

Lemma 2.9. *For a function* $f : D \to R$*, let* $F : \wp(D) \to \wp(R)$ *be defined as* $F(X) = \{f(x) : x \in X\}$*, the element-wise application of* f *to* X*, a subset of* D*. Then for all* A_1, P_1, A_2, P_2 *such that* $A_1, P_1, A_2, P_2 \subseteq D$ *and all* $f : D \to R$ *for some* R*,*

$$\big(F(A_1), F(P_1)\big) \ddagger \big(F(A_2), F(P_2)\big) \Rightarrow (A_1, P_1) \ddagger (A_2, P_2).$$

Proof. We show the contrapositive, *i.e.*,

$$\neg\big((A_1, P_1) \ddagger (A_2, P_2)\big) \Rightarrow \neg\big(\big(F(A_1), F(P_1)\big) \ddagger \big(F(A_2), F(P_2)\big)\big).$$

$$
\begin{aligned}
&\neg\big(\big((A_1, P_1) \ddagger (A_2, P_2)\big)\big) \\
&\quad \Rightarrow \exists x : x \in \mathcal{T}\big((A_1, P_1)\big) \wedge x \in A_2 && \text{(def. of } \ddagger\text{, appl. of } \neg) \\
&\quad \Rightarrow \exists x : x \in \mathcal{T}\big((A_1, P_1)\big) \wedge x \in A_2 \wedge f(x) \in \mathcal{T}\big((F(A_1), F(P_1))\big) \wedge f(x) \in F(A_2) \\
&&& \text{(def. of } F\text{)} \\
&\quad \Rightarrow \neg\big((F(A_1), F(P_1)) \ddagger (F(A_2), F(P_2))\big) && \text{(def. of } \ddagger\text{)}
\end{aligned}
$$

Therefore, $\big(F(A_1), F(P_1)\big) \ddagger \big(F(A_2), F(P_2)\big) \Rightarrow \big((A_1, P_1) \ddagger (A_2, P_2)\big)$. □

Remark 1. The converse of the conditional in lemma 2.9 is *not* true because when $(A_1, P_1) \ddagger (A_2, P_2)$ there might be some x, y such that $x \neq y \wedge x \in A_1 \wedge y \in A_2$ but $f(x) = f(y)$, and thus that $\neg\big((F(A_1), F(P_1)) \ddagger (F(A_2), F(P_2))\big)$.

2.3 The A/P Lattice

Beginning in this section we adopt the following definitions, which are used for the remainder of this paper.

- S is some set
- $E_S = \{e : e \text{ is a well-formed effect with respect to } S\}$ (when S is understood or irrelevant, it is left out and we refer simply to E)
- $\bot = (\emptyset, \emptyset)$
- $\top_S = (S, \emptyset)$ (when S is understood or irrelevant, it is left out and we refer simply to $\top$)
- $\mathfrak{L}_S = (E_S, \sqcup, \sqcap, \bot, \top_S)$ (when S is understood or irrelevant, it is left out and we refer simply to $\mathfrak{L}$)

While we have defined $\mathfrak{L}$ here only for well-formed effects, it could be similarly defined for $[e]_{\mathcal{R}}$, the equivalence class of e under $\equiv_{\mathcal{R}}$, and indeed that formulation might be preferred for some applications.

Theorem 2.1 (Well-formed effects form a lattice). *For all S, $\mathfrak{L}_S$ is a bounded lattice over E_S with operations $\sqcup$ and $\sqcap$, least element $\bot$, and greatest element $\top_S$.*

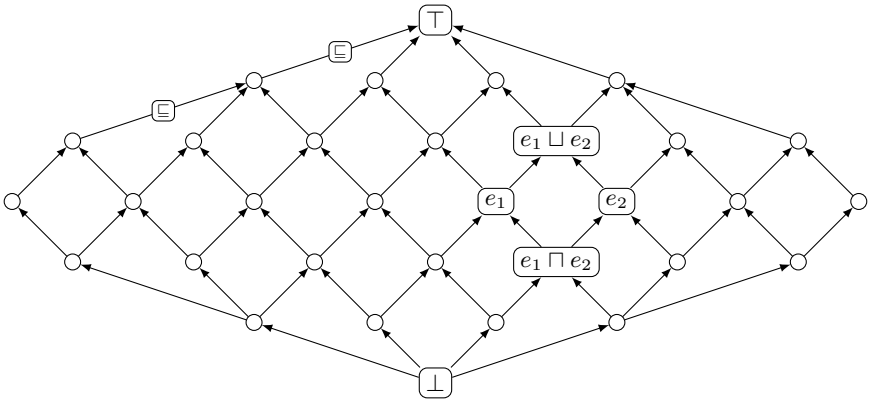
Proof. We show that $(E, \sqcup, \sqcap)$ is a lattice, then show that $\bot$ and $\top$ are its bounds.

First, by lemma 2.4, E is closed under $\sqcup$ and $\sqcap$.[3]

Claim. The operation $\sqcup$ is associative.

[3] The lemma actually states that the set of *all* well-formed effects is closed under those operations. The proofs that $\mathcal{T}(e_1 \sqcup e_2) \subseteq S$ and that $\mathcal{T}(e_1 \sqcap e_2) \subseteq S$ for $e_1, e_2 \in E_S$ are trivial and left as an exercise to the reader.

Fig. 2.1. The effects lattice $\mathfrak{L}$.



Proof. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$:

$$
\begin{aligned}
& e_1 \sqcup (e_2 \sqcup e_3) \\
&\quad = e_1 \sqcup \mathcal{R}\big((A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3))\big) && \text{(def. of } \sqcup) \\
&\quad = e_1 \sqcup \big(A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3)\big) && \text{(lemma 2.3)} \\
&\quad = \mathcal{R}\big((A_1 \cup (A_2 \cup A_3), \mathcal{T}(e_1) \cup (\mathcal{T}(e_2) \cup \mathcal{T}(e_3)))\big) && \text{(def. of } \sqcup) \\
&\quad = \mathcal{R}\big(((A_1 \cup A_2) \cup A_3, (\mathcal{T}(e_1) \cup \mathcal{T}(e_2)) \cup \mathcal{T}(e_3))\big) && \text{(assoc. of } \cup) \\
&\quad = \big(A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2)\big) \sqcup e_3 && \text{(def. of } \sqcup) \\
&\quad = \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) \sqcup e_3 && \text{(lemma 2.3)} \\
&\quad = (e_1 \sqcup e_2) \sqcup e_3 && \text{(def. of } \sqcup)
\end{aligned}
$$

■

Claim. The operation $\sqcup$ is commutative.

Proof. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$:

$$
\begin{aligned}
e_1 \sqcup e_2 &= \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(def. of } \sqcup) \\
&= \mathcal{R}\big((A_2 \cup A_1, \mathcal{T}(e_2) \cup \mathcal{T}(e_1))\big) && \text{(commut. of } \cup) \\
&= e_2 \sqcup e_1 && \text{(def. of } \sqcup)
\end{aligned}
$$

■

Claim. The operation $\sqcap$ is associative.

Proof. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$:

$$\begin{aligned}
&e_1 \sqcap (e_2 \sqcap e_3) \\
&\quad = e_1 \sqcap \mathcal{R}\big((A_2 \cap A_3, \mathcal{T}(e_2) \cap \mathcal{T}(e_3))\big) && \text{(def. of } \sqcap) \\
&\quad = e_1 \sqcap \big(A_2 \cap A_3, \mathcal{T}(e_2) \cap \mathcal{T}(e_3)\big) && \text{(lemma 2.3)} \\
&\quad = \mathcal{R}\big((A_1 \cap (A_2 \cap A_3), \mathcal{T}(e_1) \cap (\mathcal{T}(e_2) \cap \mathcal{T}(e_3)))\big) && \text{(def. of } \sqcap) \\
&\quad = \mathcal{R}\big(((A_1 \cap A_2) \cap A_3, (\mathcal{T}(e_1) \cap \mathcal{T}(e_2)) \cap \mathcal{T}(e_3))\big) && \text{(assoc. of } \cap) \\
&\quad = \big(A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2)\big) \sqcap e_3 && \text{(def. of } \sqcap) \\
&\quad = \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) \sqcap e_3 && \text{(lemma 2.3)} \\
&\quad = (e_1 \sqcap e_2) \sqcap e_3 && \text{(def. of } \sqcap)
\end{aligned}$$

■

Claim. The operation $\sqcap$ is commutative.

Proof. Given effects $e_1 = (A_1, P_1), e_2 = (A_2, P_2)$:

$$\begin{aligned}
e_1 \sqcap e_2 &= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) && \text{(def. of } \sqcap) \\
&= \mathcal{R}\big((A_2 \cap A_1, \mathcal{T}(e_2) \cap \mathcal{T}(e_1))\big) && \text{(commut. of } \cap) \\
&= e_2 \sqcap e_1 && \text{(def. of } \sqcap)
\end{aligned}$$

■

Claim. $(E, \sqcup, \sqcap)$ is a lattice.

Proof. The operations $\sqcup$ and $\sqcap$ are associative and commutative. For the operations to form a lattice, they must additionally satisfy the absorption properties for effects $e_1 = (A_1, P_1)$ and $e_2 = (A_2, P_2)$ in E:

1. $e_1 \sqcup (e_1 \sqcap e_2) = e_1$

$$\begin{aligned}
&e_1 \sqcup (e_1 \sqcap e_2) \\
&\quad = e_1 \sqcup \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) && \text{(def. of } \sqcap) \\
&\quad = e_1 \sqcup \big(A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2)\big) && \text{(lemma 2.3)} \\
&\quad = \mathcal{R}\big((A_1 \cup (A_1 \cap A_2), \mathcal{T}(e_1) \cup (\mathcal{T}(e_1) \cap \mathcal{T}(e_2)))\big) && \text{(def. of } \sqcup) \\
&\quad = \mathcal{R}\big((A_1, \mathcal{T}(e_1))\big) && \text{(appl. of } \cap, \cup) \\
&\quad = \mathcal{R}\big((A_1, A_1 \cup P_1)\big) && \text{(def. of } \mathcal{T}) \\
&\quad = (A_1, (A_1 \cup P_1) \setminus A_1) && \text{(def. of } \mathcal{R}) \\
&\quad = \big(A_1, (A_1 \setminus A_1) \cup (P_1 \setminus A_1)\big) && \text{(distrib. of } \setminus) \\
&\quad = (A_1, P_1 \setminus A_1) && \text{(appl. of } \setminus, \cup) \\
&\quad = (A_1, P_1) && \text{(def. of well-formedness, appl. of } \setminus) \\
&\quad = e_1 && \text{(def. of effect)}
\end{aligned}$$

2. $e_1 \sqcap (e_1 \sqcup e_2) = e_1$

$$
\begin{aligned}
& e_1 \sqcap (e_1 \sqcup e_2) \\
&\quad = e_1 \sqcap \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(def. of } \sqcup) \\
&\quad = e_1 \sqcap \big(A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2)\big) && \text{(lemma 2.3)} \\
&\quad = \mathcal{R}\big((A_1 \cap (A_1 \cup A_2), \mathcal{T}(e_1) \cap (\mathcal{T}(e_1) \cup \mathcal{T}(e_2)))\big) && \text{(def. of } \sqcap) \\
&\quad = \mathcal{R}\big((A_1, \mathcal{T}(e_1))\big) && \text{(appl. of } \cap, \cup) \\
&\quad = \mathcal{R}\big((A_1, A_1 \cup P_1)\big) && \text{(def. of } \mathcal{T}) \\
&\quad = (A_1, (A_1 \cup P_1) \setminus A_1) && \text{(def. of } \mathcal{R}) \\
&\quad = \big(A_1, (A_1 \setminus A_1) \cup (P_1 \setminus A_1)\big) && \text{(distrib. of } \setminus) \\
&\quad = (A_1, P_1 \setminus A_1) && \text{(appl. of } \setminus, \cup) \\
&\quad = (A_1, P_1) && \text{(def. of well-formedness, appl. of } \setminus) \\
&\quad = e_1 && \text{(def. of effect)}
\end{aligned}
$$

■

Claim. The effect $\bot = (\emptyset, \emptyset)$ is the identity of $\sqcup$. That is, $\forall e \in E : e \sqcup \bot = e$.

Proof. $\bot \in E$ because $\emptyset \cap \emptyset = \emptyset$ (so $\bot$ is well-formed) and $\emptyset \cup \emptyset \subseteq S$ (so $\mathcal{T}(\bot) \subseteq S$). Given effect $e = (A, P)$ s.t. $e \in E$,

$$
\begin{aligned}
e \sqcup \bot &= \mathcal{R}\big((A \cup \emptyset, \mathcal{T}(e) \cup \mathcal{T}(\bot))\big) && \text{(def. of } \sqcup) \\
&= \mathcal{R}\big((A \cup \emptyset, \mathcal{T}(e) \cup \emptyset)\big) && \text{(def. of } \mathcal{T}) \\
&= \mathcal{R}\big((A, \mathcal{T}(e))\big) && \text{(identity of } \cup) \\
&= (A, \mathcal{T}(e) \setminus A) && \text{(def. of } \mathcal{R}) \\
&= (A, (A \cup P) \setminus A) && \text{(def. of } \mathcal{T}) \\
&= \big(A, (A \setminus A) \cup (P \setminus A)\big) && \text{(distrib. of } \setminus) \\
&= \big(A, \emptyset \cup (P \setminus A)\big) && \text{(appl. of } \setminus) \\
&= (A, P \setminus A) && \text{(identity of } \cup) \\
&= (A, P) && \text{(def. of well-formedness, appl. of } \setminus) \\
&= e && \text{(def. of effect)}
\end{aligned}
$$

■

Claim. The effect $\top = (S, \emptyset)$ is the identity of $\sqcap$. That is, $\forall e \in E : e \sqcap \top = e$.

Proof. $\top \in E$ because $S \cap \emptyset = \emptyset$ (so $\top$ is well-formed) and $S \cup \emptyset \subseteq S$ (so $\mathcal{T}(\top) \subseteq S$). Given effect $e = (A, P)$ s.t. $e \in E$,

$$\begin{aligned}
e \sqcap \top &= \mathcal{R}\big((A \cap S, \mathcal{T}(e) \cap \mathcal{T}(\top))\big) && \text{(def. of } \sqcap) \\
&= \mathcal{R}\big((A \cap S, \mathcal{T}(e) \cap S)\big) && \text{(def. of } \top) \\
&= \mathcal{R}\big((A, \mathcal{T}(e))\big) && \text{(appl. of } \cap) \\
&= (A, \mathcal{T}(e) \setminus A) && \text{(def. of } \mathcal{R}) \\
&= (A, (A \cup P) \setminus A) && \text{(def. of } \mathcal{T}) \\
&= \big(A, (A \setminus A) \cup (P \setminus A)\big) && \text{(distrib. of } \setminus) \\
&= \big(A, \emptyset \cup (P \setminus A)\big) && \text{(appl. of } \setminus) \\
&= (A, P \setminus A) && \text{(identity of } \cup) \\
&= (A, P) && \text{(def. of well-formedness, appl. of } \setminus) \\
&= e && \text{(def. of effect)}
\end{aligned}$$

■

Therefore, $\mathfrak{L}_S = (E_S, \sqcup, \sqcap, \bot, \top)$ is a bounded lattice. □

Theorem 2.2 (Ordering on $\mathfrak{L}$). *For all S, $\sqsubseteq$ is a partial order induced by $\mathfrak{L}_S$.*

Proof. To show that $\sqsubseteq$ orders $\mathfrak{L}_S$ for any S, it suffices to show that

$$\forall e_1, e_2 : e_1 \in E_S \wedge e_2 \in E_S : e_1 \sqsubseteq e_2 \Leftrightarrow e_1 \sqcup e_2 = e_2.$$

$\Rightarrow$

$$\begin{aligned}
&e_1 \sqsubseteq e_2 \Rightarrow \\
&\qquad e_1 \sqcup e_2 = \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) && \text{(def. of } \sqcup) \\
&\qquad\qquad = \mathcal{R}\big((A_2, \mathcal{T}(e_2))\big) && \text{(def. of } \sqsubseteq\text{, appl. of } \cup) \\
&\qquad\qquad = \mathcal{R}\big((A_2, A_2 \cup P_2)\big) && \text{(def. of } \mathcal{T}) \\
&\qquad\qquad = (A_2, (A_2 \cup P_2) \setminus A_2) && \text{(def. of } \mathcal{R}) \\
&\qquad\qquad = \big(A_2, (A_2 \setminus A_2) \cup (P_2 \setminus A_2)\big) && \text{(distrib. of } \setminus) \\
&\qquad\qquad = \big(A_2, \emptyset \cup (P_2 \setminus A_2)\big) && \text{(appl. of } \setminus) \\
&\qquad\qquad = (A_2, P_2 \setminus A_2) && \text{(identity of } \cup) \\
&\qquad\qquad = (A_2, P_2) && \text{(def. of well-formedness, appl. of } \setminus) \\
&\qquad\qquad = e_2 && \text{(def. of effect)}
\end{aligned}$$

$\Leftarrow$

$$
\begin{aligned}
& e_1 \sqcup e_2 = e_2 \\
& \quad \Rightarrow \mathcal{R}\big((A_1 \cup A_2, \mathcal{T}(e_1) \cup \mathcal{T}(e_2))\big) = e_2 && \text{(def. of } \sqcup\text{, effect)} \\
& \quad \Rightarrow \big(A_1 \cup A_2, (\mathcal{T}(e_1) \cup \mathcal{T}(e_2)) \setminus (A_1 \cup A_2)\big) = e_2 && \text{(lemma 2.1)} \\
& \quad \Rightarrow A_1 \cup A_2 = A_2 \wedge \big(\mathcal{T}(e_1) \cup \mathcal{T}(e_2)\big) \setminus (A_1 \cup A_2) = P_2 && \text{(def. of effect)} \\
& \quad \Rightarrow A_1 \cup A_2 = A_2 \wedge \big((A_1 \cup P_1) \cup (A_2 \cup P_2)\big) \setminus (A_1 \cup A_2) = P_2 \\
& && \text{(def. of } \mathcal{T}\text{)} \\
& \quad \Rightarrow A_1 \cup A_2 = A_2 \wedge \big((A_1 \cup A_2) \cup (P_1 \cup P_2)\big) \setminus (A_1 \cup A_2) = P_2 \\
& && \text{(assoc., commut. of } \cup\text{)} \\
& \quad \Rightarrow A_1 \cup A_2 = A_2 \wedge A_2 \cup (P_1 \cup P_2) \setminus A_2 = P_2 && \text{(substitution)} \\
& \quad \Rightarrow A_1 \subseteq A_2 \wedge A_2 \cup (P_1 \cup P_2) \setminus A_2 = P_2 && \text{(appl. of } \cup\text{)} \\
& \quad \Rightarrow A_1 \subseteq A_2 \wedge P_1 \cup P_2 = P_2 && \text{(appl. of } \setminus\text{)} \\
& \quad \Rightarrow A_1 \subseteq A_2 \wedge P_1 \subseteq P_2 && \text{(appl. of } \cup\text{)} \\
& \quad \Rightarrow A_1 \subseteq A_2 \wedge (A_1 \cup P_1) \subseteq (A_2 \cup P_2) && \text{(appl. of } \cup\text{)} \\
& \quad \Rightarrow A_1 \subseteq A_2 \wedge \mathcal{T}(e_1) \subseteq \mathcal{T}(e_2) && \text{(def. of } \mathcal{T}\text{)} \\
& \quad \Rightarrow e_1 \sqsubseteq e_2 && \text{(def. of } \sqsubseteq\text{)}
\end{aligned}
$$

□

Theorem 2.3 ($\mathfrak{L}$ is distributive). *For all well-formed effects* $e_1 = (A_1, P_1), e_2 = (A_2, P_2), e_3 = (A_3, P_3)$,

$$e_1 \sqcap (e_2 \sqcup e_3) = (e_1 \sqcap e_2) \sqcup (e_1 \sqcap e_3)$$

Proof.

$$
\begin{aligned}
e_1 \sqcap (e_2 \sqcup e_3) &= e_1 \sqcap \mathcal{R}\big((A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3))\big) && \text{(def. of } \sqcup\text{)} \\
&= e_1 \sqcap \big(A_2 \cup A_3, \mathcal{T}(e_2) \cup \mathcal{T}(e_3)\big) && \text{(lemma 2.3)} \\
&= \mathcal{R}\big((A_1 \cap (A_2 \cup A_3), \mathcal{T}(e_1) \cap (\mathcal{T}(e_2) \cup \mathcal{T}(e_3)))\big) && \text{(def. of } \sqcap\text{)} \\
&= \mathcal{R}\big(((A_1 \cap A_2) \cup (A_1 \cap A_3), (\mathcal{T}(e_1) \cap \mathcal{T}(e_2)) \cup (\mathcal{T}(e_1) \cap \mathcal{T}(e_3)))\big) \\
& && \text{(dist. of } \cap\text{)} \\
&= \big(A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2)\big) \sqcup \big(A_1 \cap A_3, \mathcal{T}(e_1) \cap \mathcal{T}(e_3)\big) && \text{(def. of } \sqcup\text{)} \\
&= \mathcal{R}\big((A_1 \cap A_2, \mathcal{T}(e_1) \cap \mathcal{T}(e_2))\big) \sqcup \mathcal{R}\big((A_1 \cap A_3, \mathcal{T}(e_1) \cap \mathcal{T}(e_3))\big) && \text{(lemma 2.3)} \\
&= (e_1 \sqcap e_2) \sqcup (e_1 \sqcap e_3) && \text{(def. of } \sqcap\text{)}
\end{aligned}
$$

□

3 Understanding Effects in Context

On their own, effects are not particularly interesting. However, when combined with a formal description of what it means to be a program they enable us to prove several non-trivial properties of our programs, such as the fact that statements with non-interfering effects commute (section 5).

Notation While most of the notation used here is standard, there are a few notable exceptions:

- The $\in$ ("is element of") relation is overloaded to be meaningful for sequences as well as sets, in the obvious way.
- Expressions such as $x.T$ refer to the component of tuple x named T in the definition of x.
- A specific piece of a partition of an object is denoted with @, *e.g.*, $x@a$ refers to the element of $\mathcal{P}(x)$ named a.
- The notation $[x : T]$ is taken to mean "the type associated with object x is T," *i.e.*, $x.T = T$.
- A parenthesized zero-subscript such as in $\sigma_{(0)}$ is taken to mean "with every occurence of a variable x replaced with x_0".
- $[seq_1 \to seq_2]$ is taken to mean "with each occurence of a variable in seq_1 replaced with the corresponding variable in seq_2".
- $\langle a_1, a_2, \ldots \rangle$ denotes a sequence, and $\circ$ is sequence concatenation.

3.1 Definitions

Definition 3.1 (Object). *An* object $x = (T, R)$ *is a tuple consisting of a type T and a realization R.*

Definition 3.2 (Type). *A* type $T = (V, \texttt{Is\_Init}, Op)$ *is a tuple consisting of a (possibly infinite) set of values V, predicate* `Is_Init`*, and a sequence Op of operation contracts.*

Definition 3.3 (State). *A* state *for a universe of objects X is a function $\sigma : X \to \left(\bigcup T, R : (T, R) \in X : T.V\right)$ such that each object $x = (T, R)$ has a value $\sigma(x) \in T.V$.*

Definition 3.4 (State Space). *The* state space $\hat{\sigma}(X)$ *for a set or sequence of objects X is the set of all possible states on the objects in X.*

Definition 3.5 (Partition). *A* partition *over a set or sequence of objects X is a function $\mathcal{P}_X$ from X to nonempty finite sets. (The universe of objects X is usually left implicitly-defined and we write $\mathcal{P}$.)*

Definition 3.6 (Collective partition). *The* collective partition $\hat{\mathcal{P}}(X)$ *of a set or sequence of objects X is*

$$\hat{\mathcal{P}}(X) = \bigcup x : x \in X : \mathcal{P}(x).$$

Definition 3.7 (Operation contract). *An* operation contract $o = (i, \pi, pre, post, \mathcal{S})$ *is a tuple consisting of an identifier i, a sequence of parameters (*i.e.*, objects) π, a precondition predicate pre, a postcondition predicate $post$, and a specified effect $\mathcal{S}$.*

Definition 3.8 (Specified effect). *For an operation contract $o = (i, \pi, pre, post, \mathcal{S})$, the* specified effect *is a function $\mathcal{S} : \hat{\sigma}(\pi) \to \left\{(A, P) : \mathcal{T}\big((A, P)\big) \subseteq \hat{\mathcal{P}}(\pi)\right\}$.*

Definition 3.9 (Realization). *The* realization *of object $x = (T, R)$ is a tuple $R = (B, F, I, C, \mathfrak{I})$ of a sequence B of operation bodies (*i.e.*, sequences of program statements), a set F of fields (*i.e.*, objects), a representation invariant $I : \hat{\sigma}(F) \to \{\mathsf{true}, \mathsf{false}\}$, an abstraction relation $C \subseteq \hat{\sigma}(F) \times T.V$, and a function $\mathfrak{I} : \hat{\sigma}(F) \times \hat{\mathcal{P}}(F) \to \mathcal{P}(x)$.*

Definition 3.10 (Actual effect). *The* actual effect *of an operation body b with parameters π in realization $R = (B, F, I, C, \mathfrak{I})$ for some $\sigma \in \hat{\sigma}(\pi)$, denoted $\mathcal{A}_\sigma(b)$, is the combined effect of each statement in b that is executed when b starts with its parameters having the values in σ. Details of its structure are below, in section 4.*

Definition 3.11 (The Implements relation). *An operation body b* implements *an operation contract $o = (i, \pi, pre, post, \mathcal{S})$, denoted $b \mapsto o$, if*

$$\forall \sigma, \sigma' : \sigma \xrightarrow{b} \sigma' \Rightarrow \sigma \xrightarrow{o(\pi)} \sigma'$$

A realization $R = (B, F, I, C, \mathfrak{I})$ implements *a type $T = (V, v_0, O)$, denoted $R \mapsto T$, if*

$$\forall i : 0 \leq i < |O| : B_i \mapsto O_i.$$

Definition 3.12 (The Respects relation). *An operation body b* respects *an operation contract $o = (i, \pi, pre, post, \mathcal{S})$, denoted $b \rightarrowtail o$, if*

$$\forall \sigma : \sigma \vdash pre : \mathcal{A}_\sigma(b) \sqsubseteq \mathcal{S}(\sigma).$$

A realization $R = (B, F, I, C, \mathfrak{I})$ respects *a type $T = (V, v_0, O)$, denoted $R \rightarrowtail T$, if*

$$\forall i : 0 \leq i < |O| : B_i \rightarrowtail O_i.$$

Definition 3.13 (Validity). *An operation body b is* valid *for an operation contract o if $b \mapsto o \wedge b \rightarrowtail o$.*

A realization R is valid *for a type T if every operation body is valid for the corresponding contract in T.*

3.2 Well-Formedness Conditions

Definition 3.14 (Well-formedness of specified effects). *A specified effect $\mathcal{S}$ is* well-formed *if $\big(\forall \sigma : \mathcal{S}(\sigma)$ is well-formed$\big)$. $\mathcal{S}$ is* well-formed with respect to *a set T if $\mathcal{S}$ is well-formed and $\big(\forall \sigma : \mathfrak{T}(\mathcal{S}(\sigma)) \subseteq T\big)$.*

Definition 3.15 (Well-formedness of operation contracts). *An operation contract $o = (i, \pi, pre, post, \mathcal{S})$ is* well-formed *if:*

1. *Every identifier in π is unique*
2. *Every free variable in pre is in π*
3. *Every free variable in post is of the form x or x_0 where x is in π*
4. *$\mathcal{S}$ is well-formed with respect to $\hat{\mathcal{P}}(\pi)$*

Definition 3.16 (Well-formedness of types). *A type $T = (V, O)$ is* well-formed *if each contract in O is well-formed.*

Definition 3.17 (Well-formedness of realizations). *A realization $R = (B, F, I, C, \mathfrak{I})$ is* well-formed with respect to *a type $T = (V, O)$ if $R \mapsto T$ and $R \rightarrowtail T$.*

Definition 3.18 (Well-formedness of objects). *An object $x = (T, R)$ is* well-formed *if T is well-formed and R is well-formed with respect to T.*

Hereafter, we are concerned only with well-formed objects and realizations.

4 Programs

A program consists of a sequence of *statements*. Each statement is either a variable declaration, a control structure (*e.g.*, a conditional statement or a loop), or an *operation call* that has some number of *arguments* that correspond to that operation's *parameters*. Each statement has a *behavior* and an *effect*. For an operation call, the behavior of the statement is some relation on the values of the arguments, derived from the behavioral specification (*e.g.*, pre- and post-conditions). Behavioral specifications and verification are discussed at length in the literature [**?**]; we assume that there a mechanism through which we can reason about a behavioral specification, that is, a definition of correctness and a formal semantics. Effects are derived from a non-interference specification (part of an operation contract) and are manipulated via the A/P Calculus introduced in section 2. An operation is implemented by an *operation body*, a sequence of statements involving the parameters of the operation.

4.1 The Language

We define a simple programming language (and its semantics with respect to effects) that will enable us to compute the actual effect of an operation body under an assignment of the operation's parameters, the grammar for which is in fig. 4.1.

It is important to note that this language is one possible programming language the implements the programming model from section 4. There is nothing in this work that fundamentally requires this particular language, although semantics are—in this paper—defined concretely in its terms.

One key restriction placed on the structure of programs in a language that implements the programming model from section 4 (beyond the usual ones about scoping, typing, *etc.*) is that an ⟨*id-list*⟩ must consist of *unique* ⟨*id*⟩s. This restriction is left implicit in the inference rules to improve readability.

Fig. 4.1. Context-free grammar for our programming language.

⟨*body*⟩	::=	`operation` ⟨*op-name*⟩ `(`⟨*id-list*⟩`) :` ⟨*stmt*⟩ `end`	Operation Body
⟨*id-list*⟩	::=	⟨*id*⟩`,` ⟨*id-list*⟩	Identifier List
	|	⟨*id*⟩	
⟨*stmt*⟩	::=	ε	Empty Statement
	|	⟨*simp-stmt*⟩`;`	Simple Statement
	|	⟨*stmt*⟩$_1$ ⟨*stmt*⟩$_2$	Sequential Composition
	|	`if Read(`⟨*id*⟩`) then` ⟨*stmt*⟩$_1$ `else` ⟨*stmt*⟩$_2$ `end`	If Statement
	|	`while` ⟨*id*⟩ `do` ⟨*stmt*⟩ `end`	While Statement
	|	`cobegin` ⟨*par-block*⟩ `end`	Cobegin Statement
⟨*simp-stmt*⟩	::=	⟨*type-id*⟩ ⟨*id*⟩	Variable Declaration
	|	⟨*id*⟩$_1$:=: ⟨*id*⟩$_2$	Swap
	|	⟨*op-call*⟩	
⟨*op-call*⟩	::=	⟨*op-name*⟩ `(`⟨*id-list*⟩`)`	Operation Call
⟨*par-block*⟩	::=	⟨*op-call*⟩ ‖ ⟨*par-block*⟩	Parallel Block
	|	⟨*op-call*⟩	

4.2 Parameter Passing and Aliases

Our language, by design, does not premit aliases in the usual sense and parameter passing in the language is, strictly speaking, by reference. However, parameter passing by reference might be said to introduce an alias (*i.e.*, between the argument and the formal parameter) even though the two names are not in scope at the same time. In sequential programs, this cannot introduce unsoundness to reasoning with value semantics. Unsoundness can arise, however, when concurrency is introduced via a **`cobegin`** statement in which several parallel operation calls share an argument. If several parallel operation calls attempt to mutate the same object, it could produce an inconsistent state (*i.e.*, one that does not satisfy the representation invariant of that object). A consequence of the facts proven below in section 5 is that given pairwise non-interference between the ⟨*op-call*⟩s in a ⟨*par-block*⟩, this sharing does not introduce any nondeterminism so the value of an object is always well-defined even in the presence of such data sharing among threads.

It is because of this tension between reasoning about sequential and concurrent programs that the language has two *conceptual* parameter passing mechanisms. Shared arguments to multiple parallel operation calls are reasoned about using pass-by-reference semantics, while repeated arguments within a single operation call are reasoned about using pass-by-swapping semantics (although such repeated arguments should be avoided). Under pass-by-swapping, each argument is swapped into the corresponding formal parameter at the point of the call—leaving the argument with an initial value for its type—and that formal

parameter is swapped back to the argument at the end of the operation. Such a reasoning "shortcut" guarantees that in a sequential program there is *never* a time at which there are two names for the same object—even names that are not simultaneously in scope. It also sidesteps the well-known repeated arguments problem [**?**] because it provides for well-defined semantics when the same argument is provided several times to the same operation call.[4]

While alias freedom may seem like a too-restrictive condition to place on programs, it has been shown that real-world software can be built entirely without aliases [**?**], and when they are absolutely necessary aliases may be dealt with as a special case [**?**]. Several popular modern programming languages, in fact, have mechanisms to be alias-free by default [**?**,**?**], and require the program to put in extra work to introduce aliases. In our programming model, the alias-free restriction manifests in the following axiom.

Axiom 1 (Alias Freedom)**.** *For any two distinct objects x, y in scope,* $\mathcal{P}(x) \cap \mathcal{P}(y) = \emptyset$.

4.3 Primitive Operations

Without additional machinery, programs in the model of section 4 have no lowest-level implementation because everything is implemented in terms of something else. Therefore, it is necessary to ground these programs with *primitive operations* that do not themselves have operation bodies. Each primitive operation is *atomic*: it occurs in one "step" of execution and it operates on entire objects rather than pieces of objects. A primitive operation in any language that implements our programming model is a refinement of the contract below for `Prim`.

$$\left(\begin{array}{l} i : \texttt{Prim}, \\ \pi : \pi_A \circ \pi_P, \\ \mathit{pre} : \mathsf{true}, \\ \mathit{post} : \left(\forall x : \begin{cases} \exists f : x = f(x_0) & x \in \pi_A \\ x = x_0 & x \in \pi_P \end{cases}\right), \\ \mathcal{S}(\sigma) : \left(\hat{\mathcal{P}}(\pi_A), \hat{\mathcal{P}}(\pi_P)\right) \end{array}\right) \tag{Prim}$$

4.4 Possible Primitive Operations

As we use a specific language for this paper, we additionally propose several possible primitive operations that we use in the language of fig. 4.1. These operations are `Init` and `Swap` (both refinements of `Prim`).

[4] In particular, the *first* time an argument appears, its value is swapped into the corresponding formal parameter. Remaining occurences are left with an initial value.

The `Init` Operation The first primitive operation in our language is the `Init` operation. Its contract is below.

$$\begin{pmatrix} i : \texttt{Init}, \\ \pi : \langle T, x \rangle, \\ \mathit{pre} : \mathsf{true}, \\ \mathit{post} : T.\texttt{Is\_Init}(x), \\ \mathcal{S}(\sigma) : \big(\mathcal{P}(x), \emptyset\big) \end{pmatrix} \tag{Init}$$

The `Init` operation sets the value of its argument to the initial value for its type. The `Init` operation is used implicitly in variable declaration statements in our language; the argument v is equal to v_0 in the type T provided as part of that statement.

The `Swap` Operation The other primitive operation in the language is the `Swap` operation, with the following contract.

$$\begin{pmatrix} i : \texttt{Swap}, \\ \pi : \langle x, y \rangle, \\ \mathit{pre} : \mathsf{true}, \\ \mathit{post} : (x = y_0 \wedge y = x_0), \\ \mathcal{S}(\sigma) : \big(\mathcal{P}(x) \cup \mathcal{P}(y), \emptyset\big) \end{pmatrix} \tag{Swap}$$

The `Swap` operation is used implicitly by the :=: operator in the language.

Example: Implementing a Constant A constant can be implemented within the language by defining a new type for each value one wishes to use. For example, the integral constant 42 is implemented by the type `C42` (4.4.1); it is instantiated by a statement such as "`C42 FORTY_TWO;`".

$$\texttt{C42} = \big(\{42\}, 42, \langle\rangle\big) \tag{4.4.1}$$

Example: Implementing `Bit` Using the primitive operations (`Swap`) and (`Init`), one could implement any other component. For example, it is possible to implement a single bit that provides two operations (`Set` and `Unset`, defined below) using the primitive operations in the language.[5] Objects of type `Bit` have a singleton partition, that is, $\forall b : b.T = \texttt{Bit} : \mathcal{P}(b) = \{b\}$.

$$\texttt{Bit} = \big(\{\mathsf{true}, \mathsf{false}\}, \mathsf{false}, \langle \texttt{Set}, \texttt{Unset} \rangle\big) \tag{4.4.2}$$

[5] The specifics of the implementation of `Bit` are left as an exercise to the reader.

$$\begin{pmatrix} i : \texttt{Set}, \\ \pi : \langle b \rangle, \\ pre : [b : \texttt{Bit}], \\ post : b, \\ \mathcal{S}(\sigma) : \big(\{b\}, \emptyset\big) \end{pmatrix} \tag{Set}$$

$$\begin{pmatrix} i : \texttt{Unset}, \\ \pi : \langle b \rangle, \\ pre : [b : \texttt{Bit}], \\ post : \neg b, \\ \mathcal{S}(\sigma) : \big(\{b\}, \emptyset\big) \end{pmatrix} \tag{Unset}$$

Observe that by using objects of type `Bit`, it is possible to implement a functionally complete set of logical operators (*e.g.*, by implementing NOR, denoted by $\downarrow$), and therefore to implement a computer. For example, the operation body in listing 4.1 is valid for the operation contract (`Nor`).

$$\begin{pmatrix} i : \texttt{Nor}, \\ \pi : \langle a, b \rangle, \\ pre : \big([a : \texttt{Bit}] \wedge [b : \texttt{Bit}]\big), \\ post : \big(a = a_0 \downarrow b_0 \wedge b = b_0\big), \\ \mathcal{S}(\sigma) : \big(\{a\}, \{b\}\big) \end{pmatrix} \tag{Nor}$$

Listing 4.1. Valid operation body for (`Nor`).

```
operation Nor(a, b) :
  if a then
    Unset(a);
  else
    if b then
      Unset(a);
    else
      Set(a);
    end
  end
end
```

4.5 Bheavioral Semantics

In this section, we formalize the semantics of the language presented in fig. 4.1. To reiterate, the language is just *one possible implementation* of the programming

model presented in section 4. The behavior of a statement s is notated with $\xrightarrow{s}$, a (non-total) relation between states defined by $\mathsf{post}(s)$. The domain of $\xrightarrow{s}$ is restricted by $\mathsf{pre}(s)$—that is, $\sigma \xrightarrow{s} \sigma'$ is only defined when $\sigma \vdash \mathsf{pre}(s)$. For an operation call, $\mathsf{pre}(s)$ and $\mathsf{post}(s)$ are equal to the *pre* and *post* predicates in the contract for the operation.

$$\sigma \xrightarrow{s} \sigma' \Leftrightarrow \big(\sigma \vdash \mathsf{pre}(s) \wedge \sigma_{(0)}, \sigma' \vdash \mathsf{post}(s)\big) \tag{4.5.1}$$

Frame Rule There is an enormous body of work on formalizing the semantics of programming languages, so most of the rules below (4.5.3–12) are not too interesting. One deviation from normal semantics we make for statements in our language is the formulation of the frame rule. Typically, a frame rule is formulated in terms of variables or objects; here it is defined in terms of partitions and effects.

$$\textsc{Frame Rule}\quad \frac{\sigma \xrightarrow{s} \sigma' \qquad \mathcal{A}_\sigma(s).A \cap \mathcal{P}(x) = \emptyset}{\sigma(x) = \sigma'(x)} \tag{4.5.2}$$

The rule for sequential composition (4.5.8) is slightly awkward because it must account for the possiblilty of relational behavior. The awkwardness arises from the decision to formulate the semantics in terms of state transitions rather than predicates. Moreover, the semantics of ⟨*par-block*⟩ in (4.5.12) is perhaps surprising. The parallel execution of two statements is modeled as an arbitrary interleaving of the constituent instructions in the implementations of those statements (*e.g.*, operation bodies). Informally, the semantics rule for ⟨*par-block*⟩ states that whenever the operation calls in ⟨*par-block*⟩ are non-interfering, if the precondition of *any* permutation of the calls in ⟨*par-block*⟩ is satisfied by state σ then the resultant state σ' satisfies the postcondition of *every* permutation of the calls in ⟨*par-block*⟩.[6] The soundness of this rule is a consequence of the theorems proved in section 5.

The "context" or "environment" for the rules below is provided as (X, M), a symbol table consisting of X, the objects in scope, and M, the operations in scope. The context is changed by (4.5.5), in which X grows to X' (in the other rules $X = X'$). Finally, $\sigma \in \hat{\sigma}(X)$ and $\sigma' \in \hat{\sigma}(X')$.

$$\textsc{Operation Body}\quad \frac{b = \text{operation } op(ids)\text{: } s \text{ end} \qquad \sigma \vdash pre \qquad \sigma \xrightarrow{s} \sigma'}{\sigma \xrightarrow{b} \sigma'} \tag{4.5.3}$$

$$\textsc{Semantics of } \varepsilon\quad \frac{}{\sigma \xrightarrow{\varepsilon} \sigma} \qquad \textsc{Semantics of Variable Declaration}\quad \frac{\sigma = \sigma' \restriction_X \qquad X \cup \{x\} = X' \qquad [x : T] \qquad \sigma'(x) = T.v_0}{\sigma \xrightarrow{T\ x;} \sigma'}$$

[6] Strictly speaking, the rule as written only considers permutations where ⟨*op-call*⟩ is the first or last call in the sequence.

(4.5.4)

(4.5.5)

Semantics of Swap

$$\frac{\sigma = \sigma'[\langle x, y\rangle \to \langle y, x\rangle]}{\sigma \xrightarrow{x:=:y} \sigma'} \tag{4.5.6}$$

Semantics of Operation Call

$$\frac{\big(op, \pi, pre, post, \mathcal{S}\big) \in M \qquad \sigma \vdash pre[\pi \to ids] \qquad \sigma_{(0)}, \sigma' \vdash post[\pi \to ids]}{\sigma \xrightarrow{op(ids);} \sigma'} \tag{4.5.7}$$

Sequential Composition

$$\frac{\forall \sigma' : \sigma \xrightarrow{s_1;} \sigma' : \sigma' \vdash \mathsf{pre}(s_2) \qquad \forall \sigma' : \sigma' \xrightarrow{s_2;} \sigma'' : \sigma_{(0)}, \sigma' \vdash \mathsf{post}(s_1)}{\sigma \xrightarrow{s_1;s_2;} \sigma''} \tag{4.5.8}$$

If Statement

$$\frac{\sigma(x) \Rightarrow \sigma \xrightarrow{s_1;} \sigma' \qquad \neg\sigma(x) \Rightarrow \sigma \xrightarrow{s_2;} \sigma'}{\sigma \xrightarrow{\text{if } x \text{ then } s_1 \text{ else } s_2 \text{ end};} \sigma'} \tag{4.5.9}$$

While Statement

$$\frac{\sigma(x) \Rightarrow \sigma \xrightarrow{s_1;\ \text{while } x \text{ do } s_1 \text{ end};} \sigma' \qquad \neg\sigma(x) \Rightarrow \sigma \xrightarrow{\varepsilon} \sigma'}{\sigma \xrightarrow{\text{while } x \text{ do } s_1 \text{ end};} \sigma'} \tag{4.5.10}$$

Semantics of Cobegin

$$\frac{\sigma \xrightarrow{par} \sigma'}{\sigma \xrightarrow{\text{cobegin } par \text{ end};} \sigma'}$$

Semantics of Parallel Block

$$\frac{\begin{array}{c}\mathcal{A}_\sigma(op) \ddagger \mathcal{A}_\sigma(par) \\ \sigma \vdash \big(\mathsf{pre}(op; par;) \vee \mathsf{pre}(par; op;)\big) \\ \sigma_{(0)}, \sigma' \vdash \big(\mathsf{post}(op; par;) \wedge \mathsf{post}(par; op;)\big)\end{array}}{\sigma \xrightarrow{op \| par} \sigma'}$$

(4.5.11)

(4.5.12)

4.6 Effect Semantics

We define the actual effect of a statement ($\mathcal{A}_\sigma(s)$) separately for each of the various kinds of statements in the language. The determination of the actual effect of a statement must be done in tandem with the determination of the behavior of the that statement: the actual effect depends on the values of the objects, as derived from the behavioral semantics as in section 4.5.

It is frequently useful to apply the non-interference correspondence, $\mathcal{I}$, of a realization $R = (B, F, I, C, \mathcal{I})$ to an effect e (rather than to a single piece of a partition). For this purpose we define the related function $\hat{\mathcal{I}} : \hat{\sigma}(F) \times E \to E$ as follows, where $E = \{e : e \text{ is a well-formed effect}\}$ (as defined in section 2.3).

$$\hat{\mathcal{I}}(\sigma, e) = \begin{array}{c}\big(\{\mathcal{I}(\sigma, p) : p \in e|_{\hat{\mathcal{P}}(F)}.A\}, \{\mathcal{I}(\sigma, p) : p \in e|_{\hat{\mathcal{P}}(F)}.P\}\big) \sqcup \\ \big(\{p : p \in (e.A \setminus \hat{\mathcal{P}}(F))\}, \{p : p \in (e.P \setminus \hat{\mathcal{P}}(F))\}\big)\end{array}$$

The actual effect of an operation body is defined in rule (4.6.1) below. For that rule, realization $R = (B, F, I, C, \mathfrak{I})$ is well-formed with respect to a specification $T = (V, O)$, operation contract $o = (i, \pi, \mathit{pre}, \mathit{post}, \mathfrak{S})$ is such that $o \in O$, and $\sigma \in \hat{\sigma}(\pi)$.

The defintions of the actual effect for the other kinds of statements and expressions are in rules (4.6.2–10) below. For each rule below, we are given $S = (X, M)$ and $\sigma \in \hat{\sigma}(X)$ (as in section 4.5).

– $b =$ operation $\mathit{op}(\mathit{ids})$: s end

$$\mathcal{A}_\sigma(b) = \hat{\mathfrak{I}}\big(\sigma, \mathcal{A}_\sigma(s)\big)|_{\hat{\mathcal{P}}(\mathit{ids})} \tag{4.6.1}$$

– $s = \ \varepsilon$

$$\mathcal{A}_\sigma(s) = \bot \tag{4.6.2}$$

– $s = \ T\ x;$

$$\mathcal{A}_\sigma(s) = \big(\mathcal{P}(x), \emptyset\big) \tag{4.6.3}$$

– $s = \ x :=: y;$

$$\mathcal{A}_\sigma(s) = \big(\mathcal{P}(x) \cup \mathcal{P}(y), \emptyset\big) \tag{4.6.4}$$

– $s = \ \mathit{op}(\mathit{ids})$ where $(\mathit{op}, \pi, \mathit{pre}, \mathit{post}, \mathfrak{S})$ is the well-formed contract of some operation.

$$\mathcal{A}_\sigma\big(s\big) = \mathfrak{S}(\sigma[\mathit{ids} \to \pi])[\pi \to \mathit{ids}] \tag{4.6.5}$$

– $s = \ s_1; s_2$

$$\mathcal{A}_\sigma(s) = \mathcal{A}_\sigma(s_1) \sqcup \Big(\bigsqcup \sigma' : \sigma \xrightarrow{s_1} \sigma' : \mathcal{A}_{\sigma'}(s_2)\Big) \tag{4.6.6}$$

– $s = \$ if x then s_1 else s_2 end

$$\mathcal{A}_\sigma(s) = \big(\emptyset, \mathcal{P}(x)\big) \sqcup \begin{cases} \mathcal{A}_\sigma(s_1)|_{\hat{\mathcal{P}}(X)} & \sigma(x) \\ \mathcal{A}_\sigma(s_2)|_{\hat{\mathcal{P}}(X)} & \neg\sigma(x) \end{cases} \tag{4.6.7}$$

– $s = \$ while x do s_1 end

$$\mathcal{A}_\sigma(s) = \big(\emptyset, \mathcal{P}(x)\big) \sqcup \begin{cases} \mathcal{A}_\sigma(s_1; s)|_{\hat{\mathcal{P}}(X)} & \sigma(x) \\ \bot & \neg\sigma(x) \end{cases} \tag{4.6.8}$$

– $s = \$ cobegin par end

$$\mathcal{A}_\sigma(s) = \mathcal{A}_\sigma(\mathit{par}) \tag{4.6.9}$$

– $s = \ \mathit{op} \| \mathit{par}$

$$\mathcal{A}_\sigma(s) = \mathcal{A}_\sigma(\mathit{op}) \sqcup \mathcal{A}_\sigma(\mathit{par}) \tag{4.6.10}$$

Syntactic Effect It is occasionally useful to determine the effect of a statement on a conservative basis, without regard for the initial state. For these cases we define the *syntactic effect*, denoted $\mathcal{A}(s)$, so named because it is syntactically derivable from the program text. Formally,

$$\mathcal{A}(s) = \bigsqcup \sigma : \mathcal{A}_\sigma(s).$$

The syntactic effect is always more conservative than the actual effect, and so can be used in place of the actual effect without compromising soundness (although there would be a penalty to completeness). That is,

$$\forall s, \sigma : \mathcal{A}_\sigma(s) \sqsubseteq \mathcal{A}(s).$$

The syntactic effect is closely related to the theoretical underpinnings of other concurrency-focused programming languages such as DPJ [**?**] and ParaSail [**?**]; the effects calculus in section 2 and reasoning-focused language in section 4 helps to generalize those other results.

5 Results

Through the combintation of the effects calculus of section 2 and the programming model and language of section 4 and section 4.1, a variety of interesting results can be achieved. These results have implications for reasoning about and verifiying the correctness of parallel programs in languages that implement the model described above (see section 5.1). They are enumerated and proved here as a sequence of lemmas a theorems.

The first lemma, lemma 5.1, states that any object not mentioned in a statement is not in the target of the actual effect of that statement; by the frame rule, then, any object not mentioned in a statement does not have its value changed by that statement. It helps us apply the frame rule even to objects that are not mentioned in a statement.

Lemma 5.1. *For any statement s, state σ, and object x not mentioned in s, $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.*

Proof. We proceed by induction on s.

Base Cases

- $s = \varepsilon$.

  By (4.6.2), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \emptyset$ and the lemma is trivially true.

- $s = T\ y$.

  By (4.6.3), $\mathcal{T}(\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y)$. If x is not mentioned in s, then $x \neq y$. Therefore, $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

– $s =\ op(ids)$.

  By well-formedness definitions 3.14 and 3.15 and (4.6.5), if x is not mentioned in s then $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$, so the lemma holds for s.

Inductive Step

For induction, we assume that s_1 and s_2 are such that x does not appear in either statement (and thus have the property that $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big) \cap \mathcal{P}(x) = \emptyset$).

– $s =\ s_1; s_2$.

  By (4.6.6), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big)$. By our inductive hypothesis, $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big) \cap \mathcal{P}(x) = \emptyset$. Thus, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

– $s =$ if y then s_1 else s_2 end.

  By (4.6.7), either $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big)$ or $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big)$. If x is not mentioned in x, then $y \neq x$; by our inductive hypothesis $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(s_2)\big) \cap \mathcal{P}(x) = \emptyset$. Thus, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

– $s =$ while y do s_1 end.

  By (4.6.8), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{P}(y) \cup \mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big)$. If x is not mentioned in s, then $x \neq y$; by our inductive hypothesis $\mathcal{T}\big(\mathcal{A}_\sigma(s_1)\big) \cap \mathcal{P}(x) = \emptyset$. hus, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

– $s =$ cobegin $op \| par$ end.

  By (4.6.10), $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) = \mathcal{T}\big(\mathcal{A}_\sigma(op)\big) \cup \mathcal{T}\big(\mathcal{A}_\sigma(par)\big)$. Sincy by assumption $\mathcal{T}\big(\mathcal{A}_\sigma(op)\big) \cap \mathcal{P}(x) = \emptyset$ and $\mathcal{T}\big(\mathcal{A}_\sigma(par)\big) \cap \mathcal{P}(x) = \emptyset$, it follows that $\mathcal{T}\big(\mathcal{A}_\sigma(s)\big) \cap \mathcal{P}(x) = \emptyset$.

□

The second lemma, 5.2, implies that statements that share no objects commute. This result is helpful in reasoning about simple parallel programs (sometimes called "embarassingly parallel") in which there is no shared data among the parallel threads. It serves as a proof of concept that the reasoning framework introduced here is not too far-fetched.

Let $\hat{X}_\sigma(s) = \big\{x : \mathcal{P}(x) \cap \mathcal{T}(\mathcal{A}_\sigma(s)) \neq \emptyset\big\}$ (*i.e.*, the objects mentioned in s in a statement that is executed when s begins in state σ).[7] $\bar{X}_\sigma(s)$ is its complement.

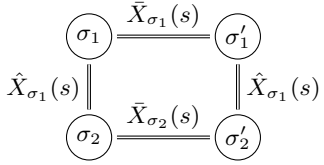
[7] $\hat{X}(s)$ (without an identifying state) is defined as $\big\{x : \mathcal{P}(x) \cap \mathcal{T}(\mathcal{A}(s)) \neq \emptyset\big\}$, the set of objects mentioned anywhere in s.

Lemma 5.2. *For all statements s and states σ_1, σ_1', σ_2, and σ_2',*

$$\left(\begin{array}{l}\Big(\sigma_1 \restriction_{\hat{X}_{\sigma_1}(s)} = \sigma_2 \restriction_{\hat{X}_{\sigma_2}(s)}\Big) \wedge \Big(\sigma_1' \restriction_{\hat{X}_{\sigma_1}(s)} = \sigma_2' \restriction_{\hat{X}_{\sigma_2}(s)}\Big) \wedge \\ \Big(\sigma_1 \restriction_{\bar{X}_{\sigma_1}(s)} = \sigma_1' \restriction_{\bar{X}_{\sigma_1}(s)}\Big) \wedge \Big(\sigma_2 \restriction_{\bar{X}_{\sigma_2}(s)} = \sigma_2' \restriction_{\bar{X}_{\sigma_2}(s)}\Big)\end{array}\right) \Rightarrow \Big((\sigma_1 \xrightarrow{s} \sigma_1') \Leftrightarrow (\sigma_2 \xrightarrow{s} \sigma_2')\Big).$$

That is, whenever σ_1, σ_1', σ_2, and σ_2' are related as in fig. 5.1, it follows that $\sigma_1 \xrightarrow{s} \sigma_1' \Leftrightarrow \sigma_2 \xrightarrow{s} \sigma_2'$.

Fig. 5.1. Relationships between σ_1, σ_1', σ_2, and σ_2' described in lemma 5.2. An edge $\overset{D}{=\!=}$ between two states indicates they are equal when restricted to the domain D.



Proof. First, observe that for the antecedent to hold, it must be the case that $\hat{X}_{\sigma_1}(s) = \hat{X}_{\sigma_2}(s)$ (and, therefore that $\bar{X}_{\sigma_1}(s) = \bar{X}_{\sigma_2}(s)$). We can therefore reason (without loss of generality) only with $\hat{X}_{\sigma_1}(s)$ and $\bar{X}_{\sigma_1}(s)$.

We proceed by induction on s.

Base Cases

- $s = \varepsilon$.

  By (4.6.2), $\mathcal{T}(\mathcal{A}_{\sigma_1}(s)) = \emptyset$; therefore $\hat{X}_{\sigma_1}(s) = \emptyset$, so $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$. By (4.5.4), for any σ, $\sigma \xrightarrow{\varepsilon} \sigma$, so the lemma holds.

- $s = T\ x;$.

  By (4.6.3), $\mathcal{T}(\mathcal{A}_{\sigma_1}(s)) = \mathcal{P}(x)$; therefore $\hat{X}_{\sigma_1}(s) = \{x\}$. By (4.5.5), $\sigma_1 \xrightarrow{s} \sigma_1'$ whenever $\sigma_1 = \sigma_1' \restriction_X$ (where X is the set of in-scope objects before s) and σ_1' is additionally defined on x. Moreover, whenever $\sigma_1 \xrightarrow{s} \sigma_1'$, and the antecedent of the lemma holds, it is also true that $\sigma_2 \xrightarrow{s} \sigma_2'$. Therefore the lemma holds for $s = T\ x;$.

- $s = op(ids);$.

  Let there be some $(op, \pi, pre, post, \mathcal{S}) \in M$. It follows from (4.5.7) and the definition of $\xrightarrow{s}$ (4.5.1) that $\sigma_1 \vdash pre \Leftrightarrow \sigma_1 \vdash \mathsf{pre}(s)$. By assumption, for each object x mentioned in s, $\sigma_1(x) = \sigma_2(x)$. Because *pre* mentions only objects

in π (by well-formedness definition 3.15), it follows that $\sigma_1 \vdash \mathit{pre} \Leftrightarrow \sigma_2 \vdash \mathit{pre}$ (and, therefore, that $\sigma_2 \vdash \mathit{pre} \Leftrightarrow \sigma_2 \vdash \mathsf{pre}(s)$). Therefore $\sigma_1 \vdash \mathsf{pre}(s) \Leftrightarrow \sigma_2 \vdash \mathsf{pre}(s)$. By analogous reasoning, we see that $\sigma_{1(0)}, \sigma_1' \vdash \mathsf{post}(s) \Leftrightarrow \sigma_{2(0)}, \sigma_2' \vdash \mathsf{post}(s)$. Therefore, $\sigma_1 \xrightarrow{s} \sigma_1' \Leftrightarrow \sigma_2 \xrightarrow{s} \sigma_2'$.

Inductive Step

Assume the lemma holds for s_1 and s_2.

- $s =$ $s_1; s_2;$.
  Assume we have some states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ related as in fig. 5.1. Without loss of generality we proceed from $\sigma_1 \xrightarrow{s} \sigma_1'$. By (4.5.8), $\sigma_1 \xrightarrow{s} \sigma_1'$ whenever

  $$\left(\forall \sigma'' : \sigma_1 \xrightarrow{s_1} \sigma'' : \sigma'' \vdash \mathsf{pre}(s_2)\right) \wedge \left(\forall \sigma'' : \sigma'' \xrightarrow{s_2} \sigma_1' : \sigma_{1(0)}, \sigma'' \vdash \mathsf{post}(s_1)\right).$$

  By our inductive hypothesis, then, it is also true that

  $$\left(\forall \sigma'' : \sigma_2 \xrightarrow{s_1} \sigma'' : \sigma'' \vdash \mathsf{pre}(s_2)\right) \wedge \left(\forall \sigma'' : \sigma'' \xrightarrow{s_2} \sigma_2' : \sigma_{2(0)}, \sigma'' \vdash \mathsf{post}(s_1)\right).$$

  Therefore, by (4.5.8), $\sigma_2 \xrightarrow{s} \sigma_2'$. Therefore, given $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ related as in fig. 5.1, it follows that $\sigma_1 \xrightarrow{s} \sigma_1' \Leftrightarrow \sigma_1 \xrightarrow{s} \sigma_2'$.
- $s =$ if x then s_1 else s_2 end;.
  By (4.5.9), $\sigma_1 \xrightarrow{s} \sigma_1'$ if and only if either $\sigma_1 \xrightarrow{s_1} \sigma_1'$ or $\sigma_1 \xrightarrow{s_2} \sigma_1'$. Since by our inductive hypothesis the lemma holds for both s_1 and s_2, it also holds for s.
- $s =$ while x do s_1 end;.
  By (4.5.10), $\sigma_1 \xrightarrow{s} \sigma_1'$ if and only if either $\neg\sigma_1(x)$ (in which case $\sigma_1 = \sigma_1'$, and the lemma holds for the same reasons as when $s = \varepsilon$) or $\sigma_1(x)$ and $\sigma_1 \xrightarrow{s_1;\ \text{while } x \text{ do } s_1 \text{ end}} \sigma_1'$. Since we have shown that the lemma holds for $s_1; s_2;$, it also holds for s.
- $s =$ cobegin *par* end;.
  By the grammar in fig. 4.1, *par* consists of a list of ⟨*op-call*⟩s o_i, which we showed above each individually satisfy the lemma. Therfore, if $\sigma_1 \vdash \mathsf{pre}(s)$, then by (4.5.12) there must be some o_i in *par* such that $\sigma_1 \vdash \mathsf{pre}(o_i)$. So it is also true that $\sigma_2 \vdash \mathsf{pre}(o_i)$, so $\sigma_2 \vdash \mathsf{pre}(s)$. Similarly, if $\sigma_1' \vdash \mathsf{post}(s)$ then it follows that $\sigma_1' \vdash \mathsf{post}(o_i)$ for every o_i in *par*. Thus, we have that $\sigma_2' \vdash \mathsf{post}(o_i)$ for each i and therefore $\sigma_2' \vdash \mathsf{post}(s)$. Therefore the lemma holds for s.

Therefore, by induction, we have that the lemma holds for all $s, \sigma_1, \sigma_2, \sigma_1', \sigma_2'$. □

Lemma 5.3. *Given two operation contracts* $o_1 = (i_1, \pi_1, pre_1, post_1, \mathcal{S}_1), o_2 = (i_2, \pi_2, pre_2, post_2, \mathcal{S}_2)$*, bodies* b_1, b_2 *such that* $(b_1 \mapsto o_1 \wedge b_1 \rightarrowtail o_1) \wedge (b_2 \mapsto o_2 \wedge b_2 \rightarrowtail o_2)$*, and state* σ*,*

$$\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma) \Rightarrow \mathcal{A}_\sigma(b_1) \ddagger \mathcal{A}_\sigma(b_2).$$

Proof. Let bodies b_1 and b_2 be in realizations with fields F_1, F_2 and interference correspondences $\mathcal{I}_1, \mathcal{I}_2$, respectively. Without loss of generality, we work only with b_1, o_1, and $\mathcal{I}_1$, for which we will omit subscripts; the proof and definitions are analogous for b_2, o_2, and $\mathcal{I}_2$.

By eq. (4.6.1), $\mathcal{A}_\sigma(b)$ is the application of function $\hat{\mathcal{I}}$ to the actual effect of the statements s that make up b.

To obtain a useful instance of lemma 2.9, we first define the function j_σ as follows:

$$j_\sigma(x) = \begin{cases} \mathcal{I}(\sigma, x) & x \in \hat{\mathcal{P}}(F) \\ x & \text{otherwise} \end{cases}$$

From this, we observe that $\hat{\mathcal{I}}(\sigma, e) = \mathcal{R}\big(J_\sigma(e.A), J_\sigma(e.P)\big)$ where J_σ is defined relative to j_σ as in lemma 2.9. We further observe that $\mathcal{A}_\sigma\big(o(p)\big) = \mathcal{S}(\sigma)[p \to \pi]$ for some sequence of arguments p; that is, by definition 3.12, $\hat{\mathcal{I}}\big(\sigma, \mathcal{A}_\sigma(s)\big) \sqsubseteq \mathcal{S}(\sigma)[p \to \pi]$.

Next, by lemma 2.7, if $\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma)$, then $\mathcal{A}_\sigma\big(o_1(p_1)\big) \ddagger \mathcal{A}_\sigma\big(o_2(p_2)\big)$. By lemma 2.9 (and lemmas 2.2 and 2.7), if $\mathcal{A}_\sigma\big(o_1(p_1)\big) \ddagger \mathcal{A}_\sigma\big(o_2(p_2)\big)$ then $\mathcal{A}_\sigma(b_1) \ddagger \mathcal{A}_\sigma(b_2)$. Therefore, $\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma) \Rightarrow \mathcal{A}_\sigma(b_1) \ddagger \mathcal{A}_\sigma(b_2)$. □

Lemma 5.4. *Operation calls can be replaced with any correct body without compromising correctness. Formally,*

$$\forall s, t, op, o, p, b : (s; o(p); t \mapsto op \wedge b \mapsto o) \Rightarrow (s; b; t \mapsto op)$$

Proof. By eq. (4.5.8),

$$s; o(p); t \mapsto op \Rightarrow$$
$$\big(\forall \sigma, \sigma', \sigma'' : \sigma \xrightarrow{s} \sigma' : \sigma' \xrightarrow{o(p)} \sigma'' \Rightarrow \sigma'' \vdash \mathsf{pre}(t)\big).$$

By definition 3.11 and eq. (4.5.1),

$$b \mapsto o \Rightarrow$$
$$\big(\forall \sigma, \sigma' : \sigma \xrightarrow{b} \sigma' \Rightarrow \sigma \xrightarrow{o(p)} \sigma'\big) \Rightarrow$$
$$\forall \sigma, \sigma' : \big(\sigma \vdash o.pre \Rightarrow \sigma \vdash \mathsf{pre}(b)\big) \wedge \big(\sigma' \vdash \mathsf{post}(b) \Rightarrow \sigma' \vdash o.post\big).$$

Therefore,

$$s; o(p); t \mapsto op \Rightarrow$$
$$\big(\forall \sigma, \sigma', \sigma'' : \sigma \xrightarrow{s} \sigma' : \sigma' \xrightarrow{b} \sigma'' \Rightarrow \sigma'' \vdash \mathsf{pre}(t)\big).$$

Then, by definition 3.11 and eq. (4.5.8),

$$\big(s; o(p); t \mapsto op \wedge b \mapsto o\big) \Rightarrow \big(s; b; t \mapsto op\big).$$

□

Define $\hat{o}_{p,q}$ be the operation contract induced by operation contracts p and q as follows (when p and q are understood, they are left out and we refer simply to $\hat{o}$):

$$\hat{o}_{p,q} = \begin{pmatrix} \pi : p.\pi \circ q.\pi, \\ pre : \mathsf{pre}(p(p.\pi); q(q.\pi)) \vee \mathsf{pre}(q(q.\pi); p(p.\pi)), \\ post : \mathsf{post}(p(p.\pi); q(q.\pi)) \wedge \mathsf{post}(q(q.\pi); p(p.\pi)), \\ \mathcal{S}(\sigma) : p.\mathcal{S}(\sigma) \sqcup q.\mathcal{S}(\sigma) \end{pmatrix} \tag{5.0.1}$$

Lemma 5.5.

$$\forall s, t_1, t_2, r, op : s; t_1; t_2; r \mapsto op : s; \hat{o}_{t_1,t_2}; r \mapsto op.$$

Proof. Suppose that $s; t_1; t_2; r \mapsto op$ for some operation contract $op = (i, \pi, pre, post, \mathcal{S})$, and that

$$\forall \sigma, \sigma' : \sigma \vdash pre \wedge \sigma \xrightarrow{s} \sigma' : \mathcal{A}'_\sigma(t_1) \ddagger \mathcal{A}'_\sigma(t_2).$$

Then, by definition 3.11 and eq. (4.5.1),

$$\forall \sigma, \sigma' : \sigma \xrightarrow{s;t_1;t_2;r} \sigma' \Rightarrow \sigma \vdash pre \wedge \sigma' \vdash post.$$

By eq. (4.5.8), the state of the program after s satisfies both $\mathsf{post}(s)$ and $\mathsf{pre}(t_1; t_2; r)$ and the state after $s; t_1; t_2$ satisfies both $\mathsf{post}(s; t_1; t_2)$ and $\mathsf{pre}(r)$. It is also true that

$$\mathsf{pre}(t_1; t_2; r) \Rightarrow \mathsf{pre}(t_1; t_2) \wedge \mathsf{post}(s; t_1; t_2) \Rightarrow \mathsf{post}(t_1; t_2).$$

By eq. (5.0.1),

$$\begin{aligned} \mathsf{pre}(\hat{o}_{t_1,t_2}) &= (\mathsf{pre}(t_1; t_2) \vee \mathsf{pre}(t_2; t_1)) \wedge \\ \mathsf{post}(\hat{o}_{t_1,t_2}) &= (\mathsf{post}(t_1; t_2) \wedge \mathsf{post}(t_2; t_1)). \end{aligned}$$

Clearly, then, $\mathsf{pre}(t_1; t_2) \Rightarrow \mathsf{pre}(\hat{o}_{t_1,t_2})$ and $\mathsf{post}(\hat{o}_{t_1,t_2}) \Rightarrow \mathsf{post}(t_1; t_2)$. Therefore, whatever state the program is in after s (which we saw above must satisfy $\mathsf{pre}(t_1; t_2; r)$ and therefore $\mathsf{pre}(t_1; t_2)$) also satisfies $\mathsf{pre}(\hat{o}_{t_1,t_2})$. Further, whatever state the program is in after $s; t_1; t_2$ (which we saw above must satisfy $\mathsf{post}(s; t_1; t_2)$ and therefore $\mathsf{post}(t_1; t_2)$) also satisfies $\mathsf{post}(\hat{o}_{t_1,t_2})$. Therefore,

$$\sigma \xrightarrow{s;\hat{o}_{t_1,t_2};r} \sigma' \Rightarrow \sigma \xrightarrow{s;t_1;t_2;r} \sigma',$$

and therefore

$$s; t_1; t_2; r \mapsto op \Rightarrow s; \hat{o}_{t_1,t_2}; r \mapsto op.$$

□

Theorem 5.1 (Non-interfering Statements Commute). *For any two operations o_1, o_2 with valid bodies s_1, s_2, if*

$$\forall \sigma, p_1, p_2 :$$
$$\big(\sigma \vdash \mathsf{pre}(o_1(p_1); o_2(p_2)) \vee \mathsf{pre}(o_2(p_2); o_1(p_1)) : o_1.\mathcal{S}(\sigma) \ddagger o_2.\mathcal{S}(\sigma)\big),$$

then

1. *$s_1; s_2$ is a valid operation body for $\hat{o}_{o_1,o_2}$ if and only if $s_2; s_1$ is a valid operation body for $\hat{o}_{s_1,s_2}$. That is,*

$$(s_1; s_2 \rightarrowtail \hat{o} \wedge s_1; s_2 \mapsto \hat{o}) \Leftrightarrow (s_2; s_1 \rightarrowtail \hat{o} \wedge s_2; s_1 \mapsto \hat{o})$$

2. `cobegin` $s_1 \| s_2$ `end` *is a valid operation body for $\hat{o}_{o_1,o_2}$ if and only if either $s_1; s_2$ is a valid operation body for $\hat{o}_{o_1,o_2}$. That is,*

$$(\texttt{cobegin } s_1 \| s_2 \texttt{ end} \rightarrowtail \hat{o} \wedge \texttt{cobegin } s_1 \| s_2 \texttt{ end} \mapsto \hat{o}) \Leftrightarrow$$
$$\big(s_1; s_2 \rightarrowtail \hat{o} \wedge s_1; s_2 \mapsto \hat{o}\big).$$

Proof. First, by eq. (4.6.6), definition 3.11, and theorems 2.1 and 2.2,

$$s_1; s_2 \rightarrowtail \hat{o} \wedge s_2; s_1 \rightarrowtail \hat{o} \wedge$$
$$\texttt{cobegin } s_1 \| s_2 \texttt{ end} \rightarrowtail \hat{o},$$

i.e., the actual effect of the sequential composition of the two statements in either order is covered by the specified effect in $\hat{o}$, as is the actual effect of their parallel composition.

We show that the implements relation ($\mapsto$) holds by induction on the "abstraction level" of the statements (*i.e.*, how far removed they are from primitive operations).

Base Case

The base case is when s_1 and s_2 both consist of a single call to `Prim` (and o_1, o_2 are equivalent to `Prim`). First, $\mathsf{pre}(s_1; s_1)$ and $\mathsf{pre}(s_2; s_2)$ are both true (*i.e.*, $\hat{o}_{o_1,o_2}.pre = \mathsf{true}$). By (`Prim`), each sequence of parameters p_1 and p_2 is divided into two subsequences, π_A and π_P. Additionally, the value of each object in π_P is the same after the operation as it was before the operation. Next, because `Prim` places *entire objects* into the A and P sets of the (constant) specified effect $\mathcal{S}$ (*i.e.*, the objects are not divided based on their partitions), the two calls to `Prim` are noninterfering ($\mathcal{A}_\sigma(s_1) \ddagger \mathcal{A}_\sigma(s_2)$) only when either no objects are shared between the two calls or all of the shared objects are in π_P for both calls. Therefore, by the Frame Rule (4.5.2), all shared objects have the same value after $s_1; s_2$ as they have after $s_2; s_1$—specifically, their value has not changed. Moreover, by lemma 5.1 and eq. (4.5.2), any object not an argument in either s_1 or s_2 has the same value before and after $s_1; s_2$ and $s_2; s_1$. Finally, we consider those objects that appear in π_A for either s_1 or s_2 (but not both!)—without loss of generality

consider those objects that appear in s_1 but not s_2. Each object x that appears in s_1 but not s_2 has, after s_1 is complete, value x'. But x does not appear in s_2, so its value after s_2 is *also* x', so the value of x after $s_1; s_2$ and $s_2; s_1$ is the same: x'. Therefore, if $\mathcal{A}_\sigma(s_1) \ddagger \mathcal{A}_\sigma(s_2)$, $\mathsf{post}(s_1; s_2) = \mathsf{post}(s_1; s_2)$; therefore by (4.5.8) $s_1; s_2 \mapsto \hat{o} \Leftrightarrow s_2; s_2 \mapsto \hat{o}$.

Since `Prim` is atomic, the parallel execution of two calls to `Prim` is exactly equivalent to one of the sequential orderings of those calls (which themselves are equivalent, as shown above); therefore, $s_1; s_2 \mapsto \hat{o} \Leftrightarrow$ `cobegin` $s_1 \| s_2$ `end` $\mapsto \hat{o}$.

Inductive Step

Assume the theorem holds for each consitituent statement in s_1 and s_2 which are valid bodies for some operations o_1, o_2 with preconditions pre_1, pre_2, postconditions $post_1, post_2$, and specified effects $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. Further assume σ is such that $\mathcal{S}_1(\sigma) \ddagger \mathcal{S}_2(\sigma)$ and $\sigma \vdash pre_1 \vee pre_2$.

We adopt the following conventions and notations for the remainder of the proof:

- Every statement is an operation call. This does not reduce generality because any other kind of statements (such as conditional statements and loop) can be refactored as an operation contract and associated body.
- An operation body s is treated as the sequential composition of statements $s[1]; s[2]; \ldots; s[|s|]$.
- $s[i, j)$ is the subsequence of statements in s from the ith through $(j-1)$th statements, well-defined whenever $1 \leq i \leq j \leq |s| + 1$.[8]
- b_s is a valid operation body for the operation call in statement s.

By eq. (4.6.6), lemmas 2.7 and 2.9, and theorems 2.1 and 2.2,

$$\forall \sigma, i, j : \sigma \xrightarrow{s_1[1,i)} \sigma_i \wedge \sigma \xrightarrow{s_2[1,j)} \sigma_j : \mathcal{A}_{\sigma_i}(s_1[i]) \ddagger \mathcal{A}_{\sigma_j}(s_2[j]). \tag{5.0.2}$$

Without loss of generality, we consider $s_1; s_2$ (the same proof applies to $s_2; s_1$). The statements $s_1[|s_1|]$ and $s_2[1]$ induce $\hat{o}_s$ (5.0.1). Let σ' be such that $\sigma \xrightarrow{s_1[1,|s_1|)} \sigma'$. By lemma 5.4, for all op,

$$s_1; s_2 \mapsto op \Rightarrow b_{1[1]}; b_{1[2]}; \ldots; b_{1[|s_1|]}; b_{2[1]}; b_{2[2]}; \ldots; b_{2[|s_2|]} \mapsto op$$

By (5.0.2),

$$\mathcal{A}_{\sigma'}(s_1[|s_1|]) \ddagger \mathcal{A}_{\sigma'}(s_2[1]).$$

So, by inductive hypothesis,

$$b_{1[|s_1|]}; b_{2[1]} \mapsto \hat{o}_s \Leftrightarrow b_{2[1]}; b_{1[|s_1|]} \mapsto \hat{o}_s.$$

[8] Observe that $\forall i, j : 1 \leq i \leq j \leq |s| + 1 : \big|s[i, j)\big| = i - j$.

By eqs. (4.5.7) and (4.5.8) and lemma 5.5,

$$\begin{aligned} s_1; s_2 \mapsto op &\Rightarrow s_1[1, |s_1|); s_1[|s_1|]; s_2[1]; s_2[1, |s_2| + 1) \mapsto op \\ &\Rightarrow s_1[1, |s_1|); \hat{o}_s; s_2[1, |s_2| + 1) \mapsto op. \end{aligned}$$

By lemma 5.4,

$$b_{1[1]}; b_{1[2]}; \ldots; b_{1[|b_1|-1]}; b_{2[1]}; b_{1[|s_1|]}; b_{2[2]}; b_{2[3]}; \ldots; b_{2[|s_2|} \mapsto op.$$

Now, the statements $s_1[|s_1| - 1]$ and $s_2[1]$ induce $\hat{o}'_s$; by the same reasoning as above we have that

$$b_{1[1]}; b_{1[2]}; \ldots; b_{1[|b_1|-2]}; b_{2[1]}; b_{1[|s_1|-1]}; b_{1[|s_1|]}; b_{2[2]}; b_{2[3]}; \ldots; b_{2[|s_2|} \mapsto op,$$

and we can continue applying the same reasoning to sift $b_{2[1]}$ to the beginning of the sequence of statements:

$$b_{2[1]}; b_{1[1]}; b_{1[2]}; \ldots; b_{1[|s_1|]}; b_{2[2]}; b_{2[3]}; \ldots; b_{2[|s_2|]} \mapsto op.$$

Furthermore, we can sift each statement $b_{2[i]}$ to the front of the $b_{1[j]}$'s, and we have that

$$b_{2[1]}; b_{2[2]}; \ldots; b_{2[|s_2|]}; b_{1[1]}; b_{1[2]}; \ldots; b_{1[|s_1|]} \mapsto op$$

Since $b_{1[1]}; b_{1[2]}; \ldots; b_{1[|s_1|]} \mapsto s_1$ and $b_{2[1]}; b_{2[2]}; \ldots; b_{2[|s_2|]} \mapsto s_2$, it follows that

$$\sigma \xrightarrow{b_{2[1]};b_{2[2]};\ldots;b_{2[|s_2|]};b_{1[1]};b_{1[2]};\ldots;b_{1[|s_1|]}} \sigma' \Rightarrow \sigma \xrightarrow{s_2;s_1} \sigma'.$$

Therefore, by definition 3.11,

$$b_{2[1]}; b_{2[2]}; \ldots; b_{2[|s_2|]}; b_{1[1]}; b_{1[2]}; \ldots; b_{1[|s_1|]} \mapsto op \Rightarrow s_2; s_1 \mapsto op.$$

By transitivity of $\Rightarrow$, then, we have that

$$s_1; s_2 \mapsto op \Rightarrow s_2; s_1 \mapsto op$$

Therefore,

$$s_1; s_2 \mapsto \hat{o}_{o_1,o_2} \Rightarrow s_2; s_1 \mapsto \hat{o}_{o_1,o_2}.$$

By swapping s_1 and s_2 and performing the same sifting process as above, we conclude

$$s_2; s_1 \mapsto \hat{o}_{o_1,o_2} \Rightarrow s_1; s_2 \mapsto \hat{o}_{o_1,o_2}.$$

Therefore,

$$s_1; s_2 \mapsto \hat{o} \Leftrightarrow s_2; s_1 \mapsto \hat{o}.$$

Through the sifting process, it was established that every possible interleaving of the consituent statements in s_1 and s_2 is *also* a valid operation body for $\hat{o}_{o_1,o_2}$. Therefore, if $s_1; s_2$ is a valid operation body for $\hat{o}$, then `cobegin` $s_1 \| s_2$ `end` is also a valid operation body for $\hat{o}_{o_1,o_2}$. □

5.1 Implications

The effects calculus and programming model have interesting implications for writing and reasoning about parallel programs. In particular, theorem 5.1 implies that non-interfering operation calls commute *even when their specifications indicate otherwise*. Consider the operation contracts for *rem* and *add* in eqs. (5.1.2) and (5.1.3) that operate on a Queue (with entries of type T) with type specification as in eq. (5.1.1). Each object of type $\texttt{Queue}_T$ has a partition with two pieces named h and t, *i.e.*, $\forall q : [q : \texttt{Queue}_T] : \mathcal{P}(q) = \{h, t\}$.

$$\texttt{Queue}_T = \big(T^*, \langle\rangle, \{\mathit{enqueue}, \mathit{dequeue}, \mathit{isEmpty}\}\big) \tag{5.1.1}$$

An object of type $\texttt{Queue}_T$ is a string with entries of type T.

$\texttt{Queue}_T.v_0$ is the empty string.

$\texttt{Queue}_T$ provides these operations (their contracts are as would be expected).

$$\mathit{rem} = \begin{pmatrix} i : \texttt{Remove\_An\_End}, \\ \pi : \langle q, x\rangle, \\ \mathit{pre} : [q : \texttt{Queue}_T] \wedge |q| > 0, \\ \mathit{post} : q_0 = q \circ \langle x\rangle \vee q_0 = \langle x\rangle \circ q, \\ \mathcal{S}(\sigma) : (\{q@h\} \cup \mathcal{P}(x), \emptyset) \end{pmatrix} \tag{5.1.2}$$

$$\mathit{add} = \begin{pmatrix} i : \texttt{Add\_An\_End}, \\ \pi : \langle q, y\rangle, \\ \mathit{pre} : [q : \texttt{Queue}_T] \wedge [y : T], \\ \mathit{post} : q = q_0 \circ \langle y_0\rangle \vee q = \langle y_0\rangle \circ q_0, \\ \mathcal{S}(\sigma) : (\{q@t\} \cup \mathcal{P}(y), \emptyset) \end{pmatrix} \tag{5.1.3}$$

Next, consider the program fragment below in which objects q, u, and v are initialized previously.

Listing 5.1. Program fragment demonstrating utility of theorem 5.1.

```
remove_An_End(q, u);
Add_An_End(q, v);
```

Assume the program is in the following state at the beginning of the fragment:

$$q = \langle 10, 20, 30\rangle \wedge u = 4.$$

By looking solely at the behavioral specifications of the two operations (*i.e.*, the *pre* and *post* in the contracts), it can be shown (via (4.5.8)) that the program is in one of the following states after the two statements:

- $q = \langle 4, 10, 20 \rangle \wedge v = 30$
- $q = \langle 20, 30, 4 \rangle \wedge v = 10$
- $q = \langle 4, 20, 30 \rangle \wedge v = 10$
- $q = \langle 10, 20, 30 \rangle \wedge v = 4$

However, the possible states can be whittled down by recognizing that the two operation calls are *non-interfering*. Using theorem 5.1, we recognize that the program above (rather, the sequential composition of two valid bodies for `Remove_An_End` and `Add_An_End`) is a valid operation body for the following contract (5.1.4).

$$rot = \left(\begin{array}{l} i : \texttt{Rotate\_Once}, \\ \pi : \langle q, u, v \rangle, \\ pre : [q : \texttt{Queue}_T] \wedge [u : T] \wedge [v : T], \\ post : \left(\begin{array}{c} \big(q = \langle u_0 \rangle \circ q_0[0, |q| - 1) \wedge \langle v \rangle = q_0[|q| - 1, |q|)\big) \vee \\ \big(q = q_0[1, |q|) \circ u_0 \wedge \langle v \rangle = q_0[0, 1)\big) \end{array} \right), \\ \mathcal{S}(\sigma) : \big(\mathcal{P}(q) \cup \mathcal{P}(u) \cup \mathcal{P}(v), \emptyset\big) \end{array} \right) \tag{5.1.4}$$

Therefore, we can conclude that the state at the end of the program in listing 5.1 is one of the following two states (rather than one of the four above):

- $q = \langle 4, 10, 20 \rangle \wedge v = 30$
- $q = \langle 20, 30, 4 \rangle \wedge v = 10$

In fact, the theorem leads to even a stronger conclusion. The two states above are also the only possible states after either of the following two program fragments given the same initial conditions:

```
Add_An_End(q, v);
remove_An_End(q, u);
```

```
cobegin
  Remove_An_End(q, v);
  Add_An_End(q, u);
end
```

Moreover, observe that except for type restrictions, $rot.pre \equiv \mathsf{true}$. This implies that, even when $|q| = 0$, the parallel execution of `Add_An_End` and `Remove_An_End` will leave the program in one of the two states above.

6 Conclusion

The plausibility and utility of an effects calculus to abstractly characterize the conditions under which parallel execution is safe was demonstrated, and a model of programming that leverages the theoretical results from the calculus was proposed. Several results were proven through the effects calculus and programming

model, most notably that non-interfering statements commute even if the behavioral specifications of those statements do not. A proof rule for the modular verification of correctness for simple fork-join parallel programs was proposed and proved sound using those results.

It is expected that the effects caluclus can be expanded to model addiitonal parallel programming primitives beyond the `cobegin` statement, such as `await`, for example by adding a third set to effects that represents "atomically affected".