Techniques for model construction in separation logic

Jonas B. Jensen

March 2014

Contents

1 Introduction

Separation logic has been very successful at giving concise specifications and short proofs to pointer-manipulating programs. Unfortunately, the term *separation logic* covers a whole zoo of different theories. Almost every publication that applies separation logic starts by defining a new logic that is general enough to attack the problem at hand but not so general that it disturbs the presentation with orthogonal features.

This proliferation of theories has been identified as a problem several times [BBTS05, Par10], but that has not stopped the flow of new logics being created. Given that no logic proposed so far is a generalisation of all others, we should welcome more exploration and diversity for the foreseeable future.

For new researchers in the field, the best sources for building an intuitive as well as formal understanding of separation logic are still the original articles from 2001-2002 [Rey02, IO01]. Unfortunately, this leads many researchers to ignore some of the recent advances that lead to more general theories with simpler and more concise presentation and metatheory.

This chapter attempts to summarise those advances in the hope that it will allow new research on separation logic to start from the cutting edge as of 2013 instead of 2002. The reader is assumed to have some familiarity with separation logic. A good place to start would be the 2002 introduction by Reynolds [Rey02] or the 2012 tutorial by O'Hearn [O'H12]. More in-depth theoretical discussions are found in [IO01] and [Rey11], although these are still introductory texts.

The level of formal detail in this chapter will be quite high since it aims to be useful for readers trying to encode separation logic in a proof assistant or building a stand-alone tool.

1.1 Overview

The general recipe for making a separation logic, and the structure of this chapter, is as follows.

1. Choose a **programming language**. Small variations in the definition and semantics of the language can have great consequences for the ease of building a separation logic for it.

2. Design the **assertion logic**; i.e., the formulas that describe machine state. It typically includes separating conjunction and points-to formulas. Assertion-logic formulas are typically modelled as sets of machine state subject to some instrumentation and/or side conditions.

3. Design the **specification logic**; i.e., the formulas that describe computations. It typically includes Hoare triples, which refer to assertions

in their pre- and postconditions. In some higher-order settings, the assertion logic may also refer to specifications, in which case the definitions of these two logics may have to be mutually recursive or may coincide.

There is also an optional 0'th step: choose a **metalogic**; i.e., a mathematical framework in which to formulate the theory. The metalogic is usually implicitly chosen as "standard math as taught in school", but recent examples have shown that some separation logics become both simpler and more general when embedded in a metalogic that provides, for example, bound names [Pit01], dependent types [KBJD13] or step-indexing [SB13a, BMSS12].

1.2 What is a separation logic?

To limit the scope of this text, we impose some minimum requirements on what is considered a separation logic. We will say it must contain:

1. An logic of **assertions** P, Q, R that is a complete BI algebra; i.e., it satisfies the rules in Figure 2 (page 9) and Figure 4 (page 13).

2. A logic of **specifications** S that is a complete Heyting algebra; i.e., it satisfies the rules in Figure 2.

3. A **Hoare-triple** specification of the form $\{P\}\ c\ \{Q\}$ or a generalisation thereof. The Hoare triple must satisfy

$$\frac{P \vdash P' \quad S \vdash \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{S \vdash \{P\}\ c\ \{Q\}}\ \textsc{Consequence}$$

$$\frac{S \vdash \forall x.\ \{P(x)\}\ c\ \{Q\}}{S \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists} \qquad \frac{S \vdash \{P\}\ c\ \{Q\}}{S \vdash \{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

These requirements are somewhat imprecise, but they have to be, since they apply to logics that have yet to be invented.

The inference rules required of the triple ensure that it admits the *narration* style of writing a Hoare-logic proof [Rey11, WDP13], where commands are interleaved with assertions. Figure 1 shows an example proof narration that uses all three rules.

1.3 Contributions

The main contribution of this chapter is to survey a portion of the first twelve years of separation-logic literature in a common mathematical framework. There is too much published literature to mention it all, so the focus is on techniques that are needed everywhere rather than the very latest developments in specific areas such as concurrency.

```
$\{\mathsf{head} \neq \mathit{nil} \wedge \mathit{list}(\mathsf{head})\}$                  (Consequence)
$\{\exists n.\ \mathsf{head} \mapsto n * \mathit{list}(n)\}$                          (Exists)
$\{\mathsf{head} \mapsto n * \mathit{list}(n)\}$                                      (Frame)
  $\{\mathsf{head} \mapsto n\}$
    **next := [head];**
  $\{\mathsf{head} \mapsto n \wedge \mathsf{next} = n\}$
$\{(\mathsf{next} = n \wedge \mathsf{head} \mapsto n) * \mathit{list}(n)\}$            (Consequence)
$\{\mathsf{head} \mapsto \mathsf{next} * \mathit{list}(\mathsf{next})\}$                (Frame)
  $\{\mathsf{head} \mapsto \mathsf{next}\}$
    **free(head);**
  $\{\mathit{emp}\}$
$\{\mathit{emp} * \mathit{list}(\mathsf{next})\}$                                      (Consequence)
$\{\mathit{list}(\mathsf{next})\}$
```

Figure 1: Narration-style proof of a two-line example program.

The chapter follows the same structure as most other articles that define a programming language and build a separation logic for it, but along the way we discuss alternatives and variations at every point. A particularly thorough treatment is given to *separation algebras*, since their theory is highly scattered across the literature and often stated in a much less general form than it could be.

Despite its length, this text is far from being self-contained. The reader is encouraged to follow literature references for more details and discussions and for fully worked-out examples.

The conclusion is that making a full-featured higher-order separation logic is not difficult. With the well-known but underused techniques of *shallow embedding* and *separation algebras*, satisfying the axioms of Figures 2 and 4 can be done by satisfying another set of axioms that is simpler and relates more directly to the memory model of a programming language.

2 Semantics of the programming language

A separation logic is typically formulated in the context of one particular programming language, and it is left as an exercise to the reader to port it to other languages. This is rarely a hurdle, but it certainly makes comparisons more difficult. In those articles that abstract away from the details of the programming language [COY07, BJSB11, DYBG$^+$13], the added generality is typically considered a central contribution.

The choice of programming language and semantics forms part of the statement of the soundness theorem, so it should be as uncontroversial as possible. It is often so uncontroversial that it is not even written out in full. There are a few choices to make, though, beginning with the style of semantics:

Operational. Some flavour of operational semantics is typically the preferred choice. A typical **big-step** semantics is an inductively-defined predicate of the form $\sigma, c \rightsquigarrow \sigma'$ that holds when command c from state σ may terminate successfully in state σ'. A typical **small-step** semantics is an inductively-defined predicate of the form $\sigma, c \rightsquigarrow \sigma', c'$ that holds when command c may take a single step from state σ, leaving a residual command c' and a modified state σ'. Both types of operational semantics often have an additional form $\sigma, c \rightsquigarrow \mathsf{fail}$ that holds when c from state σ may fail, either eventually (for big-step semantics) or in one step (for small-step semantics).

The word “may” above refers to non-determinism of the semantics, which typically arises from memory allocation and concurrency.

Denotational. It is possible to build a separation logic over a denotational semantics, but this is quite rarely seen. Examples include the work of Varming and Birkedal [VB08], Brookes's soundness proof of concurrent separation logic [Bro07] and Hoare Type Theory [PBNM08].

Axiomatic. An axiomatic semantics can be thought of as a program logic without a soundness proof, which sounds like a bad thing. But if this underlying program logic has already been proved sound elsewhere, then an axiomatic semantics can be an effective shortcut to a simpler metatheory. It also demonstrates that the underlying program logic is sufficiently expressive when a high-level program logic can be layered on top of it.

Examples of axiomatic semantics for separation logics are rare, but we found it very useful in [JB12], where the soundness of *fictional* separation logic is proved relative to the soundness of *standard* separation logic. Another example is [DYGW10], where a high-level program and its specification are translated to a low-level program and a low-level specification.

The rest of this section will discuss *operational* semantics only.

Authors of separation logics are often guilty of defining the programming-language semantics in ways that are perhaps unnatural for the language but make it easier to prove the separation-logic metatheory. It is then understood, either formally or informally, that this **instrumented semantics** is **adequate** with respect to a semantics that would be more natural for the language; i.e., any specification proved to hold in the instrumented semantics also holds in the original semantics.

For instance, it is typical to instrument the semantics so it is well-defined how commands behave in partial states – i.e., states where any location might be missing. This might be done even when modelling a machine where all memory is present all the time [JBK13, Myr10], or where the type system of the language would ordinarily guarantee that there are no dangling pointers [Par05, BJSB11]. Commands executed from a too small partial state should then fail. When this is defined just right, we can prove:

Safety monotonicity. If command c cannot fail in state σ, then it cannot fail in any extension of σ either.

Frame property. In a big-step semantics, the frame property holds if whenever a configuration (σ_0, c) cannot fail and $(\sigma_0 \cdot \sigma_1), c \rightsquigarrow \sigma'$ then there exists σ_0' such that $\sigma_0, c \rightsquigarrow \sigma_0'$ and $\sigma' = \sigma_0' \cdot \sigma_1$. Here, $\cdot$ is composition of disjoint states – see Section 3.3.

When these two properties hold, the frame rule follows easily [YO02].

The properties above surprisingly sensitive to language features. Safety monotonicity will fail if there is a command to query whether some memory location is mapped [YO02]. The frame property will even fail if the memory allocator is deterministic [YO02]. It is possible, though, to have the frame rule without having the frame property – see Section 4.1.1.

2.1 Modelling failure

Failure – i.e., the program crashing – is traditionally a crucial part of separation logic. Hoare triples $\{P\}\ c\ \{Q\}$ are always interpreted to imply that (σ, c) cannot fail when σ satisfies P; thus, there must be a possibility of failure in the semantics, or this property of triples would hold vacuously. As mentioned above, failure can be modelled with a transition $\sigma, c \rightsquigarrow \mathsf{fail}$ in the operational semantics, and it typically happens when dereferencing an invalid pointer.

One must consider whether the semantics permits the distinction between three important program behaviours: **successful termination**, **failure** and **non-termination**. One or both of the latter two behaviours might correspond to the semantics getting **stuck**; i.e., not allowing further reductions. We say that a configuration (σ, c) is stuck[1] when there is no x, not even fail, such that $\sigma, c \rightsquigarrow x$.

In a big-step semantics, stuckness should correspond to (guaranteed) non-termination, and therefore we need the fail value to model actual failure if we want to distinguish the two. But this means that a rule must be added to trigger and propagate failure in every place it might occur. In the version

[1] In a small-step semantics, we might further require $c \neq \mathsf{skip}$.

of the Charge! platform described in [BJSB11], there are about as many rules for failure as there are rules for success, and forgetting to add a failure rule renders the model of the programming language unsound. This is because it makes failure look like non-termination, and a partial-correctness program logic cannot distinguish non-termination from success.

A possible fix for this problem is to not have failure rules but instead a set of coinductive rules for when a configuration (σ, c) may diverge [LG09]. Then failure is defined as a configuration that is stuck but cannot diverge, and omission of rules leads to incompleteness instead of unsoundness.

In a small-step semantics, we can often design a system where stuckness corresponds to failure, avoiding the need for a special fail configuration. This increases confidence in the semantics and cuts the number of rules approximately in half for the reasons mentioned above. It works because stuckness of a subcommand tends to propagate to compound commands: the sequence command $\mathsf{crash}; c$ will be stuck after one step because the crash command is stuck. Unfortunately, this is sensitive to language features; in particular, it will not work in a language with a parallel operator: the command $\mathsf{crash} \mid\mid \mathsf{loop\_forever}$ is never stuck. In such a language, it is therefore necessary to add an explicit fail configuration [Vaf11], leading to the same problems as in a big-step semantics.

It is worth mentioning that the Views framework [DYBG+13] quite elegantly avoids explicit propagation of failure through control-flow commands. It is a small-step semantics, but instead of a failure *configuration* (replacing (σ, c)), there is a failure *state* (replacing σ). See [SB13b] for a generalisation of the approach that also works for procedure calls and fork-parallelism.

2.2 Modelling concurrency

Separation logic for concurrent languages is an important and active area of research, and there is certainly room for further exploration. Important choices to be made in the semantics include the following.

- The basic concurrency primitive can be either a **parallel operator** $(c_1 \mid\mid c_2)$ or a **fork command** ($\mathsf{fork}\ c$ or $\mathsf{fork}\ f$ for a function name f). The fork command is more similar to real-world programming languages, but the parallel operator is sometimes easier to model. This is because it is often less expressive – in particular, it is often impossible to write a program that executes n commands concurrently, where n is only determined at run time. This becomes possible to do in continuation-passing style if the language has recursive procedures.

  In a language without procedures and where concurrency comes from the parallel operator, it is possible to syntactically see which threads are active at which point in the program. This can be used to simplify proofs [GBC11], but that technique does not scale directly to realistic languages.

- Communication between threads must also be built into the language at some level. Common solutions include static **locks**, dynamically-allocated locks, a **compare-and-swap** command, and an **atomic** command modifier. Some of these primitives can be derived from each other in a more or less practical way.

- Local (stack) variables can be shared between threads or not. Even though real-world programming languages rarely share mutable local variables between threads, this is often allowed in the semantics of toy languages, and then races must be ruled out at the logic level [O'H07].

  Sharing of mutable local variables becomes more complicated if concurrency comes from a fork command or if they might also be captured in lambda-expressions [SBP10].

- Most separation logics published so far have been for toy languages with **sequentially-consistent** memory, meaning that all threads agree on the value of shared memory at all times. Actual programming languages and multi-core hardware have weaker memory models, and **weak-memory** separation logics have only recently started to emerge [FFS10, WB11, VN13].

3 Assertion logic

Assertions are the formulas P, Q occurring in the pre- and postconditions of Hoare triples $\{P\}\ c\ \{Q\}$. They are essentially predicates on machine state, and the distinguishing feature of separation logic is that they can contain *separating conjunction* and related operators. In this section, we will develop the theory necessary to make all this formal.

3.1 Heyting algebras

The assertion logic must first of all be a *logic.* I choose to define a logic as a **complete Heyting algebra**:

Definition 3.1. A **complete Heyting algebra** is a type equipped with operators $(\top, \bot, \wedge, \vee, \forall, \exists, \Rightarrow)$ and a binary **entailment** relation $\vdash$, satisfying the axioms of Figure 2. Read the horizontal lines in the figure as implication in the metalogic. $\diamond$

$$\frac{}{P \vdash P}\ \vdash\text{-}\textsc{Refl} \qquad \frac{P \vdash Q \quad Q \vdash R}{P \vdash R}\ \vdash\text{-}\textsc{Trans} \qquad \frac{P \vdash Q \quad Q \vdash P}{P = Q}\ \vdash\text{-}\textsc{AnSy}$$

$$\frac{}{P \vdash \top}\ \top\text{-R} \qquad \frac{}{\bot \vdash P}\ \bot\text{-L}$$

$$\frac{P \vdash Q_1 \quad P \vdash Q_2}{P \vdash Q_1 \wedge Q_2}\ \wedge\text{-R} \qquad \frac{P_1 \vdash Q}{P_1 \wedge P_2 \vdash Q}\ \wedge\text{-L1} \qquad \frac{P_2 \vdash Q}{P_1 \wedge P_2 \vdash Q}\ \wedge\text{-L2}$$

$$\frac{P_1 \vdash Q \quad P_2 \vdash Q}{P_1 \vee P_2 \vdash Q}\ \vee\text{-L} \qquad \frac{P \vdash Q_1}{P \vdash Q_1 \vee Q_2}\ \vee\text{-R1} \qquad \frac{P \vdash Q_2}{P \vdash Q_1 \vee Q_2}\ \vee\text{-R2}$$

$$\frac{P \vdash Q \Rightarrow R}{P \wedge Q \vdash R}\ \wedge\text{-}\textsc{Adjoint} \qquad \frac{P \wedge Q \vdash R}{P \vdash Q \Rightarrow R}\ \Rightarrow\text{-}\textsc{Adjoint}$$

$$\frac{\forall x : T.\ (P \vdash Q(x))}{P \vdash \forall x : T.\ Q(x)}\ \forall\text{-R} \qquad \frac{P(t) \vdash Q}{\forall x : T.\ P(x) \vdash Q}\ \forall\text{-L}$$

$$\frac{\forall x : T.\ (P(x) \vdash Q)}{\exists x : T.\ P(x) \vdash Q}\ \exists\text{-L} \qquad \frac{P \vdash Q(t)}{P \vdash \exists x : T.\ Q(x)}\ \exists\text{-R}$$

Figure 2: Axioms of a complete Heyting algebra.

For convenience, we define the following abbreviations.

$$\begin{aligned} \vdash P &\triangleq \top \vdash P && \text{pronounced “}P\text{ is } \textbf{valid}\text{”} \\ P \equiv Q &\triangleq P = Q && \text{but with low precedence, like } \vdash \\ \neg P &\triangleq P \Rightarrow \bot \end{aligned}$$

The axioms in Figure 2 are one of many equivalent presentations. Like a sequent calculus, most rules are presented as left-rules and right-rules for each operator. However, it is not a standard sequent calculus. Notice the following details.

- There is a single hypothesis on the left of the turnstile rather than a comma-separated list of hypotheses. This is the norm in separation logic because it is otherwise not clear whether the comma would denote ordinary conjunction or separating conjunction. For an alternative that is more suitable for proof theory, see [OP99].
- The rules for implication ($\wedge$-Adjoint and $\Rightarrow$-Adjoint) do not follow the pattern of left-rules and right-rules. They are instead presented as an **adjunction**, or **Galois connection**: for any P, the functor $- \wedge P$ is the left adjoint of $P \Rightarrow -$. This presentation simplifies the proof system when there is only one hypothesis on the left of the

turnstile. It also highlights the fact, coming from the general theory of adjunctions, that the implication operator is uniquely determined by the conjunction operator and vice versa.

- The rules in Figure 2 are written out quite verbosely such that they look like a logic and can be practically applied as such. A more common definition of complete Heyting algebra would characterise entailment $\vdash$ as a partial order with least upper bounds ($\bot, \vee, \exists$), greatest lower bounds ($\top, \wedge, \forall$) and an exponential ($\Rightarrow$).

Most logics in the separation-logic literature are less general than a complete Heyting algebra, either because they are missing some operators or because the domain of quantification is restricted. However, there is rarely a reason for this lack of generality, other than a perceived gain in simplicity. We will see in Section 3.3 how to define an assertion logic such that it is a complete Heyting algebra by construction.

3.1.1 Shallow embedding

Definition 3.1 does not go through the traditional indirection of defining a syntax for formulas and a denotation function from syntactic formulas to semantic assertions. We have only the semantic assertions, and operators such as $\wedge$ are merely infix functions on those.

This approach is sometimes known in the literature as a **shallow embedding** [WN04], "working directly in the semantics" or "the extensional approach" [Nip02]. It is used by the majority of separation-logic formalisations inside proof assistants [AB07, TKN07, CSV07, VB08, McC09, Myr10, BJSB11, JBK13, AM13] since it eliminates a lot of tedious work – the kind of work that tends to be dismissed as "routine" in informal mathematics but cannot be ignored when every detail has to be machine-checked. In particular, a shallow embedding eliminates the need for contexts of **logical variables** and their types, accounts of free logical variables, or capture-avoiding substitutions and notions of fresh names. It does not save us from proving tedious results about **program variables**, though; see Section 3.4.

The quantifiers in Figure 2 are annotated with a domain T, which we sometimes omit when it is clear from the context. Because we have a shallow embedding, T ranges over the types of the metalogic, and the formula under the quantifier is a metalogic function from T to assertions. Notice that the universal quantifiers in the premise of rules $\forall$-R and $\exists$-L belong to the metalogic rather than the complete Heyting algebra. Notice also that it is possible for T itself to be the type of assertions, which means that we have defined a higher-order logic.

The opposite of shallow embedding is a **deep embedding**, where formulas and inference rules are syntactic objects that can be manipulated independently of their semantic **models**, of which there can be many. A common motivation for deep embeddings is to study the rules of the logic independently from its models. But notice that we can still do this since Definition 3.1 characterises complete Heyting algebras in the abstract, separately from describing any particular such algebra. Shallow embedding is not a new way to study logic, but it is rarely used with separation logic outside proof assistants. Compare this with other mathematical fields, where it is standard, for example, to study group theory independently of particular groups, and nobody would propose to use syntactic formulas for this.

With all this said, it is justifiable to use a deep embedding when the desire is to limit expressiveness of the logic deliberately [Nip02]. This can be used for stating decidability or completeness results or when documenting a software tool that manipulates this logic and thus needs to represent it symbolically.

3.1.2 Injecting metalogic propositions

The quantifiers in a shallow embedding allow us to mention data of arbitrary type in formulas. For this to be useful, we also need to inject propositions from the metalogic that describe this data. In particular, we will need an injection $\langle p \rangle$ of metalogic propositions p into assertions.

The alternative to such an injection would be to recreate the necessary mathematical theories inside the assertion logic; i.e., equality, induction, recursion, etc. While this is certainly possible, it could end up being more work than the separation logic itself.

Fortunately, we can define a $\langle p \rangle$ for any Heyting algebra by exploiting the existential quantifier and metalogic subtyping[2]:

$$\langle p \rangle \triangleq \exists x : \{x : \mathit{unit} \mid p\}.\ \top$$

The injection is covariant with respect to entailment, and it satisfies practical left- and right-rules:

$$\frac{p \Rightarrow q}{\langle p \rangle \vdash \langle q \rangle}\ \langle\rangle\text{-}\vdash \qquad \frac{p \Rightarrow (\vdash Q)}{\langle p \rangle \vdash Q}\ \langle\rangle\text{-L} \qquad \frac{q}{P \vdash \langle q \rangle}\ \langle\rangle\text{-R}$$

We now have the theory of equality in our complete Heyting algebra for free, simply by lifting it from the metalogic. For instance, we can prove

$$P(x) \land \langle x = y \rangle \vdash P(y)$$

The corresponding entailment in a deep embedding would be written as $P \land x = y \vdash P[y/x]$. The deep-embedding concepts of free variables and substitutions are modelled here with function application.

[2] The exact definition will vary depending on the metalogic. The *unit* type can be replaced with any other non-empty type. In Coq, we can simply write $\langle p \rangle \triangleq \exists x : p.\ \top$.

$$
\begin{array}{lll}
\top \triangleq T & P \wedge Q \triangleq P \cap Q & \forall x : T.\ P(x) \triangleq \bigcap_{x:T} P(x) \\
\bot \triangleq \emptyset & P \vee Q \triangleq P \cup Q & \exists x : T.\ P(x) \triangleq \bigcup_{x:T} P(x) \\
P \vdash Q \triangleq P \subseteq Q & P \Rightarrow Q \triangleq \{t \mid \forall t' \geq t.\ t' \in P \Rightarrow t' \in Q\} &
\end{array}
$$

Figure 3: Kripke definition of a complete Heyting algebra when assertions are subsets of T, upwards closed under a preorder $\leq$.

3.1.3 Kripke models

Complete Heyting algebras are convenient because they correspond to a familiar notion of logic. They are also convenient because they are easy to construct from a type T and a **preorder** on T; i.e., a binary relation that is reflexive and transitive:

Proposition 3.1. *Given a preordered type $(T, \leq)$, the powerset $\mathcal{P}_{\leq}(T)$ is a complete Heyting algebra, where*

$$\mathcal{P}_{\leq}(T) \triangleq \{P : \mathcal{P}(T) \mid \forall t \in P.\ \forall t' \geq t.\ t' \in P\}$$

and the operators of $\mathcal{P}_{\leq}(T)$ are defined as in Figure 3.

The definitions in Figure 3 are known as a **Kripke model**; the interesting part of it is the definition of implication, which explicitly ensures closure under $\leq$, whereas that closure holds directly for all other operators. The injection from the metalogic to the assertion logic that was discussed in Section 3.1.2 can be defined as $\langle p \rangle = \{t \mid p\}$.

More generally, we can construct a Heyting algebra from an existing one as follows.

Proposition 3.2. *Given a complete Heyting algebra A and a preordered type $(T, \leq)$, the space of monotonic functions $T \to_{\leq} A$ is also a complete Heyting algebra, where*

$$T \to_{\leq} A \triangleq \{f : T \to A \mid \forall t, t'.\ t \leq t' \Rightarrow (f(t) \vdash f(t'))\}$$

and the operators of $T \to_{\leq} A$ are defined in terms of the operators on A:

$$
\begin{array}{rll}
P \oplus Q \triangleq \lambda t.\ P(t) \oplus Q(t) & & \textit{for } \oplus \in \{\wedge, \vee\} \\
\mathrm{P} \triangleq \lambda t.\ \mathrm{P} & & \textit{for } \mathrm{P} \in \{\top, \bot\} \\
\kappa x.P(x) \triangleq \lambda t.\ \kappa x.P(x)(t) & & \textit{for } \kappa \in \{\forall, \exists\} \\
P \Rightarrow Q \triangleq \lambda t.\ \forall t' \geq t.\ P(t') \Rightarrow Q(t') & & \\
P \vdash Q \triangleq \forall t.\ (P(t) \vdash Q(t)) & &
\end{array}
$$

$$\frac{}{(P * Q) * R \vdash P * (Q * R)}\ *\text{-}\textsc{Assoc} \qquad \frac{}{P * Q \vdash Q * P}\ *\text{-}\textsc{Comm}$$

$$\frac{}{P * emp \equiv P}\ *\text{-}emp \qquad \frac{P \vdash Q}{P * R \vdash Q * R}\ *\text{-}\vdash$$

$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}\ *\text{-}\textsc{Adjoint} \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}\ \mathrel{-\!\!*}\text{-}\textsc{Adjoint}$$

Figure 4: Additional axioms of a BI algebra

Proof. See [BJ], Lemma ILPre_ILogic. □

We will use these constructions to define specification logics, and we will use generalised forms of them to define assertion logics.

Note that the **law of the excluded middle**, i.e. $\vdash P \vee \neg P$ for all P, does not follow from the axioms of a complete Heyting algebra, and it is invalid in the models constructed here unless the preorder is also symmetric; i.e., an equivalence relation.

3.2 BI algebras

Defining complete Heyting algebras only got us half way to separation-logic assertions. We still need an account of the operators that make separation logic special: separating conjunction ($*$), separating implication ($\mathrel{-\!\!*}$), and *emp*. Note that *emp* is sometimes written I in the literature.

Definition 3.2. A **complete BI algebra** [Pym02] is a complete Heyting algebra with additional operators ($*, \mathrel{-\!\!*}, emp$) satisfying the axioms in Figure 4. The **precedence** of operators used in this text will be, in decreasing order:

$$= \quad * \quad \wedge \quad \vee \quad \mathrel{-\!\!*} \quad \Rightarrow \quad \forall \quad \exists \quad \vdash \quad \equiv$$ ◇

The axioms in Figure 4 are intentionally minimal. Rules for associativity and commutativity with $\equiv$ instead of $\vdash$ are derivable. It can also be derived that $*$ is covariant in both arguments; i.e.,

$$\frac{P \vdash P' \quad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

When I required in Section 1.2 that the assertion logic must be a complete BI algebra, it was not only because this gives us the rules that make separation logic intuitive to work with. It is also because it is easy to satisfy these rules. We will see in Section 3.3 how a complete BI algebra arises naturally from the memory model of a typical programming language or machine.

The operators $-\!\!*$ and $\Rightarrow$ are often omitted from presentations when they are not needed for the examples at hand. But even then, they play an important role in metatheory: they witness that $*$ and $\wedge$ commute with existential quantification as explained in the following proposition.

Proposition 3.3. *In a complete Heyting algebra with an operator $*$ satisfying rule $*$-$\vdash$, it holds that $(\exists x\colon T.\ P(x)) * Q \vdash \exists x\colon T.\ P(x) * Q$* if and only if *there is an operator $-\!\!*$ satisfying the rules $*$-*Adjoint *and* $-\!\!*$-*Adjoint.

This proposition is a direct consequence of the *adjoint functor theorem* in category theory, but it is worth seeing the proof written out for the specific case of separation logic. Notice that the left-to-right direction, which is probably the surprising one, is only provable because the assertion logic is higher order, and we can quantify over a type T that represents a set of assertions.

Proof. ($\Leftarrow$) By the following proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.\ (Q(x) * P \vdash Q(x) * P)}\ \vdash\text{-}\textsc{Refl}}{\forall x.\ (Q(x) * P \vdash \exists x.\ Q(x) * P)}\ \exists\text{-R}}{\forall x.\ (Q(x) \vdash P -\!\!* \exists x.\ Q(x) * P)}\ -\!\!*\text{-}\textsc{Adjoint}}{\exists x.\ Q(x) \vdash P -\!\!* \exists x.\ Q(x) * P}\ \exists\text{-L}}{(\exists x.\ Q(x)) * P \vdash \exists x.\ Q(x) * P}\ *\text{-}\textsc{Adjoint}$$

($\Rightarrow$) Define $Q -\!\!* R$ as $\exists P' : \{P' \mid P' * Q \vdash R\}.\ P'$. We first prove $-\!\!*$-Adjoint, which holds without appealing to the assumption we made:

$$\dfrac{\dfrac{\dfrac{}{P \vdash P}\ \vdash\text{-}\textsc{Refl} \qquad \dfrac{P * Q \vdash R}{P : \{P' \mid P' * Q \vdash R\}}\ \text{(metalogic)}}{P \vdash \exists P' : \{P' \mid P' * Q \vdash R\}.\ P'}\ \exists\text{-R with } P}{P \vdash Q -\!\!* R}\ \textsc{Def.}$$

To prove $*$-Adjoint, we appeal to our assumption under the name $*$-$\exists$.

$$\dfrac{\dfrac{P \vdash Q -\!\!* R}{P * Q \vdash (Q -\!\!* R) * Q}\ *\text{-}\vdash \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall P'.\ (P' * Q \vdash R) \Rightarrow (P' * Q \vdash R)}}{\forall P' : \{P' \mid P' * Q \vdash R\}.\ (P' * Q \vdash R)}\ \text{(metalogic)}}{\exists P' : \{P' \mid P' * Q \vdash R\}.\ P' * Q \vdash R}\ \exists\text{-L}}{(\exists P' : \{P' \mid P' * Q \vdash R\}.\ P') * Q \vdash R}\ *\text{-}\exists}{(Q -\!\!* R) * Q \vdash R}\ \textsc{Def.}}{P * Q \vdash R}\ \vdash\text{-}\textsc{Trans}$$

□

By analogy with Proposition 3.3, we can also show that the $\Rightarrow$-operator exists if and only if the $\wedge$-operator commutes with existentials.

3.2.1 Classical and intuitionistic logics

There is an important special case of BI that is relevant for separation logic: **Boolean BI** (**BBI**). BBI is obtained by adding the **law of the excluded middle** ($\vdash P \lor \neg P$ for all P) to the axioms of Figure 2, which makes Figure 2 describe a **complete Boolean algebra** – a special case of a complete Heyting algebra. Adding the axioms of Figure 4 as well, one obtains a **complete BBI algebra**.

Another important dialect is **affine BI** [GMP05]. This is obtained by adding **weakening of** $*$ to BI, meaning that $P * Q \vdash P$, or equivalently, $emp \equiv \top$. It is the preferred way to define a separation logic for a garbage-collected language, where it lets us "logically forget" the resource Q with the expectation that it will be garbage-collected some time after (or even before! [Rey00, HDV11]) logically forgetting it.

In the separation-logic literature [IO01], BBI is typically known as **classical separation logic**, and affine BI is known as **intuitionistic separation logic**. This is unlike the more established terminology from philosophical logic, where propositions that are provable in intuitionistic logic are also expected to be provable in classical logic. To add to the confusion, there also exists *classical BI* [BC10], which is something else entirely. To avoid these clashes of terminology, I will prefer the terms *Boolean* and *affine* over *classical* and *intuitionistic* in the remainder of this chapter.

A BI algebra that is both Boolean and affine collapses in the sense that $P * Q \equiv P \land Q$ [BK10]. On the other hand, as we will see in Section 3.3.4, there exist useful separation logics that are neither Boolean nor affine.

3.2.2 Design freedom

It might seem like there is a great deal of freedom when designing a logic that satisfies the axioms in Figures 2 and 4, but many operators are uniquely determined by others. In fact, all operators of a complete BI algebra are uniquely determined when $\vdash$ and $*$ are chosen.

The left- and right-rules of Figure 2 uniquely identify the operators $(\forall, \exists, \land, \lor, \top, \bot)$. For example, if another operator $\wedge\!\!\!\wedge$ satisfies the axioms for $\land$ in Figure 2, it follows that $P \wedge\!\!\!\wedge Q \equiv P \land Q$ for all P and Q.

It follows from Figure 4 that $(*, emp)$ is a monoid, so its unit emp is unique. This means that once the operator $*$ is defined, there is no freedom left to choose the operator emp.

Similarly, the proof rules for $\mathrel{-\!\!*}$ and $\Rightarrow$ essentially say that $(P \mathrel{-\!\!*} -)$ is the right adjoint of $(- * P)$ and that $(P \Rightarrow -)$ is the right adjoint of $(- \land P)$. Since adjoints are unique, there is no freedom in choosing the operators $(\mathrel{-\!\!*}, \Rightarrow)$ once $(*, \land)$ have been defined. In fact, we can define $\mathrel{-\!\!*}$ and $\Rightarrow$ in terms of other operators:

Proposition 3.4. *In a complete BI algebra asn, for all* $Q, R : asn$*,*

1. *$Q \mathrel{-\!\!*} R \equiv \exists P : \{P \mid P * Q \vdash R\}.\ P$*
2. *$Q \Rightarrow R \equiv \exists P : \{P \mid P \land Q \vdash R\}.\ P$*

Proof (of 1.). Define $Q \mathrel{-\!\!*}_\exists R \triangleq \exists P : \{P \mid P * Q \vdash R\}.\ P$. The proof of Proposition 3.3 shows that $\mathrel{-\!\!*}_\exists$ follows the same two axioms as $\mathrel{-\!\!*}$. It follows that $(P \mathrel{-\!\!*}_\exists Q \vdash P \mathrel{-\!\!*} Q)$ if $((P \mathrel{-\!\!*}_\exists Q) * P \vdash Q)$ if $(P \mathrel{-\!\!*}_\exists Q \vdash P \mathrel{-\!\!*}_\exists Q)$ if true. The converse is similar, which gives the necessary bientailment. $\square$

We can also define $(\land, \top)$ and $(\lor, \bot)$ in terms of $\forall$ and $\exists$ respectively:

Proposition 3.5.

1. *Assume $P, Q : asn$, where asn is a complete Heyting algebra. Let $f_2 : \{true, false\} \to asn$ be the function that maps true to P and false to Q. Then*

$$P \lor Q \equiv \exists b.\ f_2(b)$$
$$P \land Q \equiv \forall b.\ f_2(b)$$

2. *Let $f_0 : \emptyset \to asn$ be the unique function with that type. Then, in a complete Heyting algebra,*

$$\bot \equiv \exists x.\ f_0(x)$$
$$\top \equiv \forall x.\ f_0(x)$$

Proof. See the Coq code accompanying [JBK13], Section ILogicEquiv. $\square$

3.3 Separation algebras

3.3.1 Heaps

For a typical toy programming language, the type of heaps is defined as $heap = loc \overset{\text{fin}}{\rightharpoonup} val$, where loc is the type of heap **locations** (e.g., the natural numbers), val is the type of **values** that can be stored in the heap (e.g., integers and locations), and $\overset{\text{fin}}{\rightharpoonup}$ is the space of partial functions with finite domain. It is convenient to let loc be an infinite set and let all *heap*s have finite domain because this guarantees that allocations can always succeed – there are always infinitely many free locations in the heap.

The standard way to build a complete BI algebra from this type is to define a **composition** operation: $(\cdot) : heap \times heap \rightharpoonup heap$. The composition $h_1 \cdot h_2$ is defined when the domains of h_1 and h_2 are disjoint, in which case it takes the union of the two partial functions; i.e.,

$$(h_1 \cdot h_2)(l) = \begin{cases} h_1(l) & \text{if } l \in dom(h_1) \setminus dom(h_2) \\ h_2(l) & \text{if } l \in dom(h_2) \setminus dom(h_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The assertion logic can then be defined as $asn = \mathcal{P}(heap)$, which is a complete BBI algebra with the following operators.

$$\exists x : T.\ P(x) = \bigcup_{x:T} P(x) \tag{1}$$

$$\forall x : T.\ P(x) = \bigcap_{x:T} P(x) \tag{2}$$

$$P * Q = \{h_1 \cdot h_2 \mid h_1 \in P \wedge h_2 \in Q\} \tag{3}$$

To make this useful, we define a points-to operator as $l \mapsto v = \{[l \mapsto v]\}$, where $[l \mapsto v]$ is the singleton map that only maps l to v.

Although we could in principle reason directly about heaps and their composition [NVB10], it is typically considered easier to work in terms of the total operator $*$ than the partial operator $\cdot$. The $*$ operator also generalises better than $\cdot$, as we will see in Section 3.3.4.

Affine assertion logic. The assertion logic defined in Equations (1–3) is Boolean in the sense described in Section 3.2.1. More generally, for any set X, its powerset $\mathcal{P}(X)$ is a complete Boolean algebra with the quantifiers defined as in (1) and (2). To build an affine logic instead of a Boolean one, we define assertions to be closed under the **extension ordering** of heaps, defined as $h \sqsubseteq h'$ when h' has all the same mappings as h (and possibly more).

$$h \sqsubseteq h' \triangleq \exists h_0.\ h' = h \cdot h_0$$
$$asn = \mathcal{P}_{\sqsubseteq}(heap)$$

With the same definitions of $(\exists, \forall, *)$ as in Equations (1–3), this forms a complete affine BI algebra [IO01].

3.3.2 Motivations for generalisation

Other programming languages might define *heap* differently. For example, an object-oriented language [BJSB11] might have $heap = loc \times field \xrightarrow{\text{fin}} val$, where *field* is the set of field names; i.e., strings. A machine language for a 32-bit machine [JBK13] might have $heap = [0..2^{32}) \xrightarrow{\text{fin}} [0..2^8)$.

Furthermore, memory models are often **instrumented**, either at the level of operational semantics or at the logical level. An important example of such instrumentation is **fractional permissions** [BCOP05, Boy03], where a heap cell contains not only a value but also a rational number in $perm = \{q : \mathbb{Q} \mid 0 < q \leq 1\}$, where permission 1 is a read-write permission, and any smaller number is a read-only permission. Then we write $heap = loc \xrightarrow{\text{fin}} val \times perm$ and let heap composition $(\cdot)$ be defined even when there is overlap in the heap domains as long as the overlapping locations agree on values, and their permissions sum up to at most 1.

Very elaborately-instrumented heaps can be found in the work on **concurrent abstract predicates** [DYDG$^+$10, SBP13, SB13a]. These "heaps" can contain fractions, named regions, relations on (simpler) heaps, state machines, step-indexes [AM01] and ghost state.

There is thus clearly a need to construct complete BI algebras from the powersets of very elaborate structures. The good news is that there is a theory for doing exactly that in two steps: first, prove that the structure in question is a **separation algebra**. Then invoke a theorem that says that the powerset of any separation algebra, possibly closed under a suitable preorder, is a complete BI algebra. Proving that something is a separation algebra is often a very syntax-directed activity, so this approach greatly reduces the amount of work to be carried out when building an assertion logic.

We will first look at how to build a complete BI algebra from a separation algebra, and then we will look at how to build a separation algebra.

3.3.3 Definitions of separation algebra

The claim from Section 3.3.1 that the powerset of $loc \xrightharpoonup{\text{fin}} val$ is a complete BI algebra can be proved entirely based on the abstract properties of the composition operator $(\cdot)$: it forms a **partial commutative monoid**. Being a monoid means that composition is associative, which lifts to powersets and lets us prove associativity of the $*$ operator as we defined it in Equation (3). It also means that there is a unit element 0, and we can define $emp = \{0\}$ and prove that it is the unit of $*$. Commutativity lifts to powersets as well. The remaining rules in Figure 4 follow from (3) without appealing to the monoid properties.

The term **separation algebra** is used in the literature to describe structures of this kind. Unfortunately, there is little agreement on what a separation algebra is precisely. Some authors [JB12, GBC11] define it as a partial commutative monoid $(\Sigma, \cdot, 0)$ with a carrier set Σ, a partial binary operation $(\cdot)$ and a unit 0 for that operation, but the following variations, and more, exist.

- The original definition [COY07] required **cancellativity**, meaning that if $a_1 \cdot a$ and $a_2 \cdot a$ are defined and equal, then $a_1 = a_2$. This property is not important for constructing a complete BI algebra, but it can be important for validating the **conjunction rule**, which says that $\{P\}\ c\ \{Q_1\}$ and $\{P\}\ c\ \{Q_2\}$ implies $\{P\}\ c\ \{Q_1 \wedge Q_2\}$ [JB11, GBC11]. It is still common for definitions of separation algebras to include cancellativity [Tue09, DHA09, BJSB11, BK10], but it is less common that they make active use of that axiom.

- Newer definitions [DHA09, BK10, DYBG$^+$13] allow **multiple units**, where the intuition is that every element is associated with exactly one unit, but there does not have to be one unit that is compatible with every element. This effectively partitions the separation algebra into equivalence classes; one for each unit. This situation arises naturally when taking the disjoint union of two single-unit separation algebras – then there will be two units [DHA09].

- Pottier [Pot13] does not require there to be units but instead requires that there is a **core** for every element. Ordering elements by the extension order, the core of a is meant to be the largest **duplicable** element less than a, where an element is duplicable if it composes with itself to yield itself. This definition does not generalise partial commutative monoids; it is something different.
- Working with a partial monoid can be awkward. Asserting definedness of composition can be overly verbose when done explicitly [NVB10] and potentially ambiguous when done implicitly [JB12]. Some authors have addressed this by requiring a **total** (i.e., ordinary) commutative monoid and adding an absorbing element to represent undefinedness, either always [GMP05] or when necessary [KTDG12].
- Other authors have gone in the opposite direction [GMP05, GLW06, Pot13] and generalised the composition to have type $\Sigma \times \Sigma \to \mathcal{P}(\Sigma)$. This is called a **non-deterministic monoid** or, in the equivalent presentation of a composition with type $\mathcal{P}(\Sigma \times \Sigma \times \Sigma)$, a **relational monoid**.
- Dockins et al. [DHA09] proposed several more axioms that limit the class of separation algebras to those that resemble heaps in various senses.

In very recent work, Brotherston and Villard [BV13] propose a definition that generalises all of the above, except possibly Pottier's "core" concept:

Definition 3.3. A **separation algebra** is a triple $(\Sigma, \cdot, U)$ where $(\cdot) : \Sigma \times \Sigma \to \mathcal{P}(\Sigma)$ and $U \subseteq \Sigma$, and the following holds

1. Commutativity: $a \in a_1 \cdot a_2 \Rightarrow a \in a_2 \cdot a_1$
2. Assoc.: $a_{12} \in a_1 \cdot a_2 \wedge a_{123} \in a_{12} \cdot a_3 \Rightarrow \exists a_{23} \in a_2 \cdot a_3.\ a_{123} \in a_1 \cdot a_{23}$
3. Existence of unit: $\forall a.\ \exists u \in U.\ a \in u \cdot a$
4. Minimality of unit: $u \in U \wedge a' \in u \cdot a \Rightarrow a = a'$ $\diamond$

This definition is identical to the one in the Views framework [DYBG$^+$13] except that composition here is non-deterministic rather than partial. We will use this definition in the rest of this section and show several ways to construct a complete BI algebra from it.

The four axioms of Definition 3.3 may not look like a natural or obvious definition, but consider a lifting of $\cdot$ to sets $A \subseteq \Sigma$:

$$A_1 \cdot A_2 \triangleq \{a \mid \exists a_1 \in A_1.\ \exists a_2 \in A_2.\ a \in a_1 \cdot a_2\}$$

The axioms of Definition 3.3 are then equivalent to this lifted $\cdot$ being commutative and associative with unit U.

3.3.4 Upwards-closed assertions

All assertion logics I have encountered in the literature are essentially modelled as the powerset of some separation algebra Σ, upwards closed under some preorder $\leq$. That is, assertions are of type

$$\mathcal{P}_{\leq}(\Sigma) \triangleq \{P : \mathcal{P}(\Sigma) \mid \forall a \in P.\ \forall a' \geq a.\ a' \in P\}$$

The Heyting part of the logic is then a standard Kripke semantics as defined in Figure 3 (page 12).

Typical choices of the preorder are

- Equality ($=$), in which case $\mathcal{P}_{\leq} = \mathcal{P}(\Sigma)$, and the law of the excluded middle holds in the logic.
- Some other equivalence relation ($\equiv$), in which case the law of the excluded middle also holds.
- The **extension ordering** on the separation algebra ($\sqsubseteq$). We encountered the extension ordering for heaps in Section 3.3.1, and it can be generalised to arbitrary separation algebras as

  $$a \sqsubseteq a' \triangleq \exists a_0.\ a' \in a \cdot a_0$$

  As we will see below, this leads to an affine assertion logic.
- An **interference relation** [DYDG$^+$10, DYBG$^+$13] that describes how other threads may modify the state described by assertions. This enables local reasoning in a concurrent setting, at the cost of precision of the assertions.

  As a simple example [SBP13], a memory location could be tagged as containing a **monotonic counter**, which can be increased or read by any thread at any time. The interference relation is then chosen to allow for such counters to go up but not down, which means that assertions can only express that the counter has *at least* value n but not *exactly* value n.

  Another example is to let $\leq$ model the actions of a garbage collector, such as deallocating and moving objects in memory [HDV11]. Assertions closed under such a relation would be guaranteed immune to garbage collection.

While the Heyting part of $\mathcal{P}_{\leq}(\Sigma)$ is always as in Figure 3, there are at least two different ways to obtain the BI part, depending on how the separation algebra interacts with the preorder. The first is adapted from [DYBG$^+$13]:

Proposition 3.6. *If $(\Sigma, \cdot, U)$ is a separation algebra and $\leq$ is a preorder on Σ satisfying the following two conditions*

1. *The unit set is closed under the preorder; i.e., $\forall u \in U.\ \forall a \geq u.\ a \in U$.*
2. *$\forall a_1, a_2.\ \forall a \in a_1 \cdot a_2.\ \forall a' \geq a.\ \exists a'_1 \geq a_1.\ \exists a'_2 \geq a_2.\ a' \in a'_1 \cdot a'_2$; intuitively, the operands of $\cdot$ can be transported upwards along $\leq$ to follow the result.*

then a complete BI algebra is formed by $\mathcal{P}_{\leq}(\Sigma)$ with the operators defined as in Figure 3 and

$$\begin{aligned} emp &= U \\ P * Q &= \{a \mid \exists a_1 \in P.\ \exists a_2 \in Q.\ a \in a_1 \cdot a_2\} \\ P \mathrel{-\!\!*} Q &= \{a_2 \mid \forall a'_2 \geq a_2.\ \forall a_1 \in P.\ a_1 \cdot a'_2 \subseteq Q\} \end{aligned}$$

Proof. See [BJ], Section BIViews. That proof is actually of a slightly more general fact, analogous to how Proposition 3.2 generalises Proposition 3.1.□

If the conditions for Proposition 3.6 are not satisfied[3], then the following proposition might apply instead. It is adapted from [GMP02, POY04] and generalised from its original setting of partial commutative monoids to our setting of more general separation algebras.

Proposition 3.7. *If $(\Sigma, \cdot, U)$ is a separation algebra and $\leq$ is a preorder on Σ satisfying the following condition*

1. *$\forall a'_1, a'_2.\ \forall a' \in a'_1 \cdot a'_2.\ \forall a_1 \leq a'_1.\ \forall a_2 \leq a'_2.\ \exists a \leq a'.\ a \in a_1 \cdot a_2$; intuitively, the result of $\cdot$ can be transported downwards along $\leq$ to follow the operands.*

then a complete BI algebra is formed by $\mathcal{P}_{\leq}(\Sigma)$ with the operators defined as in Figure 3 and

$$\begin{aligned} emp &= \{a' \mid \exists u \in U.\ u \leq a'\} \\ P * Q &= \{a' \mid \exists a_1 \in P.\ \exists a_2 \in Q.\ \exists a \in a_1 \cdot a_2.\ a \leq a'\} \\ P \mathrel{-\!\!*} Q &= \{a_2 \mid \forall a_1 \in P.\ a_1 \cdot a_2 \subseteq Q\} \end{aligned}$$

Proof. See [BJ], Section BISepRel. □

[3] for instance, the extension ordering does not satisfy the first condition

The conditions of neither Proposition 3.6 nor Proposition 3.7 generalise the conditions of the other, so perhaps a unifying theorem is still waiting to be discovered. Notice that Proposition 3.6 gives a simple and standard definition of $*$ but a more involved definition of $-\!\!*$, while in Proposition 3.7 it is the other way around.

Proposition 3.7 has the following corollaries for special cases of $\leq$.

Corollary 3.1. *If $(\Sigma, \cdot, U)$ is a separation algebra, then a complete* Boolean *BI algebra is formed by $\mathcal{P}(\Sigma)$ with the operators defined as in Figure 3 and*

$$\begin{aligned} emp &\equiv U \\ P * Q &\equiv \{a \mid \exists a_1 \in P.\ \exists a_2 \in Q.\ a \in a_1 \cdot a_2\} \\ P -\!\!* Q &\equiv \{a_2 \mid \forall a_1 \in P.\ \forall a \in a_1 \cdot a_2.\ a \in Q\} \\ P \Rightarrow Q &\equiv \{a \mid a \in P \Rightarrow a \in Q\} \end{aligned}$$

Corollary 3.2. *If $(\Sigma, \cdot, U)$ is a separation algebra with extension ordering $\sqsubseteq$, then a complete* affine *BI algebra is formed by $\mathcal{P}_{\sqsubseteq}(\Sigma)$ with the operators defined as in Figure 3 and*

$$\begin{aligned} emp &\equiv \top \\ P * Q &\equiv \{a \mid \exists a_1 \in P.\ \exists a_2 \in Q.\ a \in a_1 \cdot a_2\} \\ P -\!\!* Q &\equiv \{a_2 \mid \forall a_1 \in P.\ \forall a \in a_1 \cdot a_2.\ a \in Q\} \end{aligned}$$

In logics modelled over $\mathcal{P}_{\leq}(\Sigma)$, primitive assertions such as points-to can typically be defined in terms of the injection $\cdot^{\uparrow} : \Sigma \to \mathcal{P}_{\leq}(\Sigma)$, defined as $a^{\uparrow} \triangleq \{a' \mid a' \geq a\}$. In words, $a^{\uparrow}$ is the smallest set in $\mathcal{P}_{\leq}(\Sigma)$ that includes a.

3.3.5 Constructions

In recent work on separation logic and related formalisms [JB12, KTDG12, LWN13, DYGW10], each module of the program can have its own separation algebra, so the task of verifying a module includes coming up with a separation algebra suitable for it and checking the conditions in Definition 3.3. While this is already much simpler than proving that something is a complete BI algebra, we can make it even simpler still, because separation algebras are very compositional. The following proposition is adapted from [DHA09] and [JB12].

Proposition 3.8. *Given separation algebras $(\Sigma_1, \cdot_1, U_1)$ and $(\Sigma_2, \cdot_2, U_2)$ and an arbitrary type T,*

1. *The* ***product*** *$\Sigma_1 \times \Sigma_2$ is also a separation algebra with unit $U_1 \times U_2$ and composition $(a_1, a_2) \cdot (b_1, b_2) \triangleq (a_1 \cdot_1 b_1) \times (a_2 \cdot_2 b_2)$.*

2. *The* ***tagged union*** $\Sigma_1 + \Sigma_2 \triangleq (\{1\} \times \Sigma_1) \cup (\{2\} \times \Sigma_2)$ *is also a separation algebra with unit* $U_1 + U_2$ *and composition as the smallest relation satisfying* $(i, a) \cdot (i, b) = \{i\} \times (a \cdot_i b)$ *for* $i \in \{1, 2\}$*.*

3. *The set* T *can be viewed as a* ***discrete separation algebra*** T_{discr} *if we define the units as* $U \triangleq T$ *and composition as the smallest relation satisfying* $t \cdot t = \{t\}$*.*

4. *The space* $T \overset{\text{fin}}{\Rightarrow} \Sigma_1$ *of* ***finitely-supported functions*** *is a separation algebra. Being finitely supported means that only a finite number of values from the domain are mapped to non-unit values. The units are the functions mapping everything to some unit, and composition is pointwise:*

$$U \triangleq \{f \mid \forall t.\ f(t) \in U_1\}$$
$$f \cdot g \triangleq \{h \mid \forall t.\ h(t) \in f(t) \cdot_1 g(t)\}$$

Proof. See [BJ] for items 1,2,4. See [DHA09] for item 3. □

The space of finitely-supported functions ($\overset{\text{fin}}{\Rightarrow}$) has good composition properties. In particular, it allows currying, so the set $(A \times B) \overset{\text{fin}}{\Rightarrow} \Sigma$ is isomorphic to the set $A \overset{\text{fin}}{\Rightarrow} (B \overset{\text{fin}}{\Rightarrow} \Sigma)$. This is in contrast to the space of finite partial functions ($\overset{\text{fin}}{\rightharpoonup}$), which does not have this isomorphism [Par05] since the curried form allows distinguishing between the values [] (the empty map) and $[a \mapsto []]$ (a singleton map that maps a to the empty map). We found that proofs of deallocation in fictional separation logic [JB12, JB11] became much simpler when using ($\overset{\text{fin}}{\Rightarrow}$) instead of ($\overset{\text{fin}}{\rightharpoonup}$).

We will in practice need more constructions than those given in Proposition 3.8. In fictional separation logic [JB12], we found it useful to revive the concept of a **permission algebra** [COY07], which is like a separation algebra but without units:

Definition 3.4. A **permission algebra** is a pair $(\Pi, \cdot)$ where $(\cdot) : \Pi \times \Pi \to \mathcal{P}(\Pi)$, and the following holds

1. Commutativity: $a \in a_1 \cdot a_2 \Rightarrow a \in a_2 \cdot a_1$

2. Assoc.: $a_{12} \in a_1 \cdot a_2 \wedge a_{123} \in a_{12} \cdot a_3 \Rightarrow \exists a_{23} \in a_2 \cdot a_3.\ a_{123} \in a_1 \cdot a_{23}$ ◇

A similar but more restrictive definition is given in [Hob11] and used for the same purpose: to serve as an intermediate structure when composing a separation algebra.

Proposition 3.9.

1. *Permission algebras form products and tagged unions just like separation algebras do.*

2. *A permission algebra Π together with a fresh unit element* 0 *can be viewed as a separation algebra* $(\Pi)_0 \triangleq \Pi \cup \{0\}$ *with units* $\{0\}$ *and composition defined as in* Π *for non-unit elements and as* $0 \cdot a = a \cdot 0 = \{a\}$ *in other cases.*

3. *Any set* T *can be viewed as an* ***equality permission algebra*** $T_=$ *if we define composition as the smallest relation satisfying* $t \cdot t = \{t\}$*.*

4. *Any set* T *can be viewed as an* ***empty permission algebra*** $T_\emptyset$ *if we define composition as* $t \cdot t' = \emptyset$*.*

With these constructions, we can now redefine the heaps from Section 3.3.1 as $\mathit{heap} \triangleq \mathit{loc} \overset{\text{fin}}{\rightharpoonup} (\mathit{val}_\emptyset)_0$. We have described the same separation algebra $(\mathit{heap}, \cdot, [])$ as before, but this time there is nothing further to define or prove. The composition operation and its properties follow syntactically from Propositions 3.8 and 3.9, and the fact that $\mathcal{P}(\mathit{heap})$ forms a complete BBI algebra follows from Corollary 3.1.

We can also define heaps with permissions [BCOP05, Hob11] for any permission algebra Π as $\mathit{heap}_\Pi \triangleq \mathit{loc} \overset{\text{fin}}{\rightharpoonup} (\mathit{val}_= \times \Pi)_0$. For further examples, see [JB12, JB11].

As already mentioned, the study of separation algebras is still at an early stage, and the constructions presented here could soon be superseded by better ones. Not all definitions of separation algebra support all the constructions; in particular, multiple units are needed to support tagged unions and discrete separation algebras [DHA09]. For most of the alternative definitions of separation algebras discussed in Section 3.3.3, none of the constructions have been verified. One exception is [DHA09], which proposes several specialisations of separation algebras and verifies that each one is preserved by all constructions.

3.3.6 Cyclic definitions

Advanced separation logics often feature instrumented heaps that can "store" assertions. Examples of such stored assertions include the invariant associated with a storable lock [GBC+07], the operations allowed on a shared resource [DYDG+10], or the precondition of a procedure stored in memory [NS06].

A representative example of this situation, inspired by models of storable locks, could be the following attempt to define heaps:

$$\Sigma = \mathit{loc} \overset{\text{fin}}{\rightharpoonup} ((\mathit{val} \times \mathit{asn})_\emptyset)_0$$

If $\mathit{asn} = \mathcal{P}_\leq(\Sigma)$ as usual, then this definition becomes cyclic, with Σ in a negative position:

$$\Sigma = \mathit{loc} \overset{\text{fin}}{\rightharpoonup} ((\mathit{val} \times \mathcal{P}_\leq(\Sigma))_\emptyset)_0$$

There is no set-theoretic solution to this equation, so it cannot be used as a definition.

A comprehensive treatment of the techniques that apply here is beyond the scope of this text, but the solutions can roughly be grouped into three types, ordered here by increasing expressiveness of the resulting logic.

1. Store a syntactic assertion [VN13, DYDG+10] or token [GBC+07] instead of a semantic assertion. This can work well enough for first-order theories.

2. Use **step-indexing** or similar techniques [AMRV07, DHA09, BRS+11] to **guard** the recursive occurrence. This essentially creates an approximation of the recursively-defined heap up to $n+1$ recursive iterations, exploiting that a program that has only n steps of execution left will not have time to observe what lies beyond that depth in the heap when a heap dereference takes one step. Specification validity then means that the program is valid for arbitrary values of this n.

3. The separation logic can be developed in a metalogic that does not restrict recursive occurrences to being strictly positive in the traditional sense. The **topos of trees** [BMSS12] has recently been proposed for this purpose; it allows negative occurrences as long as they are guarded by a modal operator $\triangleright$. In the model of the topos of trees, this modal operator is explained in terms of step-indexing, so this technique is sound for essentially the same reason as item 2 above.

   See [SB13a] for a recent example of using the topos of trees as the metalogic of an impredicative concurrent separation logic.

Step-indexing in logical propositions, rather than types, are discussed in Section 4.2.2.

3.4 Program variables

We have so far discussed assertions quite abstractly, but ultimately they are of course used in pre-and postconditions of commands, and they must be able to describe the values of **program variables** as named in the source program. The exact technique will necessarily be specific to the programming language, but there are some common patterns and even some reusable theory just like there was for heaps.

Using a shallow embedding gave us typed *logical* variables practically for free, but there is no such shortcut for *program* variables. Fortunately, program variables still tend to be simpler to support than logical variables since program variables tend to have a more restricted binding structure.

When every formal detail has to be right – especially when working in a proof assistant – then there are many pitfalls in the encoding of program variables. This section surveys the techniques that have been proposed for handling program variables in various programming languages. The goal is to make program variables behave much like logical variables, which is the tradition in Hoare logic, while still retaining all the benefits of a shallow embedding.

The semantics of a programming language tends to divide the state into a heap and a **stack** (i.e., stack frame). Shared mutable data lives on the heap, while the content of local variables lives on the stack. Some authors use the term **store** instead of stack. Stacks in While-like toy programming languages are typically modelled as $\mathit{stack} \triangleq \mathit{var} \to \mathit{val}$. This also suffices for modelling many realistic languages such as Java [PB05, BJSB11] or assembly [CSV07, Myr10, JBK13], where the type *var* is chosen as strings or register names respectively.

Other languages have more complex stacks, where a simple mapping from variables to values does not suffice. This tends to happen when the language enables access to the L-value of local variables, either with an explicit address-of operation as in the C programming language, or implicitly through variable capture [SBP10]. Complications may also arise in concurrent languages, where the stack becomes shared when threads fork. Variable scoping rules in JavaScript is a whole research topic in itself [GMS12].

The rest of this section assumes that (instrumented) machine states can be modelled as $\mathit{stack} \times \mathit{heap}$ for some definition of *heap*. Even separation logics for the C programming language adopt this model and simply disallow access to the address of local variables [AB07, TKN07, JSP12, AM13].

Using the constructions from Section 3.3.5, there are at least two useful ways to turn the whole machine state into a separation algebra.

1. If we let *stack* be a discrete separation algebra, then the product $\mathit{stack}_{\mathrm{discr}} \times \mathit{heap}$ is a separation algebra whose composition is defined by
$$(s,h) \in (s_1,h_1) \cdot (s_2,h_2) \iff s = s_1 = s_2 \land h \in h_1 \cdot h_2$$
With a standard construction to form a complete BI algebra from $\mathit{stack}_{\mathrm{discr}} \times \mathit{heap}$, such as Corollary 3.1, we obtain the same definition of $*$ as in the vast majority of separation-logic texts:
$$P * Q \equiv \{(s,h) \mid \exists h_1, h_2.\ h \in h_1 \cdot h_2 \land (s,h_1) \in P \land (s,h_2) \in Q\}$$
A drawback of this approach is that it typically requires a syntactic side condition on the frame rule to say that variables free in the frame R must not be modified by the command c:
$$\frac{\{P\}\ c\ \{Q\} \qquad \mathit{modifies}(c) \cap \mathit{fv}(R) = \emptyset}{\{P * R\}\ c\ \{Q * R\}}$$

A simple way to get rid of this side condition is to make local variables immutable [BTSY06], but this can of course be a major restriction of the programming language.

2. We can alternatively define stacks almost like heaps: $\mathit{stack} = \mathit{var} \xrightarrow{\mathsf{fin}} (\mathit{val}_\emptyset)_0$. This approach is known as **variables as a resource**, and the original paper about this idea [PBC06] goes even further and adds fractional permissions *perm*, defining $\mathit{stack} = \mathit{var} \xrightarrow{\mathsf{fin}} (\mathit{val}_= \times \mathit{perm})_0$.

   With variables as a resource, we get a more aesthetically-pleasing frame rule because the syntactic side condition essentially becomes integrated into the definition of $*$.

   $$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

   This is also formally better in cases where $\mathit{modifies}(c)$, the set of local variables potentially modified by c, is not easy to determine. This happens in languages where we cannot syntactically see from a program what variables might be modified, or when that over-approximation is too coarse [JBK13, MG07].

   The drawback is that the convenient similarity between program variables and logical variables is lost. Program variables have to be treated like heap locations using some type of points-to predicate, which complicates the rules for variable assignment and conditionals [PBC06] [DYBG+13, Definition 17].

3.4.1 Open terms and lifting

The two constructions above allow us to have program variables in assertions, but that was only half of the problem. We also need program variables in expressions, including logical expressions that are not part of the programming language. For instance, we might like to assert that $\mathsf{n} > \mathsf{m} + 1$, where n and m are written in a sans-serif font because they are program variables; i.e., symbols of type *var*.

If we are using variables as a resource, the example assertion above could be written as

$$\exists m.\ \mathsf{m} \mapsto m * \exists n.\ \mathsf{n} \mapsto n \wedge \langle n > m + 1\rangle.$$

Notice first the distinction between the variable name m and its value m, which makes the formula somewhat verbose. Syntactic sugar has been proposed to reduce this somewhat [PBC06], but it comes at a price: expected identities such as $e_1 \neq e_2 \equiv \neg(e_1 = e_2)$ fail to hold. On the other hand, the verbosity is not much of a burden in assembly language, where the "program variables" are uninformative register names, and their values tend to be named differently from the registers that hold them [JBK13, MG07].

The rest of this section tries to follow the Hoare-logic tradition of referring to program variables from deep inside expressions. As a first step, our example assertion of $\mathsf{n} > \mathsf{m} + 1$ can be written formally as

$$\{(s, h) \mid s(\mathsf{n}) > s(\mathsf{m}) + 1\},$$

but this is undesirable as it exposes the stack s, which increases verbosity and looks quite different from standard presentations. With some amount of syntactic sugar, it can be made practical, though [McC09].

In Charge! [BJB12, BJ], a Coq formalisation of separation logic, we distinguish between *assertions* and *open assertions*, which I will denote here as

$$\mathit{asn} \triangleq \mathcal{P}(\mathit{heap}) \qquad \text{and} \qquad \mathit{open\ asn} \triangleq \mathit{stack} \to \mathit{asn}$$

respectively. The type *open asn* is a complete BI algebra, and it is in fact isomorphic[4] to $\mathcal{P}(\mathit{stack}_{\mathrm{discr}} \times \mathit{heap})$. In general:

Proposition 3.10. *Given a complete BI algebra A and a preordered type $(T, \leq)$, the space of monotonic functions $T \to_{\leq} A$ is a complete BI algebra, where*

$$T \to_{\leq} A \triangleq \{f : T \to A \mid \forall t, t'.\ t \leq t' \Rightarrow (f(t) \vdash f(t'))\}$$

and the operators of $T \to_{\leq} A$ are defined in terms of the operators on A:

$$\begin{aligned} P * Q &\triangleq \lambda t.\ P(t) * Q(t) \\ \mathit{emp} &\triangleq \lambda t.\ \mathit{emp} \\ P \mathrel{-\!\!*} Q &\triangleq \lambda t.\ \forall t' \geq t.\ P(t') \mathrel{-\!\!*} Q(t') \end{aligned}$$

The Heyting operators are defined as in Proposition 3.2 (page 12).

Proof. See [BJ], Lemma $\mathsf{BILPreLogic}$. □

We can extend the definition of *open asn* to arbitrary types T:

$$\mathit{open}\ T \triangleq \mathit{stack} \to T$$

This allows us to give a uniform treatment of free variables and substitutions on **open terms** regardless of their type. We can **lift** functions to work on open terms, ranged over by o:

Definition 3.5. Given an n-ary function $f : (T_1 \times \cdots \times T_n) \to T$, define $\dot{f} : (\mathit{open}\ T_1 \times \cdots \times \mathit{open}\ T_n) \to \mathit{open}\ T$ as

$$\dot{f}(o_1, \ldots, o_n) \triangleq \lambda s.\ f(o_1(s), \ldots, o_n(s)).$$

We lift constants $t : T$ to $\dot{t} : \mathit{open}\ T$, by taking $n = 0$ above. Finally, we overload the same notation to mean something different for program variables $x : \mathit{var}$, defining $\dot{x} : \mathit{open\ val}$ as $\dot{x} \triangleq \lambda s.\ s(x)$. ◇

[4]They are isomorphic as complete BI algebras, meaning that the isomorphism preserves all operators.

Returning to our example assertion, we may now write it formally as

$$\dot{\mathsf{n}} \mathbin{\dot{>}} \dot{\mathsf{m}} \mathbin{\dot{+}} \dot{1}.$$

We can just as easily lift functions from the metalogic that do not also exist as programming-language expressions; for instance, given $\mathit{fac} : \mathbb{N} \to \mathbb{N}$, we can express that variable n holds the factorial of variable m as

$$\dot{\mathsf{n}} \mathbin{\dot{=}} \dot{\mathit{fac}}(\dot{\mathsf{m}}).$$

As a final and important example, we can lift operators on types that have no representation in the programming language; e.g., *list cons* and recursively-defined *list predicates* [BJB12]. This lets us give a satisfying formal account of how we reason with arbitrary mathematics inside assertions, without implicitly assuming that the theories we need have been reconstructed from scratch within *open asn*.

While the benefits of *open T* presented thus far could be dismissed as being superficial, we found in [BJSB11][5] that the lifting concept was crucial for harnessing the abstraction and modularity benefits of *higher-order* separation logic. It is a standard pattern of specification in higher-order separation logic to quantify and parametrise over assertion-logic predicates [BBTS05, PB08, BJSB11]. This means that formulas tend to involve opaque predicates F, and eventually we will have to ask what are the free variables of, say, $F(e)$. One would hope that $\mathit{fv}(F(e)) \subseteq \mathit{fv}(e)$, but this depends on the definition of F. Examples of "undisciplined" F include

$$\begin{aligned} F(e) &= e \mathbin{\dot{=}} \dot{\mathsf{x}} \\ F(e) &= e[\mathsf{y}/\mathsf{x}] \mathbin{\dot{=}} \dot{0} \\ F(e) &= \langle \mathsf{x} \in \mathit{fv}(e) \rangle \end{aligned}$$

If there is only one type of assertions, $\mathcal{P}(\mathit{stack} \times \mathit{heap})$, then it is difficult to statically rule out such undesired values in a shallow embedding. See the discussions in [App06], where side conditions about free variables and substitutions have to be carried around with predicates. In contrast, the lifting approach always gives us well-behaved free variables and substitutions. The following two subsections will define semantic notions of free variables and substitutions such that the following holds:

$$\begin{gathered} \mathit{fv}(\dot{f}(o_1, \ldots, o_n)) \subseteq \mathit{fv}(o_1) \cup \ldots \cup \mathit{fv}(o_n) \\ \dot{f}(o_1, \ldots, o_n)[\bar{e}/\bar{x}] = \dot{f}(o_1[\bar{e}/\bar{x}], \ldots, o_n[\bar{e}/\bar{x}]) \end{gathered}$$

For further examples and motivation, see [BJSB11, BJB12].

[5] In that paper, *open T* is called *sm T*, and lifting is written $\hat{f}$ instead of $\dot{f}$.

3.4.2 Free variables

Following Appel et al. [AB07], we can characterise free variables semantically by saying that x is free in $o : \mathit{open}\ T$ when a change to x can cause a change to o:

$$\mathit{fv}(o) \triangleq \{x \mid \exists s, v.\ o(s) \neq o(s[x := v])\}$$

An open term can have an infinite number of free variables; for instance, every variable is free in $\forall x : \mathit{var}.\ \dot{x} \doteq \dot{0}$. We might like to forbid such terms, but it can be hard to do without restricting expressiveness, and it turns out we do not need to. Compare this to *nominal logic* [Pit01], which is much more well-behaved. There, open terms have a finite number of free variables and elegant support for binders in the programming language, but the approach requires the entire metalogic to be replaced.

The definition of *fv* satisfies convenient rules for how it applies to pointwise-lifted functions and to variables:

$$\begin{gathered}\mathit{fv}(\dot{f}(o_1, \ldots, o_n)) \subseteq \mathit{fv}(o_1) \cup \ldots \cup \mathit{fv}(o_n)\\ \mathit{fv}(\dot{x}) \subseteq \{x\}\end{gathered}$$

The property on $\dot{f}$ only holds with inclusion, not equality, since a variable may not be semantically free even if it occurs in an expression – for instance, $\mathit{fv}(\dot{\mathsf{x}} \mathbin{\dot{-}} \dot{\mathsf{x}}) = \emptyset$. The property on $\dot{x}$ of course holds with equality for any non-trivial choice of *val*.

3.4.3 Substitutions

It is standard, both in deep and shallow embeddings, to define a substitution $\rho : \mathit{subst}$ as a function from variables to expressions, which here means

$$\mathit{subst} \triangleq \mathit{var} \to \mathit{open\ val}$$

Expressions, ranged over by e, are of type *open val* in their shallowly-embedded form. A simultaneous substitution of n distinct variables can be defined as

$$[e_1, \ldots, e_n \,/\, x_1, \ldots, x_n] \triangleq \lambda x. \begin{cases} e_i & \text{if } x = x_i \text{ for some } i \\ \dot{x} & \text{otherwise} \end{cases}$$

We can then define a semantic notion of applying a substitution to an open term. This is written here with postfix notation as per tradition, and it is defined in terms of applying a substitution to a stack.

$$\begin{aligned}&o\rho \triangleq \lambda s.\ o(s\rho), \text{ where}\\ &s\rho \triangleq \lambda x.\ \rho(x)(s)\end{aligned}$$

From these definitions, we can prove as lemmas how substitution acts on pointwise-lifted functions and on variables.

$$(\dot{f}(o_1,\ldots,o_n))\rho = \dot{f}(o_1\rho,\ldots,o_n\rho)$$
$$(\dot{x})\rho = \rho(x)$$

Contrast this to how substitutions work in a deep embedding: the *lemmas* above about how ρ acts on $\dot{f}$ and $\dot{x}$ would be taken as the *definition* of substitution on syntactic terms, and our *definition* of $o\rho$ would instead be proved as a *lemma* [SBP09, Lemma 10] [Kri12, Lemma 28.2.a.ii].

3.4.4 Typed values

Separation logic is often applied to typed programming languages such as Java [Par05, BJSB11] or C [TKN07, AB07]. The typical approach is then to generalise the syntax and operational semantics to remove static types and let the separation logic enforce typing instead – a program logic generalises a simple type system, so it is a burden to have both. It is standard [Rey02, AB07, McC09, TSFC09, SBP10, BJSB11] to do as we have been doing since Section 3.3.1 and define a type *val* as a tagged union of integers, pointers, Booleans and any other types that can be stored in local variables or on the heap.

The question is then whether an arithmetic operator, such as $>$, should have type $val \times val \to val$ or $int \times int \to bool$. In the first approach, we can immediately write logic expressions such as $\dot{\mathsf{x}} \mathrel{\dot{>}} \dot{\mathsf{y}}$, and the types will match up. On the other hand, we need to have an answer for how to compare, say, a pointer with a Boolean, even though such a comparison could never happen in the original, typed, programming language. This issue extends to any other operator we want to lift from the metalogic.

In the other approach, where $(>) : int \times int \to bool$, we cannot write $\dot{\mathsf{x}} \mathrel{\dot{>}} \dot{\mathsf{y}}$, since $\dot{>}$ has type $open\ int \times open\ int \to open\ bool$ while $\dot{\mathsf{x}}$ and $\dot{\mathsf{y}}$ have type $open\ val$. One option is to read variables not as an untyped $\dot{x} : open\ val$ but as a typed $intvar(x) : open\ int$ etc. Then we can write $intvar(x) \mathrel{\dot{>}} intvar(y)$. The problem that a *value* could have an unexpected type is now replaced with the problem that a *variable* can have an unexpected type, and *intvar* will have to return a dummy value, such as 0, if it reads a non-*int*.

We have tried both approaches in Charge! [BJSB11, BJB12], and they both ended up littering specifications and proofs with distracting coercions in and out of *val*.

A third approach is to make expression evaluation partial [PBC06, AB07], but this has its own set of problems; for instance, expected identities such as $e_1 \neq e_2 \equiv \neg(e_1 = e_2)$ no longer hold [PBC06]; here, $(\neq, =)$ are partial expressions, and $\neg$ is from the assertion logic. Despite this, many authors model stacks as partial functions without explaining what happens when lookup fails.

It is worth looking at two separation logics that are not affected by these problems at all, even down to the last formal detail.

- In [JBK13, KBJD13], we define a separation logic for x86 machine code. Assertions are of type $\mathcal{P}(\Sigma)$, where

$$\begin{aligned}\Sigma = {} & (\mathit{register} \overset{\text{fin}}{\rightharpoonup} (\mathit{DWORD}_\emptyset)_0) \times \\ & (\mathit{flag} \overset{\text{fin}}{\rightharpoonup} (\mathit{bool}_\emptyset)_0) \times \\ & (\mathit{DWORD} \overset{\text{fin}}{\rightharpoonup} (\mathit{BYTE}_\emptyset)_0)\end{aligned}$$

  The three components in the product denote CPU registers, CPU flags and main memory respectively; types BYTE and DWORD denote bool^8 and bool^{32} respectively.

  The registers and flags together can be thought of as the local variables – the *stack*, in our current terminology – and this stack can store both DWORD and bool values. The separation-algebra annotations on Σ reveal that we are using the variables-as-a-resource approach, but this is not the essence of why types on the stack work out here. It works because there is a separate name space for the registers and flags; i.e., EAX is a register, and it is clear from its name only that it holds a DWORD and not a bool. This approach can also work in more conventional programming languages [CGZ05].

  The main memory can be thought of as the *heap* in our current terminology. Types on the heap work out for a completely different reason than for the stack. To let us store other things than BYTEs in memory, there is essentially a points-to predicate $\mapsto_T$ for every type T that has a defined decoding from byte sequences to T. Then $l \mapsto_T v$ holds when $v : T$, and the memory contents starting at l decodes to v. See also [TKN07, AM13] for related approaches with slightly different goals.

- Another approach is to not *replace* the type system with a program logic but instead *extend* the type system until it becomes as powerful as a program logic. Programming-language terms with side effects, such as heap write, are given a type describing those effects, such as $\{P\}\{Q\}$, where P and Q can refer to program variables. Logical entailment is encoded as subtyping.

  Examples include [BTSY06, NAMB07, Pot08, KTDG12]. Since this requires either a deep embedding or reverification of the whole metatheory [NMS$^+$08, CMM$^+$09], we lose the advantages gained from having a shallow embedding. Features such as higher kinds and dependent types must be re-created within the type system rather than borrowed from the metalogic. See also the discussion in Section 4.3.2.

The unproblematic logics mentioned above have one thing in common: they do not define a *val* type, but instead they keep all programming-language types explicit and separate.

The problem also seems to go away when using variables as a resource. Recall the example from Section 3.4.1, where we wrote

$$\exists m.\ \mathsf{m} \mapsto m * \exists n.\ \mathsf{n} \mapsto n \wedge \langle n > m + 1 \rangle.$$

In a setting where the injection from *int* to *val* is called *intval*, we can write this assertion more explicitly as

$$\exists m : \mathit{int}.\ \mathsf{m} \mapsto \mathit{intval}(m) * \exists n : \mathit{int}.\ \mathsf{n} \mapsto \mathit{intval}(n) \wedge \langle n > m + 1 \rangle.$$

This solves the problem and can be useful for any type T with an injective function $T \to \mathit{val}$. On the other hand, variables as a resource remains very verbose and does not look like standard Hoare logic.

The above pattern for getting typed program variables using a *stack* version of points-to predicate will work just as well for any standard *heap* points-to predicate. Since separation logic, with very few exceptions [SJP10, PS12], forbids direct heap references in expressions, we are already forced into this pattern of existential quantification and distinction between a heap location and its value, so this may as well be used to get stronger typing.

4 Specifications

The primitive unit of specification in separation logic is usually the **Hoare triple**. In the most basic form, in a shallow embedding, the triple is a predicate in the metalogic. Section 4.1 discusses definitions and inference rules for the triple based on this assumption.

Section 4.2 will demonstrate benefits and techniques for considering the triple instead as a formula of **specification logic** [Rey82], which allows us to give a logical account of the context in which a given triple holds.

4.1 Hoare triples

4.1.1 Definitions

The Hoare triple is where the assertion logic from Section 3 meets the operational semantics from Section 2. The Hoare triple for **partial correctness** is usually defined to mean, intuitively, "for any state satisfying the precondition, no execution from that state will crash, and any *terminating* execution from that state will result in a state satisfying the postcondition". In the most basic form, the triple is defined as

$$\begin{aligned}\{P\}\ c\ \{Q\}_1 \triangleq \forall \sigma \in P.\ &\neg(\sigma, c \rightsquigarrow \mathsf{fail}) \wedge \\ &\forall \sigma'.\ \sigma, c \rightsquigarrow \sigma' \Rightarrow \sigma' \in Q\end{aligned}$$

For partial correctness, we are content with ignoring divergence, but we will not ignore failure.

Contrast this to **total correctness**, where we additionally require the command to *terminate* from any state satisfying the precondition. If there is a relation $\sigma, c \rightsquigarrow \infty$ meaning that σ, c may diverge, then a basic definition of the Hoare triple for total correctness can be

$$\begin{aligned}[P]\ c\ [Q]_2 \triangleq \forall \sigma \in P.\ &\neg(\sigma, c \rightsquigarrow \mathsf{fail}) \land \neg(\sigma, c \rightsquigarrow \infty)\\ &\forall \sigma'.\ \sigma, c \rightsquigarrow \sigma' \Rightarrow \sigma' \in Q\end{aligned}$$

The relation $\sigma, c \rightsquigarrow \infty$ is a simple coinductive definition for a small-step semantics, and it was discussed for big-step semantics in Section 2.1. Another approach is to describe the absence of failure and divergence together in one predicate [Nip02].

If the semantics is deterministic, then total correctness can be defined much more succinctly as

$$[P]\ c\ [Q]_3 \triangleq \forall \sigma \in P.\ \exists \sigma'.\ \sigma, c \rightsquigarrow \sigma' \land \sigma' \in Q$$

Total correctness is treated only in a minor portion of the separation-logic literature. It can be argued that there is no practical difference between a diverging program and one that terminates after a million years, but total correctness is nevertheless important for discovering bugs. See [Atk10] for an interesting take on amortised running-time analysis with separation logic.

The remainder of this chapter will discuss partial correctness only, but most concepts can be extended to total correctness.

The triples defined above all satisfy the **rule of consequence** and the **existential rule**:

$$\frac{P \vdash P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{\{P\}\ c\ \{Q\}} \qquad\qquad \frac{\forall x.\ \{P(x)\}\ c\ \{Q\}}{\{\exists x.\ P(x)\}\ c\ \{Q\}}$$

Whether they satisfy the frame rule, however, depends on whether the operational semantics satisfies the frame property and safety monotonicity[6], as discussed in Section 2. If these properties should not hold, we can still get the frame rule by instead defining the triple as follows:

$$\{P\}\ c\ \{Q\}_4 \triangleq \forall R.\ \{P * R\}\ c\ \{Q * R\}_1$$

With this definition, we are guaranteed to have the rule of consequence, the existential rule and the frame rule. The technique was first used [BTSY06] in the setting of a higher-order programming language, where it was not clear how to define the frame property, let alone prove it [RS06, BRSY08].

The new triple, $\{P\}\ c\ \{Q\}_4$, is sound with respect to $\{P\}\ c\ \{Q\}_1$ but may not be complete. For example, if the memory allocator is deterministic and always allocates the smallest free location [YO02], then we can no

[6] Safety monotonicity is not required for an affine assertion logic [BJSB11].

longer prove a triple that describes this fact; we can only prove the usual triple for allocation, where the new location is existentially quantified in the postcondition. This can be considered a shortcoming of the theory or a gain in abstraction, depending on viewpoint.

So far, we have implicitly assumed that assertions belong to a complete BI algebra of type $\mathcal{P}_{\leq}(\mathit{state})$ as constructed in Section 3.3.4, where *state* is the type of states in the operational semantics. But interesting separation logics often have some form of instrumentation, or annotations, in the assertions that is not present in the operational semantics. One example is **fractional permissions** [BCOP05], where each heap location has a permission value as well as a data value.

A powerful pattern for defining a triple in such cases has recently emerged [DYBG$^+$13, JB12]. A function $\mathit{reify} : \mathit{asn} \to \mathcal{P}(\mathit{state})$ is defined to translate from instrumented assertions to machine state. Then the triple can be defined as

$$\{P\}\ c\ \{Q\}_5 \triangleq \forall R : \mathit{asn}.\ \{\mathit{reify}(P * R)\}\ c\ \{\mathit{reify}(Q * R)\}_1$$

We still require *asn* to be a complete BI algebra, but $\mathcal{P}(\mathit{state})$ need not be. If we additionally require *reify* to preserve existentials[7] and to be covariant with respect to entailment[8], then the rules of consequence, existential and frame all hold for this triple. Those two properties always hold [JB11, DYBG$^+$13] if *asn* has been constructed from a separation algebra Σ like in Section 3.3.4 and the *reify* function has been lifted from some $f : \Sigma \to \mathcal{P}(\mathit{state})$ as

$$\mathit{reify}(P) = \bigcup_{a \in P} f(a)$$

4.1.2 Structural rules

An inference rule is informally called a **structural rule** when all Hoare triples in it have the same universally-quantified c as their command.

We have already discussed the rule of consequence, the existential rule and the frame rule:

$$\frac{P \vdash P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \vdash Q}{\{P\}\ c\ \{Q\}}\ \textsc{Consequence}$$

$$\frac{\forall x.\ \{P(x)\}\ c\ \{Q\}}{\{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists} \qquad\qquad \frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

In Section 1.2, I attempted to motivate why these three rules are essential in a separation logic. Any Hoare triple that does not satisfy these rules should come with a good explanation of why not.

[7] Means that $\mathit{reify}(\exists x : T.\ P(x)) = \bigcup_{x:T} \mathit{reify}(P(x))$.
[8] Means that $P \vdash Q$ implies $\mathit{reify}(P) \subseteq \mathit{reify}(Q)$.

There are a few variations on the above rules. Many authors print the existential rule as

$$\frac{\forall x.\ \{P(x)\}\ c\ \{Q(x)\}}{\{\exists x.\ P(x)\}\ c\ \{\exists x.\ Q(x)\}}\ \textsc{Exists}'$$

This has a somewhat satisfying symmetry to it, but it is derivable from Exists and Consequence.

The **disjunction rule** and **vacuity rule** [Rey11] shown below are derivable from Exists because $\vee$ and $\bot$ can be seen as special cases of the existential quantifier as shown in Section 3.2.2.

$$\frac{\{P_1\}\ c\ \{Q\} \quad \{P_2\}\ c\ \{Q\}}{\{P_1 \vee P_2\}\ c\ \{Q\}}\ \textsc{Disjunction} \qquad\qquad \frac{}{\{\bot\}\ c\ \{Q\}}\ \textsc{Vacuity}$$

Like the existential rule, the disjunction rule often appears in the literature in a more symmetric but redundant form.

As discussed in Section 3.4, the frame rule typically comes with the side condition that $\mathit{modifies}(c) \cap \mathit{fv}(R) = \emptyset$. We have explored a variation of the frame rule in the Charge! platform [BJB12, BJ], where that side condition is replaced by a substitution:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * \exists\bar{v} : \mathit{val}.\ R[\bar{v}/\mathit{modifies}(c)]\}}\ \textsc{Frame}$$

The notation is meant to suggest that if $\mathit{modifies}(c) = \{x_1, \ldots, x_n\}$, then $\exists\bar{v} : \mathit{val}.\ R[\bar{v}/\mathit{modifies}(c)]$ means

$$\exists v_1, \ldots, v_n : \mathit{val}.\ R[v_1, \ldots, v_n \,/\, x_1, \ldots, x_n]$$

Essentially, instead of preventing the frame rule from being applied at all, we weaken it to the extent needed for it to hold. This rule is not formally stronger than the standard one, but it allowed us to develop a separation logic without the concept of free variables, which meant there was one less concept to build theory and automation for.

Finally, the **conjunction rule** is of some interest:

$$\frac{\{P\}\ c\ \{Q_1\} \quad \{P\}\ c\ \{Q_2\}}{\{P\}\ c\ \{Q_1 \wedge Q_2\}}\ \textsc{Conjunction}$$

This rule holds for simple definitions of the Hoare triple, such as $\{P\}\ c\ \{Q\}_1$ through $[P]\ c\ [Q]_3$ above, but it fails for many other definitions. Much has been written about what restrictions must be placed on the logic for the conjunction rule to hold [OYR04, BTSY06, O'H07, DYGW11, GBC11, JB11], but comparatively little has been written about why this rule is useful at all.

Some generalisations of the conjunction rule have been proposed. When it holds for binary conjunctions, then it typically also holds for universal quantification over a non-empty domain [Rey11, DYBG$^+$13]:

$$\frac{\forall x : T.\ \{P\}\ c\ \{Q(x)\} \quad T \neq \emptyset}{\{P\}\ c\ \{\forall x : T.\ Q(x)\}}\ \textsc{Universal}$$

In fictional separation logic [JB11], the conjunction rule is generalised to the **recombination rule**, parametrised over a binary operator $\Box$:

$$\frac{\{P\}\ c\ \{Q_1\} \quad \{P\}\ c\ \{Q_2\}}{\{P\}\ c\ \{Q_1 \Box Q_2\}}\ \textsc{Recombination}$$

This rule holds for $\Box = \wedge$ in some cases and for $\Box = *$ in other cases [JB11].

4.2 Specification logic

4.2.1 Example: procedure map

The machine state of more realistic languages contains more than a stack and a heap. A language with procedures could, for example, have some form of map $m : M$ from procedure names to their implementation code. The map could in principle be added to the machine state alongside the stack and heap, so states would be $\sigma = (s, h, m)$. But when no command can modify m, at least in a big-step sense, it becomes redundant to have m repeated in the operational semantics on both sides of $\rightsquigarrow$. It is equally redundant in Hoare triples to repeat facts about the m-component in both pre- and postcondition.

The solution in big-step semantics is to separate the state σ that may change after running a command from the state m that may not. A big-step operational semantics is then a relation of the form $\sigma, c \rightsquigarrow_m \sigma'$, and a similar relation could be derived from a small-step semantics. The Hoare triple could be extended analogously, yielding a quadruple; e.g.,

$$\begin{aligned} m \Vdash \{P\}\ c\ \{Q\}_6 \triangleq \forall \sigma \in P.\ &\neg(\sigma, c \rightsquigarrow \mathsf{fail}) \wedge \\ &\forall \sigma'.\ \sigma, c \rightsquigarrow_m \sigma' \Rightarrow \sigma' \in Q \end{aligned}$$

This definition is not very practical, though: it requires all quadruples to carry the m-parameter even though it is only relevant when c is a procedure call.

The solution is to define a logic *spec* of specifications [Rey82] in which, intuitively, the truth value of a formula is measured by how many m it holds for. The Hoare triple is a formula in this logic, defined along the lines of

$$\{P\}\ c\ \{Q\}_7 \triangleq \{m \mid m \Vdash \{P\}\ c\ \{Q\}_6\}$$

The triple now appears to have only three parameters – the m-parameter has been *hidden* just like the heap is hidden in the assertion logic. Defining

$$\mathit{spec} \triangleq \mathcal{P}_{\leq}(M)$$

for an appropriate preorder, *spec* is a complete Heyting algebra, defined with a Kripke model as in Proposition 3.1 (page 12).

It remains to choose the preorder. Since separation logic emphasises local and modular reasoning, it is beneficial to let the preorder be the extension ordering $\sqsubseteq$ on M. This ensures that any specification that holds for procedure map m also holds in any $m' \sqsupseteq m$, thus enabling **local reasoning for procedures** just as the frame rule enables it for heaps.

We must then show closure under $\sqsubseteq$ for all atomic formulas in *spec*, while the Kripke construction guarantees it for the logical connectives. If the big-step relation is closed under extension of the m-component, then we have $\{P\}\ c\ \{Q\}_7 \in \mathcal{P}_{\sqsubseteq}(M)$ as desired. Otherwise, a technique similar to that used in $\{P\}\ c\ \{Q\}_4$ applies, and we can define a $\{P\}\ c\ \{Q\}_8 \in \mathcal{P}_{\sqsubseteq}(M)$:

$$\{P\}\ c\ \{Q\}_8 \triangleq \{m \mid \forall m' \sqsupseteq m.\ m' \Vdash \{P\}\ c\ \{Q\}_6\}$$

We can additionally define a primitive specification $f(\bar{x}) \mapsto c$ to say that a procedure f has parameters $\bar{x}$ and body c. Despite the similarity to points-to for heaps, no one has yet found a good reason to separate specifications with $*$! With these ingredients, we can define a formula in $\mathcal{P}_{\sqsubseteq}(M)$ to assert that f has specification (P, Q):

$$f(\bar{x}) \mapsto \{P\}\{Q\} \triangleq \exists c.\ f(\bar{x}) \mapsto c \wedge \{P\}\ c\ \{Q\}$$

See [BJSB11, JBK13] for details and variations on the this formula.

Note that if local reasoning for procedures is *not* desired, and the procedure map is static and global, then a much simpler technique applies: make the whole theory parametric in this map [Nip02, PB05, BJSB11]. This is the most common approach, but it results in logics that are not formally modular because there is formally a different theory of program verification for each program! That is, *given* a program fragment (with its procedure map), one *obtains* a theory in which to verify this one fragment. When multiple fragments have been verified this way, each in their own theory, there is no explicit theorem that tells us that the specifications of all the fragments are guaranteed valid in the theory obtained for the composite program.

4.2.2 Other examples

Above, we saw just one example of what a specification logic can do. A rule of thumb is that any state that remains unchanged across commands belongs in the specification logic, and any other state belongs in the assertion logic.

The specification logics considered here are complete Heyting algebras, constructed using the techniques in Section 3.1.3.

Below are some additional applications of specification logic. The various ingredients in the model of specifications tend not to interfere, so a full-featured specification logic can be modelled as $\mathcal{P}_{\leq_1,\ldots,\leq_n}(T_1 \times \cdots \times T_n)$, generalising from the descriptions of $\mathcal{P}_{\leq_i}(T_i)$ below.

Aliasing in call-by-name. The original specification logic by Reynolds [Rey82] was used to describe procedure mappings, much as we saw in Section 4.2.1, but also to assert non-aliasing between procedure parameters in the call-by-name setting of ALGOL 60. Unlike heap aliasing, which can be changed by commands, parameter aliasing stays fixed throughout a scope and is thus a good candidate for describing in the specification logic.

Immutable variables. Languages such as C and Java allow declaring certain variables as *constant* or *immutable*. In ML-like languages, all variables are immutable. All such variables could be described in the specification logic rather than the assertion logic. I have not seen this done in practice, though.

Recursion. To aid in verifying recursive procedures, it has proved useful to add natural numbers to the specification logic to count either the depth of recursion [vO99, Nip02] or the number of execution steps remaining [AM01, AMRV07, BJSB11]. This is known as **step indexing**. In both cases, specifications are downwards-closed sets of natural numbers: $\mathit{spec} = \mathcal{P}_{\geq}(\mathbb{N})$. Intuitively, the truth value of a specification measures how many steps of execution (or depth of procedure calls) the specification will hold for; a valid specification holds for any number of steps (or any depth of procedure calls).

This model enables the definition of a **later-operator** [Nak00, AMRV07, DAB09, DAH08, JBK13] on specifications:

$$\triangleright S \triangleq \{k \mid \forall k' < k.\ k' \in S\}$$

Intuitively, $\triangleright S$ means that S will hold after one step of execution (or for recursive calls one level deeper). The rule for procedure calls then requires only $\triangleright(f(\bar{x}) \mapsto \{P\}\{Q\})$ in its assumptions, and we can get this assumption by applying the **Löb rule** with $S = f(\bar{x}) \mapsto \{P\}\{Q\}$:

$$\frac{\triangleright S \vdash S}{\vdash S}\ \textsc{Löb}$$

Counting recursion depth clearly belongs in the specification logic because the depth cannot have changed after executing some c. On the other hand, counting steps of program execution can also be done usefully in the assertions, which allows for a $\triangleright$-operator in the assertion logic [BJSB11].

Recursive specifications. A specification can be defined recursively just like any other predicate, but this will always be subject to well-formedness restrictions, typically requiring the recursive occurrence to be in a **positive position** in the formula, or require a **well-founded** term to become smaller with each self-application. For instance, the existence of a specification S satisfying $S \equiv S \Rightarrow \bot$ would render the logic inconsistent[9].

Having step-indexes in the specifications, as sketched above, allows recursive definitions with a third type of well-formedness restriction: *contractiveness* [AM01, DAH08, BRS+11]. This allows definitions in which the recursive occurrence can be anywhere, as long as it is syntactically under a $\triangleright$-operator. This allows defining an S such that $S \equiv (\triangleright S) \Rightarrow \bot$. This was a silly example, but useful examples can be found in [DAH08, BRS+11].

A powerful alternative to this would be to use a metalogic that has a $\triangleright$-operator and then lift this into the separation logic. For an example, see the work on impredicative CAP [SB13a], which uses the *topos of trees* [BMSS12] as its metalogic.

Frames. Consider a small variation on $\{P\}\ c\ \{Q\}_4$:

$$\{P\}\ c\ \{Q\}_9 \triangleq \{R \mid \{P * R\}\ c\ \{Q * R\}_1\}$$

Then specifications are predicates on assertions, and the validity of a triple is intuitively measured by how many assertions can be framed on to it. This immediately gives us a **frame operator** [BTSY05, BTSY06, Kri12, JBK13], traditionally written $\otimes$, defined as

$$S \otimes R \triangleq \{P \mid (P * R) \in S\}$$

It satisfies the following identity, which allows us to write more concise specifications that do not repeat assertions between pre- and postcondition:

$$\{P\}\ c\ \{Q\}_9 \otimes R \equiv \{P * R\}\ c\ \{Q * R\}_9$$

As in the procedure-map example above, when we make *spec* a Kripke model, we can extend any useful closure property from atomic specifications to the full logic. Here, we expect that the frame rule holds for triples; i.e.,

$$\{P\}\ c\ \{Q\}_9 \vdash \{P\}\ c\ \{Q\}_9 \otimes R$$

[9] Exercise! First prove $S \vdash \bot$, which proves $\vdash S$, and together they prove $\vdash \bot$.

If so, we can extend the frame rule to all specifications by defining *spec* $= \mathcal{P}_{\sqsubseteq}(asn)$, where $\sqsubseteq$ is the extension ordering on the monoid $(asn, *, emp)$. This gives the following inference rule, called the **higher-order frame rule**.

$$\frac{}{S \vdash S \otimes R}$$

Under certain conditions, this rule allows framing invariants onto triples in negative positions of an entailment, whereas the ordinary frame rule only allows it for positive positions. Examples of its utility as a second-order frame rule are found in [OYR04, BTSY06, JBK13]; an example of using it as a third-order frame rule is in [BTSY06].

4.2.3 Structural rules in specification logic

The structural rules discussed in Section 4.1.2 can now typically[10] be entailments in the specification logic rather than the metalogic. For example, the existential rule becomes

$$\frac{}{\forall x.\ \{P(x)\}\ c\ \{Q\} \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists}$$

To get a presentation that looks more standard, it is customary to quantify over all specifications S and instead print the rule as

$$\frac{S \vdash \forall x.\ \{P(x)\}\ c\ \{Q\}}{S \vdash \{\exists x.\ P(x)\}\ c\ \{Q\}}\ \textsc{Exists}$$

If specifications can be usefully embedded in assertions, then the rule of consequence can also make use of this S; see the consequence rule in [PB08] for an example.

4.3 Alternative formulations

An assertion logic, as developed in Section 3, can also serve as ingredient in other theories than the specification logics defined above. A brief overview is given here.

4.3.1 Rigid specification logics

The specification logics described above are complete Heyting algebras and thus full higher-order logics. Less expressive and more disciplined logics have also been proposed. In particular, the logic of Parkinson and Bierman [PB08], extended by van Staden and Calcagno [vSC10], stands out as a specification logic that is expressive enough to specify most object-oriented code but requires specifications to follow a rigid structure that is essentially a mirror image of the class structure of an object-oriented program.

[10] I know of one exception to this: the higher-order frame rule in [SBRY11].

Several challenging specifications have been expressed in this system [PB08, DP08]. On the other hand, the extensions made in [vSC10] certainly extended the range of useful specifications that could be expressed even though the original system perhaps seemed powerful enough at first glance. It is likely that a third case study would expose the need for further extensions, and so on. Essentially, these rigid logics need almost all the features of a complete Heyting algebra: *auxiliary variables* are universal quantifiers, *abstract predicates* are existential quantifiers, *specification refinement* is entailment, *specification combination* is conjunction, and so on.

When the specification logic is a complete Heyting algebra from the beginning, there is more freedom to use it in new ways without requiring extensions. Rigid specification logics can then be built on top as needed, hopefully reducing the burden of the soundness proof for these.

Compare [KAB+09, BJSB11], where programs are specified in a separation logic in which both specifications and assertions are higher-order logics. The drawback is that specifications can be hard to understand when their structure does not follow a known pattern. They can be more or less verbose compared to a specification in a rigid framework, depending on whether that framework is a good fit for the code at hand.

Rigid logics seem to be a good fit to model stand-alone verification tools with extensive automation [vSC10, DP08, JSP12]. Full higher-order logics tend to be embedded in proof assistants such as Coq and HOL, where automation is guaranteed to be sound but runs orders of magnitude slower [CMM+09, Chl11, Chl13, McC09, Tue09, MSBS12, BJB12, JBK13].

4.3.2 Type systems

It is possible to formulate a separation logic as a type system. Type inference will of course be undecidable, and type checking will require annotations corresponding to proofs in a program logic.

In Hoare type theory [NMB08, NAMB07, PBNM08], a computation has a monadic type, similar to the IO monad in Haskell but with pre- and postcondition annotations. Ignoring variable contexts, the typing judgement $c : \{P\}\ x{:}A\ \{Q\}$ means that computation c has precondition P and returns a value of type A, bound as x in postcondition Q. Essentially, their types correspond to our *specifications*. Pre- and postconditions are predicates on the heap, almost exactly as we defined them in Section 3.

In the type system of Pottier [Pot08, SBP+12], as well as Krishnaswami et al. [KTDG12], their types essentially correspond to our *assertions*. Like in ML, a computation is a function, and the semantics has to be call-by-value to serialise side effects predictably. A function that writes to a heap cell has a dependent type along the lines of $\Pi l.\ cap(l) \times val \to cap(l)$, where $cap(l)$ is an abstract capability to access location l. The capability and the function space are **linear**, and the capability is therefore returned again by the function; otherwise, it would be lost to the caller. It is understood that capability tokens will be compiled away, but they do have a representation in the term language. Capability types can be composed with separating conjunction, which makes these systems behave much like separation logic.

One thing to beware of when building a separation-logic type system is the handling of logical variables. Consider for instance a procedure that increments the value at a given heap location, specified in the style of Section 4.2.1:

$$\forall i.\ \mathsf{inc}(\mathsf{x}) \mapsto \{\mathsf{x} \mapsto i\}\{\mathsf{x} \mapsto i+1\}$$

There is a clear distinction here between x, which has a run-time representation, and i, which exists purely in the specification. If the arrow and dependent-product types denote function spaces, as in the example with capabilities above, then there must be a different mechanism for quantifying over a logical variable. Nanevski et al. proposed "binary postconditions" [NVB10, NMS$^+$08] for addressing this in Hoare type theory, but their solution restricts the scope of logical variables to a single triple. Another branch of Hoare type theory [CMM$^+$09] proposed explicitly marking logical variables as such, but this required extending Coq with an axiom whose soundness has not been formally established. The type system in [BTSY06] does not include logical variables, and thus it cannot specify inc. The original system of Pottier [Pot08] had the same problem, but this was later addressed [PP11] by adding logical universal quantifiers and singleton types, which is also how [KTDG12] handles the problem.

5 Conclusion

At the time of writing this text, the ACM Digital Library lists 147 publications with the keyword "separation logic". It is well known that these have a lot in common underneath their cosmetic differences. Unfortunately, those commonalities are typically treated as *design patterns* to draw inspiration from when building a separation-logic theory from scratch, rather than a formally *reusable theory* that can be built upon.

In this text, I have presented a core of standard definitions and theorems in the hope that future texts on separation logic can take these for granted rather than recreate them. Those definitions lead to expressive higher-order logics without adding additional complexity over the first-order case.

In particular, a typical assertion logic should arise as the powerset of a separation algebra, closed under a preorder. The interesting contribution of future theories should be the choice of separation algebra and preorder, while turning this into a logic is standard. Similarly, specification logics are made easy since they arise from standard Kripke models, which automatically contain all the operators and quantifiers needed for abstract and modular specifications. Finally, simple but fully formal treatment of program variables can be achieved using open terms, although this is not as widely applicable as the other theories mentioned.

At the same time, we have discussed the aspects that still remain patterns and cosmetics. Variations in the operational semantics and Hoare triple are necessarily language-specific, and there is rarely any formal reuse between theories, but there are still many design patterns to be borrowed.

Much more ought to be said about concurrent separation logic and models that use guarded recursion, but these areas are still very much in flux. Hopefully, general and reusable theories will eventually emerge from those lines of research.

Acknowledgements. I would like to thank the proof readers – Jesper Bengtson, Lars Birkedal, Aleš Bizjak and Marco Paviotti – for helpful feedback and discussions.

Index

References

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C Minor. In *Proceedings of TPHOLs*, 2007.

[AM01] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 2001.

[AM13] Reynald Affeldt and Nicolas Marti. Towards formal verification of TLS network packet processing written in C. In *Proceedings of PLPV*, 2013.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of POPL*, 2007.

[App06] Andrew W. Appel. Tactics for separation logic, Draft of January 2006. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`.

[Atk10] Robert Atkey. Amortised resource analysis with separation logic. *Programming Languages and Systems*, pages 85–103, 2010.

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Proceedings of ESOP*, 2005.

[BC10] James Brotherston and Cristiano Calcagno. Classical BI: Its semantics and proof theory. *Logical Methods in Computer Science*, 6(3), 2010.

[BCOP05] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, 2005.

[BJ] Jesper Bengtson and Jonas B. Jensen. Charge! development version. `https://github.com/jesper-bengtson/Charge`.

[BJB12] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *Proceedings of ITP*, 2012.

[BJSB11] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *Proceedings of ITP*, 2011.

[BK10] J. Brotherston and M. Kanovich. Undecidability of propositional separation logic and its neighbours. In *Proceedings of LICS*, 2010.

[BMSS12] L. Birkedal, R. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), October 2012.

[Boy03] John Boyland. Checking interference with fractional permissions. In *Proceedings of SAS*, 2003.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270, 2007.

[BRS$^+$11] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. In *Proceedings of POPL*, 2011.

[BRSY08] L. Birkedal, B. Reus, J. Schwinghammer, and H. Yang. A simple model of separation logic for higher-order store. In *Proceedings of ICALP*, 2008.

[BTSY05] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of LICS*, 2005.

[BTSY06] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2(5:1), August 2006.

[BV13] James Brotherston and Jules Villard. Parametric completeness for separation theories, Submitted, 2013.

[CGZ05] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *Proceedings of POPL*, 2005.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of PLDI*, 2011.

[Chl13] Adam Chlipala. The Bedrock structured programming system. In *Proceedings of ICFP*, 2013.

[CMM$^+$09] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of ICFP*, 2009.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of LICS*, 2007.

[CSV07] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of PLDI*, 2007.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proceedings of LICS*, 2009.

[DAH08] Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multimodal separation logic for reasoning about operational semantics. *Electronic Notes in Theoretical Computer Science*, 218:5–20, 2008.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proceedings of APLAS*, 2009.

[DP08] Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *Proceedings of OOPSLA*, 2008.

[DYBG$^+$13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of POPL*, 2013.

[DYDG$^+$10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of ECOOP*, 2010.

[DYGW10] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. In *Proceedings of VSTTE*, 2010.

[DYGW11] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning, February 2011. Journal submission.

[FFS10] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. Parameterized memory models and concurrent separation logic. In *Proceedings of ESOP*, 2010.

[GBC$^+$07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings of APLAS*, 2007.

[GBC11] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. In *Proceedings of MFPS*, 2011.

[GLW06] Didier Galmiche and Dominique Larchey-Wendling. Expressivity properties of boolean BI through relational models. In *Proceedings of FSTTCS*, 2006.

[GMP02] D. Galmiche, D. Méry, and D. Pym. Resource tableaux (extended abstract). In *Proceedings of CSL*, 2002.

[GMP05] D. Galmiche, D. Mery, and D. Pym. Semantics of BI and resource tableaux. *Mathematical Structures in Computer Science*, 15(6):1033–1088, 2005.

[GMS12] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a program logic for javascript. In *Proceedings of POPL*, 2012.

[HDV11] C-K Hur, Derek Dreyer, and Viktor Vafeiadis. Separation logic in the presence of garbage collection. In *Proceedings of LICS*, 2011.

[Hob11] Aquinas Hobor. Improving the compositionality of separation algebras, July 2011. Unpublished draft.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, 2001.

[JB11] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic: Appendix, 2011. `http://itu.dk/~jobr/research/fsl-appendix.pdf`.

[JB12] Jonas B. Jensen and Lars Birkedal. Fictional separation logic. In *Proceedings of ESOP*, 2012.

[JBK13] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In *Proceedings of POPL*, 2013.

[JSP12] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial, December 2012.

[KAB$^+$09] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *In Proceedings of TLDI*, 2009.

[KBJD13] Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. Coq: The world's best macro assembler? In *Proceedings of PPDP*, 2013.

[Kri12] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2012.

[KTDG12] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In *Proceedings of ICFP*, 2012.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[LWN13] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of POPL*, 2013.

[McC09] Andrew McCreight. Practical tactics for separation logic. In *Proceedings of TPHOLs*, 2009.

[MG07] M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proceedings of TACAS*, 2007.

[MSBS12] Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In *Proceedings of VSTTE*, 2012.

[Myr10] M. O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of POPL*, 2010.

[Nak00] Hiroshi Nakano. A modality for recursion. In *Proceedings of LICS*, 2000.

[NAMB07] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *In Proceedings of ESOP*, 2007.

[Nip02] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Proceedings of CSL*, 2002.

[NMB08] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5–6):865–911, 2008.

[NMS$^+$08] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP*, 2008.

[NS06] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proceedings of POPL*, 2006.

[NVB10] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of POPL*, 2010.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[O'H12] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). *NATO Science for Peace and Security Series D*, 33:286–318, 2012.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[OYR04] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, 2004.

[Par05] Matthew Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, November 2005.

[Par10] Matthew Parkinson. The next 700 separation logics. In *Proceedings of VSTTE*, 2010.

[PB05] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, 2008.

[PBC06] M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logic. In *Proceedings of LICS*, 2006.

[PBNM08] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model of impredicative Hoare type theory. In *Proceedings of ESOP*, 2008.

[Pit01] A. M. Pitts. Nominal logic, a first order theory of names and binding. In *Proceedings of TACS*, 2001.

[Pot08] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *Proceedings of LICS*, 2008.

[Pot13] François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, January 2013.

[POY04] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of bi. *Theoretical Computer Science*, 315(1):257—-305, May 2004.

[PP11] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Proceedings of TLDI*, 2011.

[PS12] Matthew Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 2012.

[Pym02] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[Rey82] John C. Reynolds. Idealized Algol and its specification logic. *Tools and notions for program construction*, pages 121–161, 1982.

[Rey00] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial Perspectives in Computer Science*, pages 303—321, 2000.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.

[Rey11] John C. Reynolds. Introduction to separation logic, 2011. Course notes for 15-818A3 at Carnegie Mellon University.

[RS06] Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In *Proceedings of CSL*, 2006.

[SB13a] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates, 2013. Submitted for publication.

[SB13b] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates (technical appendix), 2013.

[SBP09] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying generics and delegates (technical appendix). Technical report, IT University of Copenhagen, 2009. Available at `http://itu.dk/~kasv/generics-delegates-tr.pdf` or as part of Kasper Svendsen's PhD thesis.

[SBP10] K. Svendsen, L. Birkedal, and M.J. Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, 2010.

[SBP$^+$12] J. Schwinghammer, L. Birkedal, F. Pottier, B. Reus, K. Støvring, and H. Yang. A step-indexed Kripke model of hidden state. *Mathematical Structures in Computer Science*, 2012.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation for concurrent data structures. In *Proceedings of ESOP*, 2013.

[SBRY11] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3:21), July 2011.

[SJP10] Jan Smans, Bart Jacobs, and Frank Piessens. Heap-dependent expressions in separation logic. In *Proceedings of FMOODS*, 2010.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Proceedings of POPL*, 2007.

[TSFC09] Gang Tan, Zhong Shao, Xinyu Feng, and Hongxu Cai. Weak updates and separation logic. In *Proceedings of APLAS*, 2009.

[Tue09] Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Proceedings of TPHOLs*, 2009.

[Vaf11] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011.

[VB08] C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. *Electr. Notes Theor. Comput. Sci.*, 218:371–389, 2008.

[VN13] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of OOPSLA*, 2013.

[vO99] David von Oheimb. Hoare logic for mutual recursion and local variables. 1999.

[vSC10] Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with MultiStar. In *Proceedings of OOPSLA*, 2010.

[WB11] Ian Wehrman and Josh Berdine. A proposal for weak-memory local reasoning, Presented at the LOLA workshop, 2011.

[WDP13] John Wickerson, Mike Dodds, and Matthew Parkinson. Ribbon proofs for separation logic. In *Proceedings of ESOP*, 2013.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proceedings of TPHOLs*, 2004.

[YO02] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FoSSaCS*, 2002.