A Generalised Union of Rely–Guarantee and Separation Logic using Permission Algebras

Vincent Jackson
The University of Melbourne, Australia

Toby Murray
The University of Melbourne, Australia

Christine Rizkallah
The University of Melbourne, Australia

Abstract

This paper describes GenRGSep, an Isabelle/HOL library for the development of RGSep logics using a general algebraic state model. In particular, we develop an algebraic state models based on resource algebras that assume neither the presence of unit resources or the cancellativity law. If a new resource model is required, its components need only be proven an instance of a permission algebra, and then they can be composed together using tuples and functions.

The proof of soundness is performed by Vafeiadis' operational soundness method. This method was originally formulated with respect to a concrete heap model. This paper adapts it to account for the absence of both units as well as the cancellativity law.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Concurrency; Theory of computation → Separation logic

Keywords and phrases verification, concurrency, rely-guarantee, separation logic, resource algebras

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material GeneralRGSep
Software (Formalisation): `https://github.com/vjackson725/GeneralRGSep/tree/itp24`

1 Introduction

This paper describes GenRGSep, an Isabelle/HOL [35] library for the development of RGSep logics [38, **?**], using a general algebraic state model. This library bases its state model on permission algebras, the most generic form of resource algebra [9]. Permission, multi-unit and single-unit separation algebras [13, 3] are developed as a type-class hierarchy that enables the integration of permissions and values into a common algebraic language. This allows for useful resource models to be developed from simple components and then automatically applied to RGSep. The soundness of GenRGSep has been formally verified by an operational soundness proof that generalises a method of Vafeiadis' [37] to work without the cancellativity law.

This project is motivated, in part, by a desire to have a general separation logic framework for verifying concurrent code in Isabelle/HOL. There are several very general frameworks for the development of separation logics for the verification of concurrent programs in other theorem provers: for example, VST [4] and Iris [25]. There is, as yet, none in Isabelle/HOL. While this work is not yet as comprehensive as these projects, we hope it will provide a good foundation for the future development of such projects in Isabelle/HOL.

In order to achieve generality, we develop separation logic from resource algebras, an abstract algebraic model of resources [9]. A resource algebra is a specific sort of partial semigroup or monoid, which defines a model of separated resources. There are many variations, usually on which unit elements the algebra is guaranteed to have (no guarantee, some unit

© Vincent Jackson and Toby Murray and Christine Rizkallah;
licensed under Creative Commons License CC-BY 4.0

for every resource, or a universal unit), and whether the logic admits the cancellativity law. The resource algebra approach has been used in many projects [13, 26, 3, 28, 25, 29].

All Isabelle/HOL separation algebras up to this point assume the existence of resources that act like a unit for resource addition. Similar to VST [3], GenRGSep is based on a type-class hierarchy of resource algebras which includes permission algebras, that do not require units to exist. This approach treats permissions as first-class citizens that compositionally integrate with other resources and into larger structures.

The RGSep family of logics [38, **?**] combines the rely–guarantee method [24, 23] and concurrent separation logic [31, 8]. The rely–guarantee method is a method of concurrent program verification that requires rely and guarantee relations for each proces. The rely relation establishes how the shared state can be changed by the environment (other processes), and the guarantee relation establishes how the current process can change the shared state. Concurrent separation logic is a method of concurrent program verification that requires all state to either be local: in which case it can be separated into pieces which are acted upon by parallel processes separately, or shared, in which case, it must obey a resource invariant. RGSep combines the benefits of both methods: the separate reasoning about local state of concurrent separation logic and the fine-grained concurrency of rely–guarantee.

This paper introduces GenRGSep, a generalisation of RGSep to non-cancellative resource algebras without units, and prove its soundness. Our paper is structured as follows: in Section 2, we review the construction of resource algebras and describe the particular issues we encountered in translating them to Isabelle/HOL. In Section 3, we describe our GenRGSep language, which in addition to standard programming constructs, includes external nondeterminism and non-deterministic **do** statements [20, 21], and the RGSep logic for it. We also review explicit stabilisation [**?**, 40], which simplifies reasoning about stability. In Section 4.2, we discuss the soundness proof for GenRGSep, using an extension of a method by Vafeiadis [37]. This method encounters some problems with the combination of non-cancellative resource models and external-nondeterminism, which we demonstrate how to address.

Contributions

The paper presents the following contributions:

1. an encoding of an Isabelle/HOL type-class hierarchy for permission and separation algebras that allows for the compositional construction of resource models;
2. the formalisation of the soundness of RGSep over general permission algebras in Isabelle/HOL; and
3. a re-examination of Vafeiadis' operational soundness method, showing how to extend it to non-cancellative resource algebras.

2 Formalising the Foundations

We construct separation logic from the foundations of an algebraic model of separated resources; these are often called separation algebras or resource algebras [9, 3]. In particular, we define three structures as type-classes in Isabelle/HOL: permission algebras, multi-unit separation algebras, and (single-unit) separation algebras. We will refer to these collectively as *resource algebras*, and to the elements of these algebras as *resources*. The axioms for these structures are listed in Figure 1.

class perm-alg$(\#, +)$ =
  partial-add-assoc: $a \# b \longrightarrow b \# c \longrightarrow a \# c \longrightarrow (a + b) + c = a + (b + c)$
  partial-add-commute: $a \# b \longrightarrow a + b = b + a$
  disjoint-sym: $a \# b \longrightarrow b \# a$
  disjoint-add-rightL: $b \# c \longrightarrow a \# b + c \longrightarrow a \# b$
  disjoint-add-right-commute: $b \# c \longrightarrow a \# b + c \longrightarrow b \# a + c$
  positivity: $a \# a' \longrightarrow b \# b' \longrightarrow a + a' = b \longrightarrow b + b' = a \longrightarrow a = b$

class multi-sep-alg$(\#, +, \mathsf{unitof})$ =
  perm-alg$(\#, +)$ +
  unitof-disjoint: $(\mathsf{unitof}\ a) \# a$
  unitof-add: $(\mathsf{unitof}\ a) \# b \longrightarrow (\mathsf{unitof}\ a) + b = b$

class sep-alg$(\#, +, \mathsf{unitof}, 0)$ =
  multi-sep-alg$(\#, +, \mathsf{unitof})$ +
  zero-disjoint : $0 \# x$
  zero-unit : $0 + x = x$

class cancel-perm-alg$(\#, +)$ =
  perm-alg$(\#, +)$ +
  cancel-right: $a \# c \longrightarrow b \# c \longrightarrow a + c = b + c \longrightarrow a = b$

Figure 1 Resource algebra axioms

2.1 Resource Algebras

A *permission algebra* (perm-alg), is a partial commutative semigroup, where $+$ is the semigroup operator and $\#$ (disjoint) specifies when $+$ is defined. The $+$ operator and $\#$ obey a number of laws, namely: the disjointness relation is commutative, the disjoint parts of a resource remain disjoint to (other) resources disjoint to the whole (that is, $y \# z$ and $x \# y + z$ implies $x \# y$), and the disjoint parts of a resource remain disjoint to the other parts of that resource when added to (another) resource disjoint from the whole (that is, $y \# z$ and $x \# y + z$ implies $x + y \# z$). In addition, resources that contain each other as parts are equal (positivity).

A *multi-unit separation algebra* (multi-sep-alg) is a permission algebra with an additional operation $\mathsf{unitof} : \alpha \Rightarrow \alpha$, which produces the unit of the given resource of the algebra. A *separation algebra* (sep-alg) is a multi-unit separation algebra with the single unit, 0.

Resources form an order: a resource is strictly less than another ($\prec$) if they are not equal and there is some resource that adds to the first to make the second ($a \neq b \wedge (\exists c.\ a + c = b)$). A resource is less than or equal to another ($\preceq$) if they are equal or there is some third resource that adds to the first to make the second ($a = b \vee (\exists c.\ a + c = b)$). These definitions form an order, but this order is not necessarily Isabelle/HOL's standard order instance. Note that the order is anti-symmetric by virtue of the law of positivity. Note also that, in a multi-unit separation algebra, we have that $a \preceq b \longleftrightarrow (\exists c.\ a \# c \wedge a + c = b)$, because units are guaranteed to exist.

Permission algebras are useful for representing values with constraints on how those values may be used. The classic model is fractional permissions [7, 6] ($\mathbf{P}_{\mathbb{Q}}$ in Figure 2), where 1 represents the ability to change the value, and fractional quantities ($0 < x < 1$) represent only the ability to read the value. By placing this permission in a tuple with the discrete

Fractional Permissions

typedef $\mathbf{P}_{\mathbb{Q}} \coloneqq \{x \in \mathbb{Q}.\ 0 < x \leq 1\}$

instance $\mathbf{P}_{\mathbb{Q}}$: perm-alg

$$\begin{array}{rcl} a \# b & \coloneqq & a + b \leq 1 \\ a + b & \coloneqq & \min\ (a + b)\ 1 \end{array}$$

Multiplicative Unit

datatype $\mathbb{1} = \mathbb{1}$

instance $\mathbb{1}$: perm-alg

$$\begin{array}{rcl} a \# b & \coloneqq & \bot \\ a + b & \coloneqq & \mathsf{undefined} \end{array}$$

Discrete Type

typedef $\alpha\,\mathtt{discr} \coloneqq (\mathsf{UNIV} : \alpha\ \mathtt{set})$

instance $\alpha\,\mathtt{discr}$: multi-sep-alg

$$\begin{array}{rcl} a \# b & \coloneqq & a = b \\ a + b & \coloneqq & a \\ \mathsf{unitof}\ a & \coloneqq & a \end{array}$$

Functions

instance $(\alpha \Rightarrow (\beta : \mathsf{perm\text{-}alg})) : \mathsf{perm\text{-}alg}$

$$\begin{array}{rcl} f \# g & \coloneqq & \forall x.\ (f\ x) \# (g\ x) \\ a + b & \coloneqq & \lambda x.\ (f\ x) + (g\ x) \end{array}$$

instance $(\alpha \Rightarrow (\beta : \mathsf{multi\text{-}sep\text{-}alg})) : \mathsf{multi\text{-}sep\text{-}alg}$

$$\mathsf{unitof}\ f \quad \coloneqq \quad \lambda x.\ \mathsf{unitof}\ (f\ x)$$

instance $(\alpha \Rightarrow (\beta : \mathsf{sep\text{-}alg})) : \mathsf{sep\text{-}alg}$

$$0 \quad \coloneqq \quad \lambda x.\ 0$$

Tuples

instance $((\alpha : \mathsf{perm\text{-}alg}) \times (\beta : \mathsf{perm\text{-}alg})) : \mathsf{perm\text{-}alg}$

$$\begin{array}{rcl} (a_1, a_2) \# (b_1, b_2) & \coloneqq & (a_1 \# b_1) \wedge (a_2 \# b_2) \\ (a_1, a_2) + (b_1, b_2) & \coloneqq & (a_1 + b_1, a_2 + b_2) \end{array}$$

instance $((\alpha : \mathsf{multi\text{-}sep\text{-}alg}) \times (\beta : \mathsf{multi\text{-}sep\text{-}alg})) : \mathsf{multi\text{-}sep\text{-}alg}$

$$\mathsf{unitof}\ (a_1, a_2) \quad \coloneqq \quad (\mathsf{unitof}\ a_1, \mathsf{unitof}\ a_2)$$

instance $((\alpha : \mathsf{sep\text{-}alg}) \times (\beta : \mathsf{sep\text{-}alg})) : \mathsf{sep\text{-}alg}$

$$0 \quad \coloneqq \quad (0, 0)$$

Option Type

datatype $\alpha\,\mathrm{option} = \mathsf{Some}\ \alpha \mid \mathsf{None}$

instance $(\alpha : \mathsf{perm\text{-}alg})\,\mathrm{option} : \mathsf{sep\text{-}alg}$

$$\begin{array}{rcl} a \# b & \coloneqq & \begin{array}{l} \textbf{case}\ (a, b)\ \textbf{of} \\ \quad (\mathsf{None}, b) \Rightarrow \mathrm{True} \\ \mid (a, \mathsf{None}) \Rightarrow \mathrm{True} \\ \mid (\mathsf{Some}\ x, \mathsf{Some}\ y) \Rightarrow (x \# y) \end{array} \\ a + b & \coloneqq & \begin{array}{l} \textbf{case}\ (a, b)\ \textbf{of} \\ \quad (\mathsf{None}, b) \Rightarrow b \\ \mid (a, \mathsf{None}) \Rightarrow a \\ \mid (\mathsf{Some}\ x, \mathsf{Some}\ y) \Rightarrow \mathsf{Some}\ (x + y) \end{array} \\ \mathsf{unitof} a & \coloneqq & \mathsf{None} \\ 0 & \coloneqq & \mathsf{None} \end{array}$$

Figure 2 Resource algebras instances for basic types

permission algebra (α `discr`, Figure 2), we obtain a model of these read-write values.

The multiplicative unit ($\mathbb{1}$, Figure 2) is another permission algebra; it acts as an indivisible permission. By placing this permission in a tuple with the discrete permission algebra (α `discr`, Figure 2), we obtain a model of non-duplicable values.

Using these type-classes, we can develop compositional instances for standard data-types, such as sums ($+$), tuples ($\times$), functions ($\Rightarrow$), and options (α `option`). Note that such instances have already been described in previous literature [13]. For this paper, it is sufficient to note the following: tuples inherit the (least specific) class of their components, options transform permission algebras to separation algebras, and functions inherit the class of their co-domain.

Given these instantiations, it becomes simple to create various complex separation algebras built from these simple ones. For example, the standard heap model is encoded as

$$(\alpha, \beta)\ \texttt{heap} \coloneqq \alpha \rightharpoonup (\beta\ \texttt{discr} \times \mathbb{1}),$$

where $\alpha \rightharpoonup \beta \coloneqq \alpha \Rightarrow \beta$ `option`. One key point to structuring our type-class hierarchy in this manner, distinguishing permission algebras from multi-unit algebras from separation algebras, is to allow *flexibility* for the proof engineer. For example: to change the previous heap instance to use fractional permissions [7, 6], one only needs to swap the $\mathbb{1}$ for $\mathbf{P}_{\mathbb{Q}}$.

3 The GenRGSep Logic

Using these resource algebras, we can construct a generic RGSep [38, **?**], a combination of separation logic and rely–guarantee, to reason over programs in resource models other than the standard heap model.

3.1 Language

The language (Figure 3) includes skip statements (**skip**), sequencing ($c_1; c_2$), parallel ($c_1 \parallel c_2$), and do-loops (**do** c **od**). Atomic statements ($\langle b \rangle$) are specified by a relation between states (b), and execution is *blocked* when the state is not in the domain of the relation. Inspired by CSP [21], we also distinguish between internal ($c_1 + c_2$) and external ($c_1 \,\square\, c_2$) non-determinism. We have chosen to include both internal and external non-determinism, and also relational atomic actions, because they provide a generic foundation upon which to build more concrete languages. The standard while-loop and if-then-else constructs can be encoded using external non-determinism, blocking guards, and do loops.

The state model for this language is composed of two parts: local and shared state. Thus we represent our state as a tuple, the left representing the local state and the right representing the shared state. Local state splits among the processes on parallel composition, whereas shared state is shared identically between the processes. Note that we choose *not* to explicitly model a store, because such an abstraction is not present in low-level state models.

The relational atomic statement, in particular, allows the definition of the specific atomic actions appropriate for whichever resource model the logic is instantiated with. For the same reason, the relation acts over a pair of local and shared state, which allows the resource models for local and shared state to differ. Moreover, this removes the requirement from standard RGSep that the shared part of the pre- and postconditions must pick out the shared state precisely.

Logical Variables

r, g, b (state relations)

p, q (state predicates)

Commands

$c ::= \mathbf{skip}$ (skip)

$\mid c_1; c_2$ (sequence)

$\mid c_1 + c_2$ (internal non-det.)

$\mid c_1 \,\square\, c_2$ (external non-det.)

$\mid c_1 \parallel c_2$ (parallel)

$\mid \langle b \rangle$ (relational atomic action)

$\mid \mathbf{do}\ c\ \mathbf{od}$ (do loop)

Abbreviations

$[p] \coloneqq \langle \lambda x\ y.\ p\ x \wedge x = y \rangle$ (guard)

$\mathbf{while}\, p\, \mathbf{do}\, c\, \mathbf{done} \coloneqq\ \mathbf{do}\ ([p]; c) \,\square\, [\neg p]\ \mathbf{od}$ (while loop)

$\mathbf{if}\, p\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\, \mathbf{fi} \coloneqq ([p]; c_1) \,\square\, ([\neg p]; c_2)$ (if-then-else)

Small-Step Semantics

$$\frac{(h, c_1) \sim\alpha\rightsquigarrow (h', c_1')}{(h, c_1; c_2) \sim\alpha\rightsquigarrow (h', c_1'; c_2)}\ \mathrm{Seq_L} \qquad \frac{}{(h, \mathbf{skip}; c_2) \sim\tau\rightsquigarrow (h, c_2)}\ \mathrm{Seq_R}$$

$$\frac{(h, c_1) \sim\alpha\rightsquigarrow (h', c_1')}{(h, c_1 + c_2) \sim\alpha\rightsquigarrow (h', c_1')}\ \mathrm{INDet_L} \qquad \frac{(h, c_2) \sim\alpha\rightsquigarrow (h', c_2')}{(h, c_2 + c_2) \sim\alpha\rightsquigarrow (h', c_2')}\ \mathrm{INDet_R}$$

$$\frac{}{(h, \mathbf{skip} \,\square\, c_2) \sim\tau\rightsquigarrow (h', c_2)}\ \mathrm{ENDetSkip_L} \qquad \frac{}{(h, c_1 \,\square\, \mathbf{skip}) \sim\tau\rightsquigarrow (h', c_1)}\ \mathrm{ENDetSkip_L}$$

$$\frac{(h, c_1) \sim\tau\rightsquigarrow (h', c_1')}{(h, c_1 \,\square\, c_2) \sim\tau\rightsquigarrow (h', c_1' \,\square\, c_2)}\ \mathrm{ENDetTau_L} \qquad \frac{(h, c_2) \sim\tau\rightsquigarrow (h', c_2')}{(h, c_2 \,\square\, c_2) \sim\tau\rightsquigarrow (h', c_1 \,\square\, c_2')}\ \mathrm{ENDetTau_R}$$

$$\frac{(h, c_1) \sim a\rightsquigarrow (h', c_1')}{(h, c_1 \,\square\, c_2) \sim a\rightsquigarrow (h', c_1')}\ \mathrm{ENDet_L} \qquad \frac{(h, c_2) \sim a\rightsquigarrow (h', c_2')}{(h, c_2 \,\square\, c_2) \sim a\rightsquigarrow (h', c_2')}\ \mathrm{ENDet_R}$$

$$\frac{}{(h, \mathbf{skip} \parallel \mathbf{skip}) \sim\tau\rightsquigarrow (h, c_2)}\ \mathrm{ParSkip}$$

$$\frac{(h, c_1) \sim\alpha\rightsquigarrow (h', c_1')}{(h, c_1 \parallel c_2) \sim\alpha\rightsquigarrow (h', c_1' \parallel c_2)}\ \mathrm{Par_L} \qquad \frac{(h, c_2) \sim\alpha\rightsquigarrow (h', c_2')}{(h, c_2 \parallel c_2) \sim\alpha\rightsquigarrow (h', c_1 \parallel c_2')}\ \mathrm{Par_R}$$

$$\frac{(h, c) \sim\alpha\rightsquigarrow (h', c')}{(h, \mathbf{do}\ c\ \mathbf{od}) \sim\alpha\rightsquigarrow (h', c'; \mathbf{do}\ c\ \mathbf{od})}\ \mathrm{DoStep} \qquad \frac{\neg\, \mathrm{enabled}\ c\ h}{(h, \mathbf{do}\ c\ \mathbf{od}) \sim\tau\rightsquigarrow (h', \mathbf{skip})}\ \mathrm{DoEnd}$$

$$\frac{b\ h\ h'}{(h, \langle b \rangle) \sim \mathsf{Upd}\rightsquigarrow (h', \mathbf{skip})}\ \mathrm{Atomic}$$

where ‘enabled c h’ holds when there is some head atomic command $\langle b \rangle$ in c where h is in the domain of b.

Figure 3 Language syntax and small-step semantics

3.2 Semantics

We give the language a small step semantics (Figure 3) with the relation $(h, c) \sim\alpha\rightsquigarrow (h', c')$. This should be interpreted as: starting with state h and program c, an α-step can be taken to state h' and program c'.

Steps are divided into two sorts of actions: τ-actions that represent internal decisions a process makes that are not directly observable by other processes and observable actions that are visible to other processes. Examples of τ-actions include the outcome of a non-deterministic choice and the end of a while loop. An example of an observable action is heap updates. This distinction is reflected by the fact that, when we connect these semantics to RGSep, it will be the observable actions that generate the guarantee. As is traditional, we will use the variable α to stand for any action and the variable a to stand for any observable action.

We only have one observable action: Upd, for atomic update actions. The distinction between internal and update commands is all that is necessary to prove soundness with respect to the operational semantics.

3.3 Separation Logic

We shallowly embed the predicates in Isabelle/HOL, rather than constructing a deeply embedded predicate language. The definitions of separating conjunction, separating implication, and the empty predicate are standard.

$$(*) : ((\alpha : \mathsf{perm\text{-}alg}) \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \Rightarrow \mathsf{bool})$$
$$p * q \coloneqq \lambda x.\ \exists x_1\ x_2.\ x_1 \# x_2 \wedge x = x_1 + x_2 \wedge p\ x_1 \wedge q\ x_2$$

$$(-\!\!*) : ((\alpha : \mathsf{perm\text{-}alg}) \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \Rightarrow \mathsf{bool})$$
$$p -\!\!* q \coloneqq \lambda h.\ \forall h_1.\ h \# h_1 \longrightarrow p\ h_1 \longrightarrow q\ (h + h_1)$$

$$\mathsf{emp} : ((\alpha : \mathsf{perm\text{-}alg}) \Rightarrow \mathsf{bool})$$
$$\mathsf{emp} \coloneqq \lambda x.\ x \# x \wedge (\forall a.\ a \# x \longrightarrow a + x = a)$$

Slightly less standard (notationally) is the connective $(*\!\wedge)$. This is defined as

$$(*\!\wedge) : ((\alpha : \mathsf{perm\text{-}alg}) \times (\beta : \mathsf{perm\text{-}alg}) \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \mathsf{bool})$$
$$p *\!\wedge q \coloneqq \lambda(x, y).\ \exists x_1\ x_2.\ x_1 \# x_2 \wedge x = x_1 + x_2 \wedge p\ (x_1, y) \wedge p\ (x_2, y),$$

and plays the role of the RGSep separating conjunction. We define this connective separately, as the standard permission algebra instance for tuples splits both the left and right parts of the tuple, not only the left part (the local resources), which is what RGSep requires.

Note also that, as we wish to formalise RGSep shallowly, we do not have Vafeiadis' boxed-predicates, which are a syntactic construct which demarcates predicates on the shared state. To regain the ease of reasoning that predicates acting on just the local or shared state provide, we define two liftings $\mathcal{L}$ and $\mathcal{S}$ from predicates on local and shared states, respectively, to predicates on the overall state

$$\mathcal{L} : (\alpha \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \mathsf{bool}) \qquad \mathcal{S} : (\beta \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \mathsf{bool})$$
$$\mathcal{L}\ p \coloneqq p \circ \mathrm{fst} \qquad \mathcal{S}\ p \coloneqq p \circ \mathrm{snd}$$

Unlike Vafeiadis, our $\mathcal{S}$ does not enforce that the local state is empty, as units are not guaranteed to exist in a permission algebra.

Using these, we can prove that $*\!\!\wedge$ is indeed the standard RGSep separating conjunction, by showing that the connective separates over local state, $\mathcal{L}\,p *\!\!\wedge \mathcal{L}\,q = \mathcal{L}(p * q)$, and is additive over shared state, $\mathcal{S}\,p *\!\!\wedge \mathcal{S}\,q = \mathcal{S}(p \wedge q)$.

3.4 Stabilisation Predicate Transformers

In our formalisation of RGSep, instead of adding side-conditions to the reasoning rules asserting that our pre- and postconditions are stable (invariant under the action of the rely, guarantee, or both), we instead use stabilisation predicate transformers [**?**, 40]. These ease reasoning about stability in RGSep, because they semi-distribute over $*\!\!\wedge$. This means that the stability of a predicate can be proven from the stability of its parts, unlike stability side-conditions, which do not distribute at all with $*\!\!\wedge$. They are defined using relational weakest precondition (**wlp**) and strongest postcondition (**sp**) predicate transformers [12], defined as follows: $\mathbf{wlp}\ r\ q \coloneqq (\lambda x.\ \forall y.\ r\ x\ y \longrightarrow q\ y)$ and $\mathbf{sp}\ r\ p \coloneqq (\lambda y.\ \exists x.\ r\ x\ y \wedge p\ x)$.

If we know we have a state that meets the predicate q, and we wish to know what the state could have been *before* the interference of the environment, we calculate the *weakest* assertion *stronger* than q and *stable* under r (the weakest stronger stable assertion, **wssa**). If we know we have a state that meets the predicate p, and we wish to know what the state might be *after* the interference of the environment, we calculate the *strongest* assertion *weaker* than p and *stable* under r (the strongest stable weaker assertion, **sswa**). These are defined as follows:

$$\mathbf{wssa}\ r\ q \coloneqq \mathbf{wlp}\ ((=) \times_{\mathcal{R}} r^*)\ q \qquad \mathbf{sswa}\ r\ p \coloneqq \mathbf{sp}\ ((=) \times_{\mathcal{R}} r^*)\ p.$$

where $r_1 \times_{\mathcal{R}} r_2 \coloneqq \lambda(x_1, x_2)\ (y_1, y_2).\ r_1\ x_1\ y_1 \wedge r_2\ x_2\ y_2$, and thus $((=) \times_{\mathcal{R}} r^*)$ is the relation that leaves the local state the same, and changes the shared state by the reflexive transitive closure of r.

Useful facts are that **wssa** is an interior operator and **sswa** is a closure operator,

$$\begin{array}{ll} \mathbf{wssa}\ r\ p \longrightarrow p & p \longrightarrow \mathbf{sswa}\ r\ p \\ \mathbf{wssa}\ r\ (\mathbf{wssa}\ r\ p) \longleftrightarrow \mathbf{wssa}\ r\ p & \mathbf{sswa}\ r\ (\mathbf{sswa}\ r\ p) \longleftrightarrow \mathbf{sswa}\ r\ p \\ (p \longrightarrow q) \wedge \mathbf{wssa}\ r\ p \longrightarrow \mathbf{wssa}\ r\ q & (p \longrightarrow q) \wedge \mathbf{sswa}\ r\ p \longrightarrow \mathbf{sswa}\ r\ q; \end{array}$$

they distribute or semi-distribute over the logical connectives

$$\begin{array}{ll} \mathbf{wssa}\ r\ (p \wedge q) \longleftrightarrow \mathbf{wssa}\ r\ p \wedge \mathbf{wssa}\ r\ q & \mathbf{sswa}\ r\ (p \wedge q) \longrightarrow \mathbf{sswa}\ r\ p \wedge \mathbf{sswa}\ r\ q \\ \mathbf{wssa}\ r\ p \vee \mathbf{wssa}\ r\ q \longrightarrow \mathbf{wssa}\ r\ (p \vee q) & \mathbf{sswa}\ r\ (p \vee q) \longleftrightarrow \mathbf{sswa}\ r\ p \vee \mathbf{sswa}\ r\ q \\ \mathbf{wssa}\ r\ p *\!\!\wedge \mathbf{wssa}\ r\ q \longrightarrow \mathbf{wssa}\ r\ (p *\!\!\wedge q) & \mathbf{sswa}\ r\ (p *\!\!\wedge q) \longrightarrow \mathbf{sswa}\ r\ p *\!\!\wedge \mathbf{sswa}\ r\ q; \end{array}$$

and they do not interact with local state

$$\mathbf{wssa}\ r\ (\mathcal{L}\ p) \longleftrightarrow \mathcal{L}\ p \qquad \mathbf{sswa}\ r\ (\mathcal{L}\ p) \longleftrightarrow \mathcal{L}\ p;$$

and this is the case even under a $*\!\!\wedge$ for **sswa**

$$\mathbf{sswa}\ r\ (\mathcal{L}\ p *\!\!\wedge q) \longleftrightarrow \mathcal{L}\ p *\!\!\wedge \mathbf{sswa}\ r\ q.$$

3.5 RGSep Reasoning

The RGSep judgement, $r, g \vdash \{\,p\,\}\ c\ \{\,q\,\}$, should be interpreted as follows: if we can rely on the environment changing the shared state according to r, and we start in a state that satisfies the precondition p, then successful execution of the program c will result in a state that satisfies the postcondition q, only changing the shared state according to g. The rules for this judgement can be found in Figure 4.

$$\frac{}{r, g \vdash \{\, p \,\}\ \mathbf{skip}\ \{\, \mathbf{sswa}\ r\ p \,\}}\ \text{Skip} \qquad \frac{r, g \vdash \{\, p_1 \,\}\ c_1\ \{\, p_2 \,\} \qquad r, g \vdash \{\, p_2 \,\}\ c_2\ \{\, p_3 \,\}}{r, g \vdash \{\, p_1 \,\}\ c_1; c_2\ \{\, p_3 \,\}}\ \text{Seq}$$

$$\frac{\begin{array}{c}\mathbf{sp}\ b\ (\mathbf{wssa}\ r\ p) \subseteq \mathbf{sswa}\ r\ q \\ \forall f.\ \mathbf{sp}\ b\ (\mathbf{wssa}\ r\ (p \mathbin{*\!\wedge} f)) \subseteq \mathbf{sswa}\ r\ (q \mathbin{*\!\wedge} f) \qquad \top \times_{\mathcal{R}} g \subseteq b\end{array}}{r, g \vdash \{\, \mathbf{wssa}\ r\ p \,\}\ \langle b \rangle\ \{\, \mathbf{sswa}\ r\ q \,\}}\ \text{Atomic}$$

$$\frac{\begin{array}{c}r, g \vdash \{\, p \,\}\ c_1\ \{\, q_1 \,\} \\ r, g \vdash \{\, p \,\}\ c_2\ \{\, q_2 \,\}\end{array}}{r, g \vdash \{\, p \,\}\ c_1 + c_2\ \{\, q_1 \vee q_2 \,\}}\ \text{INDet} \qquad \frac{\begin{array}{c}r, g \vdash \{\, p \,\}\ c_1\ \{\, q_1 \,\} \\ r, g \vdash \{\, p \,\}\ c_2\ \{\, q_2 \,\}\end{array}}{r, g \vdash \{\, p \,\}\ c_1 \,\square\, c_2\ \{\, q_1 \vee q_2 \,\}}\ \text{ENDet}$$

$$\frac{\begin{array}{c}(r \cup g_2), g_1 \vdash \{\, p_1 \,\}\ c_1\ \{\, q_1 \,\} \qquad (r \cup g_1), g_2 \vdash \{\, p_2 \,\}\ c_2\ \{\, q_2 \,\} \\ \mathbf{sswa}\ (r \cup g_2)\ q_1 \subseteq q_1' \qquad \mathbf{sswa}\ (r \cup g_1)\ q_2 \subseteq q_2'\end{array}}{r, (g_1 \cup g_2) \vdash \{\, p_1 \mathbin{*\!\wedge} p_2 \,\}\ c_1 \parallel c_2\ \{\, q_1' \mathbin{*\!\wedge} q_2' \,\}}\ \text{Par}$$

$$\frac{r, g \vdash \{\, \mathbf{sswa}\ r\ i \,\}\ c\ \{\, \mathbf{sswa}\ r\ i \,\}}{r, g \vdash \{\, i \,\}\ \mathbf{do}\ c\ \mathbf{od}\ \{\, \mathbf{sswa}\ r\ i \,\}}\ \text{Do} \qquad \frac{r, g \vdash \{\, p \,\}\ c\ \{\, q \,\} \qquad \mathbf{sswa}\ (r \cup g)\ f \subseteq f'}{r, g \vdash \{\, p \mathbin{*\!\wedge} f \,\}\ c\ \{\, q \mathbin{*\!\wedge} f' \,\}}\ \text{Frame}$$

$$\frac{r, g \vdash \{\, p_1 \,\}\ c\ \{\, q_1 \,\} \qquad r, g \vdash \{\, p_2 \,\}\ c\ \{\, q_2 \,\}}{r, g \vdash \{\, p_1 \vee p_2 \,\}\ c\ \{\, q_1 \vee q_2 \,\}}\ \text{Disj}$$

$$\frac{r, g \vdash \{\, p_1 \,\}\ c\ \{\, q_1 \,\} \qquad r, g \vdash \{\, p_2 \,\}\ c\ \{\, q_2 \,\} \qquad \text{for all local states } h_1, \mathbf{cancellative}\ h_1}{r, g \vdash \{\, p_1 \wedge p_2 \,\}\ c\ \{\, q_1 \wedge q_2 \,\}}\ \text{Conj}$$

$$\frac{r', g' \vdash \{\, p' \,\}\ c\ \{\, q' \,\} \qquad p \subseteq p' \qquad q' \subseteq q \qquad r \subseteq r' \qquad g' \subseteq g}{r, g \vdash \{\, p \,\}\ c\ \{\, q \,\}}\ \text{Weaken}$$

where

$$\mathbf{cancellative} : (\alpha : \mathsf{perm\text{-}alg}) \Rightarrow \mathsf{bool}$$

$$\mathbf{cancellative}\ z := \forall x\ y.\ x \mathbin{\#} z \wedge y \mathbin{\#} z \wedge x + z = y + z \longrightarrow x = y.$$

Figure 4 The GenRGSep Logic

4 Soundness

To prove soundness, we must extend the individual small-step rules above to a semantics of the execution of the entire program. We apply Vafeiadis' operational soundness approach [37], where the program execution not only integrates the transitive closure of small steps, but requires that each small step be closed under framing by a local state. We generalise this approach to permission algebras, which means that we do not assume either the presence of units or the cancellativity law (Figure 1).

4.1 Safety

The inductive judgement 'safe' establishes that a program c can: take n steps from the state $(h_\mathrm{l}, h_\mathrm{s})$, where h_l is the local state and h_s is the shared state; under interference from rely relation r; while ensuring the guarantee g for each Upd step; and that the state satisfies the postcondition q if c has terminated. (Note also that rely steps are counted as steps.) The formal definition of safe is as follows:

▶ **Definition 1** (Safety)**.**
Inductively:

1. 0*:* safe 0 $c\ h_\mathrm{l}\ h_\mathrm{s}\ r\ g\ q$ *always holds;*
2. Suc n*:* safe (Suc n) $c\ h_\mathrm{l}\ h_\mathrm{s}\ r\ g\ q$ *holds if*
   a. *Post-condition Safety:*
      $c = \mathbf{skip} \longrightarrow q\ (h_\mathrm{l}, h_\mathrm{s})$,
   b. *Rely Safety:*
      $\forall h'_\mathrm{s}.\ r\ h_\mathrm{s}\ h'_\mathrm{s} \longrightarrow \mathrm{safe}\ n\ c\ h_\mathrm{l}\ h'_\mathrm{s}\ r\ g\ q$,
   c. *Guarantee Safety:*
      $\forall \alpha\ h_\mathrm{lx}\ h'_\mathrm{lx}\ h'_\mathrm{s}\ c'.\ \alpha \neq \tau \wedge h_\mathrm{l} \preceq h_\mathrm{lx} \wedge ((h_\mathrm{lx}, h_\mathrm{s}), c) \sim\!\alpha\!\rightsquigarrow ((h'_\mathrm{lx}, h'_\mathrm{s}), c') \longrightarrow g\ h_\mathrm{s}\ h'_\mathrm{s}$
   d. *Opstep Safety*
      $\forall \alpha\ h'_\mathrm{l}\ h'_\mathrm{s}\ c'.\ ((h_\mathrm{l}, h_\mathrm{s}), c) \sim\!\alpha\!\rightsquigarrow ((h'_\mathrm{l}, h'_\mathrm{s}), c') \longrightarrow \mathrm{safe}\ n\ c\ h'_\mathrm{l}\ h'_\mathrm{s}\ r\ g\ q$*, and*
   e. *Frame Safety*
      $\forall \alpha\ h'\ c'\ h_\mathrm{lf}.\ ((h_\mathrm{l} + h_\mathrm{lf}, h_\mathrm{s}), c) \sim\!\alpha\!\rightsquigarrow (h', c') \longrightarrow$
      $(\exists h'_\mathrm{l}.\ h'_\mathrm{l}\ \#\ h_\mathrm{lf} \wedge h' = h'_\mathrm{l} + h_\mathrm{lf} \wedge (\alpha = \tau \longrightarrow h'_\mathrm{l} = h_\mathrm{l}) \wedge \mathrm{safe}\ n\ c\ h'_\mathrm{l}\ h'_\mathrm{s}\ r\ g\ q)$.

The function of each clause is as follows: taking zero steps is always safe, and when a step is taken; if execution has terminated ($c = \mathbf{skip}$) the postcondition is established, taking a rely step is safe, taking a local step under any expanded state ensures the guarantee, taking a local step is safe, and finally taking a local step under a frame is also safe and a framed local tau step steps to the same (unframed) local state.

We make a number of changes to Vafeiadis' original definition. By adding actions, we can distinguish between τ actions, that do not induce a guarantee step, and observable actions, that do. This also means that g is not forced to be reflexive by internal actions. Moreover, it allows us to combine non-cancellative models with operations such as external non-determinism, which have τ actions that do not collapse part of the program. (Compare sequencing and internal non-determinism, which destroy their connectives upon the τ move. See Paragraph 4.2.1.1 for more discussion of this.)

As we only have a single atomic statement, we do not need abort conditions to prevent multiple acquisitions of the same lock. If multiple locks are desired, these can be added either by the extension of the proof, as Vafeiadis does, or by instantiation with the appropriate resource model.

4.2 Soundness

For each language construct, a theorem is proven that the safety of the sub-commands shows the safety of the overall command. In addition, it is shown that framing by $*\wedge$ and weakening the precondition preserves safety. This then allows us to show the soundness of the RGSep proof system.

▶ **Theorem 2** (Soundness).

$$r, g \vdash \{\, p \,\}\ c\ \{\, q \,\} \longrightarrow p\ (h_\mathrm{l}, h_\mathrm{s}) \longrightarrow \mathrm{safe}\ n\ c\ h_\mathrm{l}\ h_\mathrm{s}\ r\ g\ q$$

Proof Sketch. The proof is by induction over the RGSep rules [**?**]. Each safe-preservation rule discussed above corresponds to an RGSep proof rule, and proves it essentially directly, with occasional weakening of the postcondition. ◀

4.2.1 Proving Operational Soundness Without Cancellativity

Perhaps surprisingly, Vafeiadis' approach to soundness *almost* generalises to non-cancellative models without any amendment. That is, the respective safety preservation rule for each command can be proven without issue, except for external non-determinism and the conjunction rule. The reason for this is that, while the frame safety condition appears to require that we cancel a non-cancellative resource, it does not actually make the true claim of cancellativity: that the resources are *equal*. It only requires that we can safely *continue* from some unframed resource.

4.2.1.1 External Non-determinism

One place where the original proof breaks is in the τ-substep rules for external non-determinism (Figure 3), $\mathrm{ENDetTau_L}$ and $\mathrm{ENDetTau_R}$. Here, we do find that, using the original definition of safe, which does not distinguish between actions, we need to appeal to cancellativity. External non-determinism, uniquely, has a rule which executes a τ-step, but keeps the primary operation ($\Box$) over that executed sub-command after execution. This creates issues with the inductive proof of safety, as τ-steps always produce *equal* heaps, but Vafeiadis' original frame safety condition only required that we find *some* smaller heap. Thus, in the soundness proof of $\Box$, in, for example, the left-step case, we would have that $\mathrm{safe}\ n\ h_\mathrm{l}\ h_\mathrm{s}\ r\ g\ q$ and

$$((h_\mathrm{l} + h_\mathrm{lf}, h_\mathrm{s}), c_1) \sim\!\tau\!\rightsquigarrow ((h'_\mathrm{l} + h_\mathrm{lf}, h_\mathrm{s}), c'_1),$$

(from the inductive frame safety hypothesis), but be required to prove $\mathrm{safe}\ n\ h'_\mathrm{l}\ h_\mathrm{s}\ r\ g\ q$. This problem is resolved by strengthening the existential heap condition in frame safety, to require that $h'_\mathrm{l} = h_\mathrm{l}$ in the case of a τ move.

4.2.1.2 Cancel and The Conjunction Rule

A more fundamental appeal to cancellativity appears in the safety proof of the conjunction rule. When proving the frame safety condition, as there are *two* safe assumptions, we obtain, by reduction of the hypotheses, two safe assumptions

$$\mathrm{safe}\ n\ c'\ h'_\mathrm{l}\ h'_\mathrm{s}\ r\ g\ q_1 \wedge \mathrm{safe}\ n\ c'\ h''_\mathrm{l}\ h'_\mathrm{s}\ r\ g\ q_2$$

and the relation

$$h_\mathrm{l} + h_\mathrm{lf} = h'_\mathrm{l} + h_\mathrm{lf},$$

but are required to find a single h_l^* such that

$$h_l^* + h_{lf} = h_l' + h_{lf} \wedge \text{safe } n\ c'\ h_l^*\ h_s'\ r\ g\ (q_1 \wedge q_2).$$

There is no way to satisfy the inductive step, because the two safe assumptions disagree on their local states, but the inductive step requires them to be equal.

This is another appearance of the well-studied *precision* side-condition for the conjunction rule [17], as cancellativity is an instance of the precision law:

$$((=)\ a \mathbin{*\!\wedge} (=)\ c) \wedge ((=)\ a \mathbin{*\!\wedge} (=)\ c) \longrightarrow ((=)\ a \wedge (=)\ b) \mathbin{*\!\wedge} (=)\ c.$$

Thus we make the pessimistic assumption that, when applying conj, every possible local state is cancellative.

4.2.1.3 Atomic

Lastly, care must be taken with atomic, as the natural framing condition to apply to the relation is the frame property [41],

$$p\ (x, z) \wedge x \#\ f \wedge b\ (x + f, z)\ xfz' \longrightarrow$$
$$\exists x'\ z'.\ x' \#\ f \wedge xfz' = (x' + f, z') \wedge b\ (x, z)\ (x', z') \wedge q\ (x', z'),$$

but this is stronger than necessary to prove safety, and rules out useful atomic commands.

We only require that

$$p\ (x, z) \wedge x \#\ f \wedge b\ (x + f, z)\ xfz' \longrightarrow \exists x'\ z'.\ x' \#\ f \wedge xfz' = (x' + f, y') \wedge q\ (x', z'),$$

which does not require that b also admits the unframed step. Note that this condition can be written more neatly as $\forall f.\ \mathbf{sp}\ b\ (p \mathbin{*\!\wedge} f) \subseteq (q \mathbin{*\!\wedge} f)$.

5 Related Work

5.1 Resource Algebras

The resource algebra approach to building separation logic was introduced by Calcagno et. al. [9], although similar ideas had been applied much earlier to relevant logic by Routley and Meyer [32, 5]. There are two main styles to formalising these algebras either represent the partial plus operation with a ternary relation or have a total plus operation and a binary disjointness relation that marks when the monoid/semigroup laws actually hold. Iris [25] takes yet another approach, and has a total plus operation and total laws, but has a validity predicate which marks when the *output* of plus is not a meaningful resource.

Calcagno et. al. introduce both separation algebras and permission algebras, but assume only a single unit (for separation algebras) and the cancellativity property (for both). Separation algebras were revisited by Dockins et. al. [13], who formalised them in ternary style in Coq [34], noted that the algebraic structure could be weakened to include multiple units, and distilled many useful laws that extend the basic resource algebra laws. Klein et. al. [26] implemented separation algebra and separation logic as an Isabelle/HOL type-class, in disjoint-plus style, which pairs well with Isabelle/HOL's simplifier. Appel et. al. [3] constructed a permission–separation algebra type-class hierarchy in ternary style in Coq for VST. This implementation weakens the positivity axiom from Dockins et. al. to account for the lack of the cancellativity law. Krebbers [28] formalised separation algebras in disjoint-plus

style in Coq, and built a C memory model on top of them. Lastly, Iris [25] develops a very powerful concurrent separation logic in Coq, based on a generalisation of resource algebras called a Camera, that allow for the approximation of impredicative invariants using step-indexing.

5.2 RGSep

Vafeiadis' original soundness proof for RGSep was proven using cancellative separation algebra, by a pen-and-paper proof [**?**]. Vafeiadis later proved the soundness of RGSep for the heap model, using a much simpler proof method [37]; this proof was mechanically formalised in Coq and Isabelle/HOL.

5.3 Explicit Stabilisation

Explicit stabilisation, or, the connectives **wssa** and **sswa**, were originally defined by Vafeiadis [**?**] to analyse where stabilisation needed to occur in an RGSep proof. However, they were defined impredicatively. Wickerson et. al. [40, 39] noted that they could be defined predicatively: respectively, as the weakest precondition and strongest postcondition of the transitive closure of the destabilising relation (e.g. the rely). They applied them to rely–guarantee, RGSep, and GSep, a proof system for reasoning about sequential programs with modules. They were applied to the verification of barriers by Dodds et. al. [15], where they were noted to improve the ease of reasoning about stability, because they could be distributed through the separating conjunction.

5.4 Separation Logic Frameworks

There are many frameworks for the verification of programs using separation logic. RGSep was integrated into the automated verification tool SmallfootRG [10]. It employs symbolic execution to automatically prove the correctness of program assertion. It is specific to the abstract heap model. SmallfootRG was formalised [36] in the HOL4 theorem prover [33], again for the heap model. The Verified Software Toolchain (VST) [2] is a toolchain and framework for the verification of C code. Its foundations are built on permission algebras in Coq. Iris [25] is a particularly powerful concurrent separation logic framework, that provides an algebraic model of ghost state for the verification of concurrent code and protocols. However, the Iris logic cannot simply embedded into Isabelle/HOL, as the later modality is incompatible with the law of excluded middle, and thus incompatible with standard Isabelle/HOL predicates.

In Isabelle/HOL, Dodds et. al. [14] implement deny–guarantee, a close relative of RGSep; they use a separation algebra approach, but assume a singular unit and cancellativity. Separation Algebras have been formalised by Klein et. al. [26, 27], but they assume a single unit, which prevents them from developing permissions separately, and also prevents the development of the multi-sep-alg instance for `discr` and sums. Lammich and Meis [30] develop imperative separation logic specifically for heaps. Lammich [29] develops a Concurrent Separation Logic in Isabelle/HOL based on Klein et. al.'s Isabelle/HOL library, which, as noted earlier assumes a single unit. Lastly, Eilers et. al. [16, 11] develop a Relational Information Flow Concurrent Separation Logic, which is specific to a combination of a fractional heap, guard state, and guard condition heap.

6 Conclusion and Future Work

In this paper, we have introduced a new Isabelle/HOL library for the development of RGSep logics. It provides a foundation for future verification of concurrent code in in Isabelle/HOL.

In the future, we would like to generalise the semantics of safe to a proper failure trace semantics, where update actions record the state update that occurs. We believe Vafeiadis' soundness method [37] should generalise quite nicely to this, as it resembles the method of Aczel traces [1], except that extra traces are added to allow for intermittent framing.

Moreover, we would like to replace **do-od** with μ-recursion, as it appears in later CSP languages [21]. This would allow for a simple implementation of general recursion, and remove the notion of *enabled* from our semantics. This is frustrated by the fact that the standard Hoare rule for recursion [19, 22] requires non-well-founded induction on the triple. This could be solved by adding concurrent specification statements [18] to our language.

References

1 P Aczel. On an inference rule for parallel composition. Private communication to Cliff Jones, February 1983. URL: https://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf.

2 Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

3 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. Chapter 6 - Separation algebras. In *Program Logics for Certified Compilers*. Cambridge University Press, 1 edition, April 2014. URL: https://www.cambridge.org/core/product/identifier/9781107256552/type/book, doi:10.1017/CBO9781107256552.

4 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, April 2014. doi:10.1017/CBO9781107256552.

5 Katalin Bimbó, Jon Michael Dunn, and Nicholas Ferenz. Two manuscripts, one by Routley, one by Meyer: The origins of the Routley–Meyer semantics for relevance logics. *The Australasian Journal of Logic*, 15(2), 2018.

6 Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1040305.1040327.

7 John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

8 Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, aug 2016. doi:10.1145/2984450.2984457.

9 Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, 2007. doi:10.1109/LICS.2007.30.

10 Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 233–248, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

11 Thibault Dardinier. Formalization of commcsl: A relational concurrent separation logic for proving information flow security in concurrent programs. *Archive of Formal Proofs*, March 2023. https://isa-afp.org/entries/CommCSL.html, Formal proof development.

12 Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag, Berlin, Heidelberg, 1990. doi:10.1007/978-1-4612-3228-5.

13 Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 161–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

14 Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. Technical Report UCAM-CL-TR-736, University of Cambridge, Computer Laboratory, January 2009. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-736.pdf`, `doi:10.48456/tr-736`.

15 Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2), jan 2016. `doi:10.1145/2818638`.

16 Marco Eilers, Thibault Dardinier, and Peter Müller. CommCSL: Proving information flow security for concurrent programs using abstract commutativity. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. `doi:10.1145/3591289`.

17 Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electronic Notes in Theoretical Computer Science*, 276:171–190, September 2011. URL: `https://linkinghub.elsevier.com/retrieve/pii/S1571066111001125`, `doi:10.1016/j.entcs.2011.09.021`.

18 Ian J. Hayes. Generalised rely-guarantee concurrency: an algebraic foundation. *Formal Aspects Computing*, 28(6):1057–1078, 2016. URL: `https://doi.org/10.1007/s00165-016-0384-0`, `doi:10.1007/S00165-016-0384-0`.

19 C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, pages 102–116, Berlin, Heidelberg, 1971. Springer Berlin Heidelberg.

20 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, aug 1978. `doi:10.1145/359576.359585`.

21 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. URL: `http://www.usingcsp.com/cspbook.pdf`.

22 C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In *Essays in Computing Science*, chapter 6. Prentice-Hall, Inc., USA, 1989. URL: `https://dl.acm.org/doi/10.5555/63445.C1104361`.

23 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983. `doi:10.1145/69575.69577`.

24 Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

25 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. `doi:10.1017/S0956796818000151`.

26 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 332–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

27 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. `https://isa-afp.org/entries/Separation_Algebra.html`, Formal proof development.

28 Robbert Krebbers. Separation algebras for C verification in Coq. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, pages 150–166, Cham, 2014. Springer International Publishing.

29 Peter Lammich. Refinement of parallel algorithms down to LLVM. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL:

https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2022.24, doi:10.4230/LIPIcs.ITP.2022.24.

30 Peter Lammich and Rene Meis. A separation logic framework for imperative hol. *Archive of Formal Proofs*, November 2012. https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.

31 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007. Festschrift for John C. Reynolds's 70th birthday. URL: https://www.sciencedirect.com/science/article/pii/S030439750600925X, doi:10.1016/j.tcs.2006.12.035.

32 Richard Routley and Robert K. Meyer. The semantics of entailment. In Hugues Leblanc, editor, *Truth, Syntax and Modality*, volume 68 of *Studies in Logic and the Foundations of Mathematics*, pages 199–243. Elsevier, 1973. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08715416, doi:10.1016/S0049-237X(08)71541-6.

33 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

34 The Coq Development Team. The Coq reference manual – release 8.19.1. https://coq.inria.fr/doc/V8.19.1/refman, 2024.

35 Lawrence C. Paulson Tobias Nipkow, Markus Wenzel, editor. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Springer Berlin, Heidelberg, 2002. doi:https://doi.org/10.1007/3-540-45949-9.

36 Thomas Tuerk. A formalisation of smallfoot in hol. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 469–484, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

37 Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII). URL: https://www.sciencedirect.com/science/article/pii/S1571066111001204, doi:10.1016/j.entcs.2011.09.029.

38 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

39 John Wickerson. Concurrent verification for sequential programs. Technical Report UCAM-CL-TR-834, University of Cambridge, Computer Laboratory, May 2013. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-834.pdf, doi:10.48456/tr-834.

40 John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 610–629, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

41 Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 402–416, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.