New Foundations for Separation Logic

Hiep, H.A.

Chapter 4

Reynolds' logic

In Chapter B, we recall Hoare's logic for reasoning about **while**-programs and recursive programs. In this chapter, we investigate Reynolds' logic, an extension of Hoare's logic for reasoning about pointer programs. We introduce a novel semantics and several original proof systems for Reynolds' logic, all interpreted with respect to the same semantics: it is possible to reinterpret the original proof systems in the novel semantics. We then introduce an extension to dynamic logic, called dynamic separation logic, and show how to simplify program modalities. This yields the discovery of an alternative proof system of Reynolds' logic.

What is remarkable is that all the proof systems in this chapter are *equivalent* in the following sense: the original proof system proposed by Reynolds and all other proof systems in this chapter have *exactly the same set of provable objects*! This justifies us to call all these proof systems which have a single common semantics 'Reynolds' logic', in honor of J.C. Reynolds. But, why an alternative proof system for Reynolds' logic? It would be fair to say, that the alternative proof systems sheds light on the same matter from a different angle. In fact, what distinguishes the new proof systems (both the weakest precondition calculus and the strongest postcondition calculus) from the original ones is the property of gracefulness. Gracefulness means that the weakest precondition of any statement and a first-order formula remains a first-order formula, and similar for the strongest postcondition. Furthermore, by the techniques developed in this chapter, we are able to fill in a missing gap of proving that the global axioms can be derived from the local axioms and the frame rule without using the *magic wand*, the connective of separating implication.

Reynolds' logic can be seen as an extension of Hoare's logic in two ways:

- In Hoare's logic the programming language is based on a first-order program signature, but in Reynolds' logic the programming language is based on a pointer program signature. Moreover, the proof objects of the two proof systems are specifications $\{\phi\}\ S\ \{\psi\}$. In Hoare's logic the formulas ϕ and ψ are first-order formulas, whereas in Reynolds' logic these formulas are further extended to the formulas of separation logic.

- Since every formula of first-order logic is *also* a formula of separation logic, all the instances of the axioms and rules of Hoare's logic can also be considered part of Reynolds' logic.

However, the second point raises the question: how much rules of Hoare's logic remain sound also when extending the instances to formulas of separation logic? If we consider the invariance rule

$$\frac{\{\phi\}\ S\ \{\psi\}}{\{\phi \wedge \chi\}\ S\ \{\psi \wedge \chi\}}$$

of Hoare's logic, where the free variables of χ are not changed by the statement S, then clearly we cannot extend this rule to arbitrary formulas of separation logic: the rule would become unsound. The problem here lies in the fact that the program S may change a location on the heap, but it is not easily recognized from the syntax of the program what location has changed. This is in contrast to the invariance rule, where an approximation of the effects of a statement in Hoare's logic is captured by $access(S)$ and $change(S)$.

Thus, an important proof rule that is novel in Reynolds' logic, that is not part of Hoare's logic, is the so-called *frame rule*

$$\frac{\{\phi\}\ S\ \{\psi\}}{\{\phi * \chi\}\ S\ \{\psi * \chi\}}$$

where the free variables of χ are not changed by the statement S. Note that in this rule we use separating conjunction instead of logical conjunction. However, it turns out that the soundness of the frame rule is quite delicate [232]. In this chapter we shall revisit the soundness proof of the frame rule, and give an alternative proof that is more proof theoretic in nature.

As one can recall in Chapter B, due to the compositional nature of the semantics of programs, it suffices here to restrict our attention to the base case, the pointer manipulating operations, and not to the control structures of sequential composition, **if** and **while**-statements, or recursive procedures, since these latter constructs of the programming language are orthogonal to our current concerns. In fact, already in [125], Ishtiaq and O'Hearn recognize that the novelty of Reynolds' logic lies in the treatment of the basic operations:

> "We will not give a full syntax of [statements], as the treatment of conditionals and looping statements is standard. Instead, we will concentrate on assignment statements, which is where the main novelty of the approach lies."

It is the case that our previous discussion concerning the compositional nature of program semantics naturally transfers to the context of pointer programs.

First, we introduce the proper semantic basis in which we can interpret the proof objects of Reynolds' logic. Although the semantics is a generalization of the standard semantics of separation logic, it is quite remarkable to observe that the

original axiomatization due to Reynolds is still sound with respect to this general semantics.

Secondly, we present Reynolds' logic in its usual way. There are four ways in which the proof system of Reynolds' logic can be presented: a *local* axiomatization, and a *global* axiomatization, a *backwards* weakest precondition axiomatization, and a *forwards* strongest postcondition axiomatization. In the first proof system with the *local* axiomatization, however, the frame rule plays a crucial role. One can recover the *global* axiomatization from the *local* axiomatization by using the frame rule. Our only contribution here, is that we shall argue there is an alternative way of proving the soundness of the frame rule, from a proof theoretical point of view, rather than the 'surprisingly delicate' semantic point of view as done by Yang and O'Hearn [232].

Third, we show the soundness and (relative) completeness of a novel weakest precondition axiomatization, by introducing an extension to separation logic, called *dynamic separation logic*, in which a logical modality is added that captures the weakest precondition. We introduce pseudo-operations corresponding to *heap update* and *heap clear*. It is crucial that these pseudo-operations are *not* part of the programming language, and for both of them we show they satisfy a useful meta-property that we capture by proving corresponding *substitution lemmas*. That these pseudo-operations are not part of the programming language is crucial, since the frame rule is not sound with respect to them. However, by introducing the pseudo-operations on the logical level, and thereby not allowing the application of the frame rule over them, we obtain a way of expressing the weakest precondition of all other basic instructions in terms of these pseudo-instructions. In fact, the pseudo-instructions satisfy a property called *gracefulness*, in the sense that computing the weakest precondition of a pseudo-instruction with respect to a first-order postcondition results in a first-order weakest precondition. Even stronger, computing the weakest precondition of a pseudo-instruction with respect to the fragment of separation logic without magic wand also results in a formula without magic wand: thus eliminating magic wand from the generated weakest precondition. It turns out that all the previous axiomatizations of Reynolds' logic lack this property of gracefulness.

Fourth, we recall two strongest postcondition axiomatizations of Reynolds' logic, again a *local* strongest postcondition axiomatization and a *global* strongest postcondition axiomatization. We then also introduce a novel strongest postcondition axiomatization by again using our previously introduced pseudo-instructions, and we compare our alternative axiomatization with the two existing calculi.

In this chapter, we focus on classical separation logic as assertion language. In Appendix C, our approach is also extended to intuitionistic separation logic, also resulting in novel weakest preconditions and strongest postconditions. This shows that our approach, of introducing pseudo-instructions, is robust under different interpretations.

4.1 General semantics and memory models

In the Sections 2.2 and 2.3 we have established that both the standard semantics and the full semantics of separation logic are not compact, leading to the impossibility to have a complete finitary proof system. In this section we investigate two more general semantics for separation logic. We introduce *general heap structures*, which extends structures to include a set of heaps over which quantification in the semantics of the separating connectives ranges. This approach is similar to the general structures of Henkin semantics [108]. Although this semantics is sufficient for interpreting separation logic formula, the set of heaps does not necessarily contain all heaps needed to reason about memory modification effects of pointer programs. Therefore, we also introduce the *heap semantics*, in which we extend structures with a particular set of heaps called a *memory model*. We further show the relation between the semantics based on memory models and the standard semantics, full semantics, and novel semantics of separation logic for which we obtained a sound and complete, finitary proof system in Section 3.2 and Section 3.4.

Given a structure $\mathfrak{A} = (A, \mathcal{I})$ with domain A, and let H be a set of heaps (partial functions from A to A) with $h, h_1, h_2 \in H$. Recall that we can express the partitioning of heaps by $h \equiv h_1 \uplus h_2$ which satisfies the following properties:

$$h \equiv h_1 \uplus h_2 \Longrightarrow \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset, \tag{4.1}$$

$$h \equiv h_1 \uplus h_2 \Longrightarrow h(a) = \begin{cases} h_1(a) & \text{if } a \in \mathrm{dom}(h_1), \\ h_2(a) & \text{if } a \in \mathrm{dom}(h_2), \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{4.2}$$

A *general heap structure* (for separation logic) can be used to give semantics to separating connectives, where only the heaps in a given set of heaps are considered. Our approach here is similar to Henkin's general heap structures for higher-order logic, where quantification over values of higher-order arities is restricted to range over a given set of values.

Definition 4.1.1 (General heap structures)**.** A *general heap structure* $\mathfrak{H} = (\mathfrak{A}, H)$ consists of a structure $\mathfrak{A} = (A, \mathcal{I})$ with domain A and interpretation $\mathcal{I}$ and a set of heaps H of partial functions from A to A.

A general heap structure thus includes a set of possible heaps, and in the general semantics one limits quantification in the semantics of the separating conjunction and separating implication to the given set of heaps. The satisfaction relation over general heap structures can now be given. It is similar to Definition 2.2.1, except for the following clauses that are relative to the given set of heaps.

Definition 4.1.2 (Satisfaction relation)**.** Given a general heap structure $\mathfrak{H} = (\mathfrak{A}, H)$, a valuation ρ of $\mathfrak{A}$, a heap h of H, and a separation logic formula ϕ. The satisfaction relation $\mathfrak{H}, h, \rho \models \phi$ is defined inductively on the structure of ϕ:

- ...

- $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi * \psi$ iff $\mathfrak{H}, h_1, \rho \models \phi$ and $\mathfrak{H}, h_2, \rho \models \psi$ for some $h_1, h_2 \in H$ such that $h \equiv h_1 \uplus h_2$,
- $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi \mathrel{-\!\!*} \psi$ iff $\mathfrak{A}, h', \rho \models \phi$ implies $\mathfrak{A}, h'', \rho \models \psi$ for every $h', h'' \in H$ such that $h'' \equiv h \uplus h'$,
- ...

The superscript **GSL** on $\models$ stands for General Separation Logic. Note that in general heap structures $\mathfrak{H} = (\mathfrak{A}, H)$ we have that H can be empty, similar to the situation of an empty domain in the classical semantics. However, in the context of the satisfaction relation, we may assume the set of heaps is non-empty since h is a heap in H. This situation is similar to the situation in classical logic where the domain must be non-empty, since a valuation ρ assigns values to variables.

Similar to before, we have the coincidence condition and invariance under renaming. Both propositions are with respect to a fixed heap in our memory model.

Proposition 4.1.3 (Coincidence condition)**.** *Given that $\rho[FV(\phi)] = \rho'[FV(\phi)]$, it follows that $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi$ if and only if $\mathfrak{H}, h, \rho' \models^{\mathbf{GSL}} \phi$.*

Proposition 4.1.4 (Invariance under renaming)**.** *Given a renaming π such that all free variables of ϕ stay the same, i.e. $\pi(v) = v$ for all $v \in FV(\phi)$. It follows that $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi$ if and only if $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \pi(\phi)$.*

Definition 4.1.5 (Denotation)**.** The *denotation* of a formula $\mathfrak{H}[\![\phi]\!]^{\mathbf{GSL}}$ is defined:

$$\mathfrak{H}[\![\phi]\!]^{\mathbf{GSL}} = \{(h, \rho) \mid \mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi\}.$$

We write $\mathfrak{H}, h \models^{\mathbf{GSL}} \phi$ to mean $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi$ for all valuations ρ of $\mathfrak{A}$, and we write $\mathfrak{H} \models^{\mathbf{GSL}} \phi$ to mean $\mathfrak{A}, h \models^{\mathbf{GSL}} \phi$ for all heaps $h \in H$. Given a sentence that is satisfied, using the coincidence condition we can obtain that it is also satisfied by the same general heap structure but with any other valuation: the valuation has no influence on whether a sentence is satisfied by the structure, but the heap does have such influence (as it does with standard and full semantics). So if ϕ is a sentence, $\mathfrak{H}, h \models^{\mathbf{GSL}} \phi$ if and only if $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} \phi$ for some valuation ρ.

Given a sentence ϕ, we write $\models^{\mathbf{GSL}} \phi$ to mean that $\mathfrak{H} \models^{\mathbf{GSL}} \phi$ for all gneral structures $\mathfrak{H}$, and we then say that ϕ is *valid*. Valid sentences in general separation logic thus are properties that hold for all general heap structures.

An interesting instance is the general heap structure $(\mathfrak{A}, \{\epsilon\})$ where ϵ is the partial function that is never defined anywhere. This general heap structure clearly satisfies the formula **emp**. Further, since we can split the empty heap only in two empty heaps, we have that separating conjunction is equivalent to logical conjunction. For a similar reason we have that separating implication is equivalent to logical implication. Thus, any valid classical formula in which we replace (some) logical conjunction and implication connectives by separating conjunction and implication, respectively, is a valid separation logic formula with respect to the general semantics. Note that this works since classical formulas do not have any occurrence of a points-to relation, since the points-to relation is only available to separation logic and not present in the signature.

A well-known example of a valid formula in the general semantics is

$$\models^{\mathbf{GSL}} (\phi * (\phi \mathrel{-\!\!*} \psi)) \to \psi$$

where ϕ and ψ are arbitrary separation logic formulas. It is easily verified that this formula is indeed valid: *if* a split is possible, then we can always recombine the two separate parts of the heap by the separating implication to obtain the succedent of the separating implication.

However, in the general semantics, there are formulas which are not valid, but which are valid in both the standard and the full semantics of separation logic. Take the formulas $(x \hookrightarrow y)$ and $(x \mapsto y) * \mathbf{true}$. These formulas are equivalent in both **WSL** and **FSL**. While it is the case that

$$\models^{\mathbf{GSL}} \forall x, y.\, ((x \mapsto y) * \mathbf{true}) \to (x \hookrightarrow y),$$

the converse does not hold. Take the general heap structure $\mathfrak{H} = (\mathfrak{A}, H)$ with the domain A of $\mathfrak{A}$ having at least two elements, and H being the set $\{\epsilon, h\}$ where $h(a) = a$ for some $a \in A$, and $h(b) = b$ for some other $b \in A$, and undefined on all other elements. Now we have $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} (x \hookrightarrow y)$ where ρ assigns x and y to a. However, it is not the case that $\mathfrak{H}, h, \rho \models^{\mathbf{GSL}} (x \mapsto y) * \mathbf{true}$, since H contains the heap h but not the two subheaps making the split possible.

Another counter-example is the following implication (for any formula ϕ):

$$(x \not\mapsto -) \wedge ((x \mapsto -) \mathrel{-\!\!*} ((x \mapsto -) * \phi)) \to \phi$$

which is valid in both **WSL** and **FSL** (in **WSL** we do not even need $(x \not\mapsto -)$). However, this fails for **GSL**. We now take $H = \{\epsilon\}$. Now, clearly, both a and b are not allocated in every heap. Take a valuation in which x has value a, and y has the value b. We could for example let ϕ express that y is allocated: $(y \hookrightarrow -)$. The antecedent of the implication is satisfied in the empty heap: x is not allocated, and for every heap in which x is allocated the succedent of the separating implication holds for the joined heap (vacuously). However, ϕ is not satisfied, since y is not allocated.

The counter-examples above demonstrates two issues with the general semantics. We expect certain closure conditions to hold for the set H to be able to naturally reason about the semantics of separation logic (stated informally):

- we expect that $\phi * \mathbf{emp}$ and ϕ are equivalent, so $\epsilon \in H$;
- we expect that $(x \mapsto y) * \mathbf{true}$ and $(x \hookrightarrow y)$ are equivalent, but for that to work we need that splitting off finite parts from any heap in H is also in H;
- under the assumption that there is some free space, that is $(x \not\mapsto -)$, we expect that $(x \mapsto -) \mathrel{-\!\!*} ((x \mapsto -) * \phi)$ and ϕ are equivalent, but for that to work every finite extension of any heap in H must be in H too.

We thus consider sets of heaps which satisfy certain closure conditions, called *memory models*. A memory model is a set of heaps that is closed under the operations of heap update and heap clear, and closed under heap existence conditions.

The combination of heap update and heap clear operations allows us to express finite splits and finite extension as required. For example, for any heap h where $a \notin \mathrm{dom}(h)$ we can apply the heap update $h[a := a']$ to obtain a larger heap, which can be split into the disjoint heaps h and $\epsilon[a := a']$, since $h[a := a'] \equiv h \uplus \epsilon[a := a']$. Similarly, if a heap h can be split so that $h \equiv h_1 \uplus \epsilon[a := a']$ then we know $a \notin \mathrm{dom}(h_1)$ and consequently that $h_1 = h[a := \bot]$.

Definition 4.1.6 (Memory model)**.** A *memory model* over A is a set H of heaps (partial functions from A to A) such that all the following conditions hold:

$$\epsilon \in H, \tag{4.3}$$

$$h[a := a'] \in H, \tag{4.4}$$

$$h[a := \bot] \in H, \tag{4.5}$$

$$\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset \Longrightarrow h' \equiv h_1 \uplus h_2 \text{ for some heap } h' \in H, \tag{4.6}$$

$$h_1 \subseteq h \Longrightarrow h \equiv h_1 \uplus h' \text{ for some heap } h' \in H, \tag{4.7}$$

for every $h, h_1, h_2 \in H$ and $a, a' \in A$.

A memory model which satisfies all conditions except Equation (4.7) is called a *classical memory model*. In fact, in the remainder of this chapter, all results also hold for classical memory models: only in Chapter C, where we introduce intuitionistic separation logic, it is necessary to consider memory models with the additional condition of Equation (4.7). However, for uniformity in presentation, we shall include the condition, even when it is strictly speaking not necessary.

It is not a problem to confuse a memory model and the set H of heaps, as long as we have that H is a memory model: this convention is not different from taking H as the carrier set that is closed under the given operations and conditions. There are many memory models, and we have a look at various ways of constructing them later in this section.

The above conditions require compatibility conditions on the heap partitioning relation. In particular, Equation (4.6) imposes the condition on the set H of heaps that any two disjoint heaps can be merged, and Equation (4.7) imposes the condition on H that if we know that heap h is an extension of heap h_1 then there must exists a heap h_2 which consists of the remaining location-value mappings. In fact, the last condition imposes that the empty heap must be in H by taking $h_1 = h$, thus Equation (4.3) is redundant.

Note that, in the case we work with the set of finite heaps, Equations (4.6) and (4.7) do not add much. For any two disjoint finite heaps h_1, h_2, we can always construct the heap which is their union $h_1 \uplus h_2$. This is easy to see, since any finite heap can simply be regarded as a finite construction from the empty heap and a finite association list of locations and values. The resulting finite heap simply zips the two association lists together. Similarly, for any finite heap h, and hence also a finite heap $h_1 \subseteq h$, we clearly can find a heap h' that is the remainder: we just look at the association list of locations and values that comprise h and filter out any of the locations that are in h_1. These constructions are rather obvious.

However, and this is crucial, in the case of infinite heaps these constructions are no longer straightforward. Equation (4.6) thus expresses that it must always be possible to merge heaps, even if one of the two heaps is infinite. And, furthermore, Equation (4.7) expresses that in case we have an infinite heap h and a (finite or infinite) heap h_1 we can always find the remainder of 'subtracting' h_1 from h, that is potentially infinite too.

Proposition 4.1.7. *If $h \equiv h_1 \uplus h_2$ and $g \equiv h_1 \uplus h_2$ then $h = g$.*

Proof. We have heap extensionality, $h = g$ if $h(a) = g(a)$ for all $a \in A$. But $h(a)$ is fixed by Equation (4.2), and so is $g(a)$. Our property follows from a case analysis of a: for either $a \in \mathrm{dom}(h_1)$, or $a \in \mathrm{dom}(h_2)$, or $a \notin \mathrm{dom}(h_1)$ and $a \notin \mathrm{dom}(h_2)$, $h(a) = g(a)$. The case $a \in \mathrm{dom}(h_1)$ and $a \in \mathrm{dom}(h_2)$ does not occur by Equation (4.1). □

Now we can use memory models to give semantics to the separating connectives, where we consider memory models instead of arbitrary non-empty sets of heaps.

Definition 4.1.8 (Memory structures)**.** A *memory structure* $\mathfrak{H} = (\mathfrak{A}, H)$ is a general heap structure consisting of a structure $\mathfrak{A} = (A, \mathcal{I})$ with domain A and interpretation $\mathcal{I}$, and as set of heaps a memory model H over A.

Since **MSL** can be seen as **GSL** but restricted to a certain class of structures, we also have the coincidence condition, invariance under renaming, and the denotation of a formula $\mathfrak{H}[\![\phi]\!]^{\mathbf{MSL}}$:

$$\mathfrak{H}[\![\phi]\!]^{\mathbf{MSL}} = \{(h, \rho) \mid \mathfrak{H}, h, \rho \models^{\mathbf{MSL}} \phi\}.$$

We shall use the superscript **MSL** to mean that the notions of satisfaction, validity, and denotation, previously defined for general heap structures, are restricted to memory structures.

Given a theory, i.e. a set of sentences Γ, we write $\mathfrak{H}, h \models^{\mathbf{MSL}} \Gamma$ to mean that all sentences in Γ are satisfied by all memory structures $\mathfrak{H} = (\mathfrak{A}, H)$ and heaps $h \in H$, that is, $\mathfrak{H}, h \models^{\mathbf{MSL}} \phi$ for all $\phi \in \Gamma$. We may also speak of 'Γ is satisfied by $\mathfrak{H}$ and h'. A theory Γ is *satisfiable* if there exists a memory structure $\mathfrak{H}$ and heap h such that $\mathfrak{H}, h \models \Gamma$. A theory Γ is *finitely satisfiable* if every finite subset of Γ is satisfiable.

Given a sentence ϕ, we write $\Gamma \models^{\mathbf{MSL}} \phi$ to mean $\mathfrak{H}, h \models^{\mathbf{MSL}} \phi$ for all memory structures $\mathfrak{H} = (\mathfrak{A}, H)$ and heaps $h \in H$ such that $\mathfrak{H}, h \models^{\mathbf{MSL}} \Gamma$, and say that ϕ is a *semantic consequence* of Γ. In case Γ is a context and ϕ a formula, then by $\Gamma \models^{\mathbf{MSL}} \phi$ we mean $\mathfrak{H}, h, \rho \models^{\mathbf{MSL}} \phi$ for all memory structures $\mathfrak{H} = (\mathfrak{A}, H)$, heaps $h \in H$, and valuations ρ of $\mathfrak{A}$ such that $\mathfrak{H}, h, \rho \models^{\mathbf{MSL}} \psi$ for all $\psi \in \Gamma$. Both readings coincide if Γ and ϕ have no free variables. (The superscript **MSL** may be dropped if clear from context.)

To investigate the relation between the semantics defined above, and the standard and full semantics of separation logic defined earlier, we relate their notions of validity. Our main objective now is to show that $\models^{\mathbf{MSL}} \phi$ implies $\models^{\mathbf{WSL}} \phi$ and $\models^{\mathbf{FSL}} \phi$. To do so, we construct a number of different memory

models given a fixed structure $\mathfrak{A}$ with domain A. Note that there is a natural order between memory models by measuring their cardinality. We will show that the smallest memory model corresponds with the standard semantics, and the largest memory model corresponds with the full semantics.

To consider the smallest memory model H, suppose we start out with the empty set and add in only the necessary heaps to satisfy the requirements for H to be a memory model. The empty heap ϵ must be in H. For every heap and pair of elements of A, we can perform a heap update operation. Every finite sequence of heap updates results in a heap. Performing a heap clear operation only removes a location from the heap, which can be undone by performing a heap update again. The set of finitely-based partial functions over A is the smallest memory model.

The largest memory model is simply the set of all partial functions over A.

Definition 4.1.9. A *finite memory structure* is a memory structure $\mathfrak{H} = (\mathfrak{A}, H)$ where H is the smallest memory model over A.

Definition 4.1.10. A *full memory structure* is a memory structure $\mathfrak{H} = (\mathfrak{A}, H)$ where H is the largest memory model over A.

Every structure induces a full memory structure, since the largest memory model is unique. Similarly, every structure also induces a finite memory structure, since the smallest memory model is also unique. It is easy to verify that the general semantics restricted to finite memory structures coincides with the standard semantics, and the general semantics restricted to full memory structures coincides with the full semantics of separation logic. If the underlying structure has a finite domain, then the smallest and largest memory model coincide (and this confirms that the standard and full semantics coincide in this case too). If the structure has an infinite domain, the smallest and largest memory model are separated.

Proposition 4.1.11.

- $\models^{\mathbf{GSL}} \phi$ *implies* $\models^{\mathbf{MSL}} \phi$,
- $\models^{\mathbf{MSL}} \phi$ *implies* $\models^{\mathbf{WSL}} \phi$ *and* $\models^{\mathbf{FSL}} \phi$.

Proof. If $\models^{\mathbf{GSL}} \phi$ then $\mathfrak{H} \models \phi$ for all general heap structures $\mathfrak{H}$. That includes all memory structures $\mathfrak{H}$. Since each structure $\mathfrak{A}$ induces a finite and full memory structure, we also have $\models^{\mathbf{WSL}} \phi$ and $\models^{\mathbf{FSL}} \phi$. □

From the above proposition, there is an easy heuristic for finding invalid sentences with respect to the general semantics: sentences that are invalid in either the standard semantics or the full semantics are also invalid in the general semantics. The counter-examples for either of them can be used as counter-example in the general semantics, by taking one of the induced memory structures corresponding to the specific counter-example.

Finally, before turning to the semantics of pointer programs, we consider the relation between **MSL** and the semantics underlying the proof system **SL** (or, **RSL** under the assumption that every heap is functional). In the completeness

proof of Chapter 3, we have constructed a structure out of a maximally consistent set of separation logic formulas. It turns out that this constructed structure *also* is a memory structure. Consider that in the constructed structure we have as set H of heaps the set of first-order definable heaps. Now it is easy to see that the empty heap is first-order definable, and given any heap h then the operations of heap update and heap clear are also first-order definable since every element of the domain is expressible, since we have explicitly taken as domain the equivalence classes of terms and hence every element of the underlying structure has a denotation as a term. Going further, for any two first-order definable heaps that are disjoint, we can also form the disjoint union by simply taking the disjunction of the corresponding formulas. And, finally, for any first-order definable heap and any first-order definable subheap, we can also express the 'subtraction' of one heap from the other as a first-order formula based on the given descriptions.

In the remainder we shall speak of *comprehensive memory structures* to mean structures which lie in the intersection of the two classes of general structures, namely those general structures that satisfies both the properties of a comprehensive structure (i.e. closed under semantic comprehension) and the properties of a memory structure (i.e. the set of heaps must be a memory model).

4.2 Semantics of pointer programs

In this section, we introduce the semantics of pointer programs following along the lines of Chapter B on Hoare's logic.

Given a first-order signature Σ. A *pointer program signature* $PPS(\Sigma)$ includes the following operations:

- the *assignment* operation $x := y$
  (where y is an accessible and x is a changed program variable),
- the *lookup* operation $x := [y]$
  (where y is an accessible and x is a changed program variable),
- the *mutation* operation $[x] := y$
  (where x and y are accessible program variables),
- the *allocation* operation $x := \mathbf{new}(y)$
  (where y is an accessible and x is a changed program variable),
- the *deallocation* operation $\mathbf{delete}(x)$
  (where x is an accessible program variable),

and every test is a quantifier-free pure formula $\phi(x_1, \ldots, x_n)$, with as accessible program variables $x_1, \ldots, x_n$ corresponding to the free variables of ϕ. Every pointer program signature is also a first-order program signature (see Definition B.1.2). A *pointer program* is a statement formed from statements based on a pointer program signature (and, similarly, a *recursive pointer program* consists of a set of declarations and a main statement). Note that the tests of (statements of)

pointer programs are pure quantifier-free formulas, as before. To give semantics to pointer programs, we introduce *spatial machine models*, analogous to *logical machine models* of Chapter B. A spatial machine model is based on a memory structure $\mathfrak{H}$. The spatial machine model is a failure-sensitive machine model in the sense that its states are pairs of heaps (in the set of heaps of $\mathfrak{H}$) and valuations (of the underlying structure of $\mathfrak{H}$).

Before introducing the semantics, we introduce auxiliary concepts needed in the formulation of spatial machine models. Given two heaps h, h' (partial functions over some set), then we say that h' has a *finite distance* to h whenever it is the case that $h' = h[a_1 := a'_1] \dots [a_n := a'_n][a_{n+1} := \bot] \dots [a_{n+k} := \bot]$. Informally, one obtains h' by finitely many applications of heap update and heap clear to h. This notion is symmetric: h' has a finite distance to h if and only if h has a finite distance to h'. Note that it is also possible that h and h' have a finite distance even in the case that both h and h' are partial functions without a finite basis (i.e. both have an infinite domain). Further, when we compare sets of states in the notations $X \equiv Y$ **mod** Z and $X \equiv f(X)$ **mod** Z, we speak only about the valuations and leave the heaps untouched.

Definition 4.2.1. A *spatial machine model* $\mathcal{M}$ is a pair of a memory structure $\mathfrak{H}$ and an operationalization consisting of:

- for each operation O, a transition function which is a partial function $O^{\mathcal{M}}$ of valuations to a set of valuations of $\mathfrak{H}$,
- for every operation $x := y$, the transition function $(x := y)^{\mathcal{M}}$ is defined by mapping (h, ρ) to $(h, \rho[x := \rho(y)])$,
- for every operation $x := [y]$, the transition function $(x := [y])^{\mathcal{M}}$ is defined by mapping (h, ρ) to $(h, s[x := h(s(y))])$ if $s(y) \in dom(h)$ and to **fail** otherwise,
- for every operation $[x] := y$, the transition function $([x] := y)^{\mathcal{M}}$ is defined by mapping (h, ρ) to $(h[s(x) := s(y)], s)$ if $s(x) \in dom(h)$ and to **fail** otherwise,
- for every operation $x :=$ **new**(y), the transition function $(x :=$ **new**$(y))^{\mathcal{M}}$ is defined by mapping (h, ρ) to the set $\{(h[n := s(y)], s[x := n]) \mid n \notin dom(h)\}$,
- for every operation **delete**(x), the transition function $($**delete**$(x))^{\mathcal{M}}$ is defined by mapping (h, ρ) to $(h[s(x) := \bot], s)$ if $s(x) \in dom(h)$ and to **fail** otherwise,
- for the transition function $O^{\mathcal{M}}$ we have the *change condition* that either $O^{\mathcal{M}}(h, \rho) =$ **fail** or $\rho'[V_1 \setminus change(O)] = \rho[V_1 \setminus change(O)]$ for every $(h', \rho') \in O^{\mathcal{M}}(h, \rho)$,
- for the transition function $O^{\mathcal{M}}$ we have the *access condition* that states that $O^{\mathcal{M}}(h, \rho) \equiv O^{\mathcal{M}}(h', \rho')$ **mod** $var(O)$ for every ρ, ρ' for which $\rho[access(O)] = \rho'[access(O)]$ holds,
- for the transition function $O^{\mathcal{M}}$ we have the *effect condition* that for every $(h', \rho') \in O^{\mathcal{M}}(h, \rho)$ we have that h' has a finite distance to h,

- for the transition function $O^{\mathcal{M}}$ we have the *frame condition* that for every $O^{\mathcal{M}}(h_0, \rho) \neq \mathbf{fail}$ and $(h', \rho') \in O^{\mathcal{M}}(h_0 \uplus h_1, \rho)$, there is a h'_0 such that $(h'_0, \rho'_0) \in O^{\mathcal{M}}(h_0, \rho)$ and $h' = h'_0 \uplus h_1$.

Since spatial machine models are failure-sensitive machine models, we get the operational and denotational semantics 'for free'. We also write $\langle \mathcal{M}, \mathfrak{H} \rangle$ for a spatial machine model to indicate its underlying memory structure $\mathfrak{H}$. A spatial machine model is a failure-sensitive machine model in the following sense: the state space of a spatial machine model is the set of pairs of heaps and valuations of $\mathfrak{H}$, and the given operationalization induces an operationalization for tests by associating every quantifier-free pure formula ϕ to the set $\mathfrak{H}[\![\phi]\!]^{\mathbf{MSL}}$ that denotes the pairs of heaps and valuations that satisfy ϕ in structure $\mathfrak{H}$. Note that for these formulas ϕ we have $(h, \rho) \in \mathfrak{H}[\![\phi]\!]^{\mathbf{MSL}}$ if and only if $\rho \in \mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$ where $\mathfrak{A}$ is the underlying structure of $\mathfrak{H}$, that is, tests do not depend on the heap.

In fact, in the operational semantics of pointer programs, we have as configurations triples consisting of a program S, a heap h, and a valuation ρ. The successful execution of any basic instruction S is denoted by $(S, h, \rho) \longrightarrow\!\!\!\!\!\rightarrow (\checkmark, h', \rho')$, whereas $(S, h, \rho) \longrightarrow\!\!\!\!\!\rightarrow \mathbf{fail}$ denotes a failing execution (e.g. due to access of a 'dangling pointer'). The small-step semantics (and similar for the big-step semantics) of the primitive operations are as follows, as follows from the definition of the operationalization of spatial machine models above:

- $(x := y, h, \rho) \longrightarrow (\checkmark, h, \rho[x := \rho(y)])$,
- $(x := [y], h, \rho) \longrightarrow (\checkmark, h, \rho[x := h(\rho(y))])$ if $\rho(y) \in dom(h)$,
- $(x := [y], h, \rho) \longrightarrow \mathbf{fail}$ if $\rho(y) \notin dom(h)$,
- $([x] := y, h, \rho) \longrightarrow (\checkmark, h[\rho(x) := \rho(y)], \rho)$ if $\rho(x) \in dom(h)$,
- $([x] := y, h, \rho) \longrightarrow \mathbf{fail}$ if $\rho(x) \notin dom(h)$,
- $(x := \mathbf{new}(y), h, \rho) \longrightarrow (\checkmark, h[n := \rho(y)], \rho[x := n])$ where $n \notin dom(h)$,
- $(\mathbf{delete}(x), h, \rho) \longrightarrow (\checkmark, h[\rho(x) := \bot], \rho)$ if $\rho(x) \in dom(h)$,
- $(\mathbf{delete}(x), h, \rho) \longrightarrow \mathbf{fail}$ if $\rho(x) \notin dom(h)$.

Crucially, with this definition we abstract from out of memory errors, e.g. from the heap which has no free location it is not possible to take a step with the program $x := \mathbf{new}(y)$. Intuitively, this makes 'out of memory' behave similarly as a divergence: it is as if the underlying implementation would keep searching for a free spot on the heap, but diverge since it will never find one.

The other operations do have explicit failures: looking up the value of an unallocated location results in **fail**, so do mutation of an unallocated location, or the deallocation thereof.

Note that it is possible to instantiate this program semantics with different structures, and thus different conceptions of the heap. In the standard semantics one would use finite heaps, but it is also possible to base the operational semantics

on top of the full set of heaps (finite or infinite heaps) of any underlying structure, or to base it on top of any other memory structure introduced in the previous section. However, it is now obvious from the definition above that general structures are insufficient, because we want to define transitions in terms of heap updates and heap clear operations.

Since we also have an interpretation where there are potentially infinite heaps, the design choice to let allocation behave as a divergence in case of no free location becomes more clear. In the case of finite heaps (but an infinite domain of the underlying structure), allocations do not diverge, as is the case in the standard semantics.

Just like in Hoare's logic, we have that the access and change conditions can be lifted to statements S.

Lemma 4.2.2 (Change Lemma)**.** *Given a set of proper states X,*

$$X \equiv \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](X) \textbf{ mod } \mathit{change}(S).$$

Lemma 4.2.3 (Access Lemma)**.** *Given two sets of proper states X, Y such that $X \equiv Y$ **mod** $(V \setminus \mathit{access}(S))$, then*

$$\langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](X) \equiv \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](Y) \textbf{ mod } (V \setminus \mathit{var}(S)).$$

Similar to Hoare's logic, these express that a statement only modifies the variables $\mathit{change}(S)$, and that the outcome of a statement is only dependent on the variables $\mathit{access}(S)$. However, note that these notions require the same initial heap. These properties do not capture the so-called 'heap footprint' of a statement. For dealing with the heap, we have the other two conditions: the effect and frame conditions can also be lifted to statements.

Lemma 4.2.4 (Effect Lemma)**.** *For any $(h', \rho') \in \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h, \rho)$ we have that h' has a finite distance to h.*

Proof. Intuitively we only have to check the small-steps, and check that the property is preserved compositionally. Since we already have that for every primitive operation this property holds due to the definition of spatial machine models, it suffices to observe that in a terminating execution we have only finitely many steps. □

A statement S which has no effect on the heap is called an *effect-free statement*, whereas a statement that does have an effect on the heap is called *effectful*.

Lemma 4.2.5 (Frame Lemma I)**.** *If* **fail** $\notin \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h_0, \rho)$ *then it is also the case that* **fail** $\notin \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h_0 \uplus h_1, \rho)$.

Proof. We know an execution from the smaller heap h_0 does not lead to failure. Suppose we now add additional locations and this causes a failure to appear in the computation. This failure must happen in some execution at a primitive operation (the other statements do not cause failures). Due to the frame condition, and the

fact that the operationalization maps a state to either **fail** or a set of states, we know that if a smaller heap does not lead to failure, a larger heap must also not. But that contradicts the assumption that a failure appears in the computation from a larger heap. □

Lemma 4.2.6 (Frame Lemma II)**.** *Given* $\mathbf{fail} \notin \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h_0, \rho)$ *and* $(h', \rho') \in \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h_0 \uplus h_1, \rho)$*, there is a* h'_0 *such that* $(h'_0, \rho'_0) \in \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h_0, \rho)$ *and* $h' = h'_0 \uplus h_1$*.*

Proof. The intuition is that from the first premise, we know that statement S accesses or changes the locations on heap h_0. Hence, adding additional locations to the heap does not affect the execution of the statement, and these additional locations remain unmodified during the execution (otherwise, if it would change these locations, then the statement S would lead to a failure when executing it on the smaller heap h_0 that lacks these additional locations). □

In fact, the first premise suggests an important concept of pointer programs: the 'footprint' of a program. Let the footprint of a statement S be the set of states (h, ρ) such that $\mathbf{fail} \notin \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](h, \rho)$.

We again have program specifications $\{\phi\}\ S\ \{\psi\}$, being a triple that consists of a precondition ϕ, a program S, and a postcondition ψ. Note that, in Reynolds' logic, the precondition and postcondition are formulas of separation logic, and statements are formed according to a pointer program signature. Every program specification of Hoare's logic is *also* a program specification in Reynolds' logic, since every statement of a pointer program signature is also a statement of a first-order program signature and every formula of classical logic is also a formula of separation logic.

Again we formally define whether a program specification is satisfied, but now in a spatial machine model. Recall that formulas never denote the improper state **fail**. Thus we have the interpretation of program specifications called *strong partial correctness*, defined as such:

$$\langle \mathcal{M}, \mathfrak{H} \rangle \models^{\mathbf{RL}} \{\phi\}\ S\ \{\psi\} \text{ if and only if } \langle \mathcal{M}, \mathfrak{H} \rangle [\![S]\!](\mathfrak{H}[\![\phi]\!]^{\mathbf{MSL}}) \subseteq \mathfrak{H}[\![\psi]\!]^{\mathbf{MSL}}.$$

Since **fail** is never in $\mathfrak{H}[\![\psi]\!]^{\mathbf{MSL}}$, this interpretation explicitly states that the machine never fails when executing program S starting from any state in $\mathfrak{H}[\![\phi]\!]^{\mathbf{MSL}}$. The superscript **RL** stands for Reynolds' Logic.

Before introducing the proof system for Reynolds' logic, note that in Hoare's logic we distinguish the background theory and program theory. Recall that a program theory is a set of program specifications (see Section B.4): it is possible to consider theories to consist of program specifications *or* formulas, since every formula in the background theory can be encoded as a program specification over the **skip** statement. As such, we directly introduce the following notion of semantic consequence in Reynolds' logic, without distinguishing the background theory from the program theory.

Let Γ be a theory (a set of program specifications or formulas). We write $\Gamma \models^{\mathbf{RL}} \{\phi\}\ S\ \{\psi\}$ to mean $\langle \mathcal{M}, \mathfrak{H} \rangle \models^{\mathbf{RL}} \{\phi\}\ S\ \{\psi\}$ for every spatial machine

model $\langle \mathcal{M}, \mathfrak{H} \rangle$ such that $\langle \mathcal{M}, \mathfrak{H} \rangle \models^{\mathbf{RL}} \{\phi'\}\ S'\ \{\psi'\}$ for each $\{\phi'\}\ S'\ \{\psi'\} \in \Gamma$. We then say that the program specification $\{\phi\}\ S\ \{\psi\}$ is a *semantic consequence of* Γ. For an empty theory, we simply write $\models^{\mathbf{RL}} \{\phi\}\ S\ \{\psi\}$ to mean that the program specification is *universally valid*.

4.3 Standard proof system

The standard proof system of Reynolds' logic **RL** is an extension of Hoare's logic **HL** (see Section B.4). Reynolds' logic is an extension of Hoare's logic, so we first revisit the proof rules of Hoare's logic.

If we would interpret the proof rules (axioms are proof rules without premises) of **HL** under the intended interpretation of **RL**, being spatial machine models, do they remain sound? Clearly, if we only consider the instances of the proof rules where the formulas are restricted to pure formulas in separation logic, i.e. those that are heap-independent, these proof rules remain sound also under our new interpretation with respect to spatial machine models. This follows from the fact that spatial machine models are extensions of logical machine models and as such have the same conditions as logical machine models: the interpretation of the basic assignment $x := y$ is the same, and the access and change conditions are also present.

But in Reynolds' logic, we want to extend these rules to all formulas of separation logic. The **skip** and **halt** axioms are easily shown sound, and so is the assignment axiom. The rules concerning complex statements remain sound, since these do not depend on the interpretation of formulas: their semantics is defined at the level of states, which abstracts away from the particular logical structure (being either valuations as in Hoare's logic, or heaps and valuations in Reynolds' logic).

What remains to be considered are the so-called adaptation rules.

- The consequence rule (conseq) is sound because
$$\langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](\mathfrak{A}[\![\phi']\!]^{\mathbf{MSL}}) \subseteq \mathfrak{A}[\![\psi']\!]^{\mathbf{MSL}}$$
follows from monotonicity of the semantics, and the fact that we have both $\mathfrak{A}[\![\psi']\!]^{\mathbf{MSL}} \subseteq \mathfrak{A}[\![\psi]\!]^{\mathbf{MSL}}$ and $\mathfrak{A}[\![\phi]\!]^{\mathbf{MSL}} \subseteq \mathfrak{A}[\![\phi']\!]^{\mathbf{MSL}}$.

- In the substitution rule (subst) we make use of the access lemma to take any computation from $\{\phi\}\ S\ \{\psi\}$ and change the initial state with respect to variable x that is not occurring in S to obtain another computation (the variable x can then not be overwritten by S). This still works as before. Further, the value to assign to x is the value of y, which must have the same value in the initial and final state again due to the change lemma. The specification then is satisfied by applying the substitution lemma on the initial and final state. Since the substitution lemma also holds for separation logic, this works out.

- The $\exists$-introduction rule ($\exists$-intro) still follows from the access lemma, since the value of x cannot have any effect on the computation of S nor influence the denotation of ψ.

- However, the invariance rule (invar) no longer follows from the change lemma. The problem is that the formula χ can be heap-dependent, whereas $\mathit{change}(S)$ only tracks variables and not locations on the heap. If S is effect-free, then it does not change the heap and thus the formula remains invariant. Otherwise, S is effectful and could thus affect (dynamic) parts of the heap on which the formula χ depends. Concretely, we could take a program $[x] := y$ that modifies the location in x to become the value of y. However $\mathit{change}([x] := y)$ is empty, because none of the program variables change value after its execution. If we take the (valid) program specification $\{(x \hookrightarrow -)\}\ [x] := y\ \{\mathbf{true}\}$ as premise, and we would take as invariant formula $(x \hookrightarrow z)$, then the resulting program specification $\{(x \hookrightarrow -) \wedge (x \hookrightarrow z)\}\ [x] := y\ \{\mathbf{true} \wedge (x \hookrightarrow z)\}$ no longer is valid! Namely, take y and z to be different values in the initial state. No variable is changed in the final state when compared to the initial state. However, it no longer is the case that $(x \hookrightarrow z)$ holds in the final state, since the location was modified.

Summarizing, in Reynolds' logic we can have all proof rules of Hoare's logic, also extended to instances which have all formulas of separation logic, *except* for the invariance rule: the invariance rule only remains sound for pure formulas χ.

Furthermore, in the standard proof system for Reynolds' we have the following proof rules [188]. The *frame rule* is introduced to fill up the gap left by the invariance rule, allowing one to adapt local specifications to global specifications. We first introduce $\mathbf{RL}^-$, and later give different sets of axioms to describe the primitive operations of every pointer program.

Definition 4.3.1. The proof system $\mathbf{RL}^-$ consists of:

- program specifications or formulas of separation logic as objects,
- the smallest deduction relation $\vdash^{\mathbf{RL}^-}$ satisfying the conditions:

  (skip) $\vdash^{\mathbf{RL}^-} \{\phi\}\ \mathbf{skip}\ \{\phi\}$,

  (halt) $\vdash^{\mathbf{RL}^-} \{\phi\}\ \mathbf{halt}\ \{\mathbf{false}\}$,

  (assign) $\vdash^{\mathbf{RL}^-} \{\phi[x := y]\}\ x := y\ \{\phi\}$,

  (block) $\{\phi[\vec{x} := \vec{y}]\}\ S\ \{\psi\} \vdash^{\mathbf{RL}^-} \{\phi\}\ \mathbf{begin\ local}\ \vec{x} := \vec{y}; S\ \mathbf{end}\ \{\psi\}$
  if $FV(\psi) \cap \vec{x} = \emptyset$,

  (comp) $\{\phi\}\ S_1\ \{\psi\}, \{\psi\}\ S_2\ \{\chi\} \vdash^{\mathbf{RL}^-} \{\phi\}\ S_1; S_2\ \{\chi\}$,

  (if) $\{\phi \wedge \chi\}\ S_1\ \{\psi\}, \{\phi \wedge \neg\chi\}\ S_2\ \{\psi\} \vdash^{\mathbf{RL}^-} \{\phi\}\ \mathbf{if}\ \chi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{\psi\}$,

  (while) $\{\phi \wedge \chi\}\ S\ \{\phi\} \vdash^{\mathbf{RL}^-} \{\phi\}\ \mathbf{while}\ \chi\ \mathbf{do}\ S\ \mathbf{od}\ \{\phi \wedge \neg\chi\}$,

  (conseq) $(\phi' \to \phi), \{\phi\}\ S\ \{\psi\}, (\psi \to \psi') \vdash^{\mathbf{RL}^-} \{\phi'\}\ S\ \{\psi'\}$,

  (subst) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{RL}^-} \{\phi[x := y]\}\ S\ \{\psi[x := y]\}$
  for $x \notin \mathit{var}(S), y \notin \mathit{change}(S)$,

  (invar) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{RL}^-} \{\phi \wedge \chi\}\ S\ \{\psi \wedge \chi\}$
  for either pure χ or effect-free S, $FV(\chi) \cap \mathit{change}(S) = \emptyset$,

(∃-intro) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{RL}^-} \{\exists x\phi\}\ S\ \{\psi\}$ for $x \notin var(S) \cup FV(\psi)$,

(frame) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{RL}^-} \{\phi * \chi\}\ S\ \{\psi * \chi\}$ if $FV(\chi) \cap change(S) = \emptyset$.

Lemma 4.3.2 (Soundness)**.**

$$\Gamma \vdash^{\mathbf{RL}^-} \{\phi\}\ S\ \{\psi\} \textit{ implies } \Gamma \models^{\mathbf{RL}} \{\phi\}\ S\ \{\psi\}.$$

Proof. By induction on the structure of the deduction. Most cases are already discussed above, except the frame rule. The soundness of the frame rule goes along the following lines, see also [230]. From the premise we know that $\Gamma \models^{\mathbf{RL}} \{\phi\}\ S\ \{\psi\}$. We thus know that the execution of S does not fail in a state that satisfies ϕ. Due to the first frame lemma we know that, if we extend the initial heap with an additional part, it does not lead to failure either. From the second frame lemma we also know that the final state can be split again in the unaffected part of the heap, in which the additional formula χ still holds as was assumed as part of the semantics of the separating conjunction in the initial state. □

Now consider the set of local axioms [188].

Definition 4.3.3. The set of *local* axioms is:

(lookup) $\vdash \{(y \mapsto z)\}\ x := [y]\ \{(x \doteq z) \wedge (y \mapsto z)\}$ where $x \neq y$ and z is fresh,

(lookup') $\vdash \{(x \doteq w) \wedge (x \mapsto z)\}\ x := [x]\ \{(x \doteq z) \wedge (w \mapsto z)\}$ where z, w are fresh,

(mutation) $\vdash \{(x \mapsto -)\}\ [x] := y\ \{(x \mapsto y)\}$,

(allocation) $\vdash \{\mathbf{emp}\}\ x := \mathbf{new}(y)\ \{(x \mapsto y)\}$ where $x \neq y$,

(allocation') $\vdash \{(x \doteq z) \wedge \mathbf{emp}\}\ x := \mathbf{new}(x)\ \{(x \mapsto z)\}$ where z is fresh,

(deallocation) $\vdash \{(x \mapsto -)\}\ \mathbf{delete}(x)\ \{\mathbf{emp}\}$.

Lemma 4.3.4. *The* local *axioms are sound with respect to* **MSL**.

By applying the frame rule it becomes possible to extend some of these program specifications to a global description of the heap.

Next, consider the set of *global* axioms [188].

Definition 4.3.5. The set of *global* axioms is:

(lookup) $\vdash \{\exists z.\, (y \mapsto z) * \phi[w := x]\}\ x := [y]\ \{\exists w.\, (y[x := w] \mapsto x) * \phi[z := x]\}$ where z, w, x are distinct, z, w, y are distinct and $x \notin FV(\phi)$,

(mutation) $\vdash \{(x \mapsto -) * \phi\}\ [x] := y\ \{(x \mapsto y) * \phi\}$,

(allocation) $\vdash \{\phi\}\ x := \mathbf{new}(y)\ \{\exists z.\, (x \mapsto y[x := z]) * \phi[x := z]\}$ where z is fresh,

(deallocation) $\vdash \{(x \mapsto -) * \phi\}\ \mathbf{delete}(x)\ \{\phi\}$.

Lemma 4.3.6. *The* global *axioms are sound with respect to* **MSL**.

There is also the set of *backwards* axioms [188], in the sense that these axioms allow reasoning backwards from a given postcondition.

Definition 4.3.7. The set of *backwards* axioms is:

(assign) $\vdash \{\phi[x := y]\}\ x := y\ \{\phi\}$,

(lookup) $\vdash \{\exists z.\, (y \hookrightarrow z) \wedge \phi[x := z]\}\ x := [y]\ \{\phi\}$ where z is fresh,

(mutation) $\vdash \{(x \mapsto -) * ((x \mapsto y) \mathrel{-\!\!*} \phi)\}\ [x] := y\ \{\phi\}$,

(allocation) $\vdash \{\forall z.\, (z \mapsto y) \mathrel{-\!\!*} \phi[x := z]\}\ x := \mathbf{new}(y)\ \{\phi\}$ where z is fresh,

(deallocation) $\vdash \{(x \mapsto -) * \phi\}\ \mathbf{delete}(x)\ \{\phi\}$.

These backwards axioms also express the weakest precondition:

Lemma 4.3.8. *The* backwards *axioms are sound with respect to* **MSL***, and describe the weakest precondition with respect to the given primitive operations and postcondition.*

And finally the set of *forwards* axioms [15], in the sense that these axioms allow reasoning forwards from a given precondition.

Definition 4.3.9. The set of *forwards* axioms is:

(assign) $\vdash \{\phi\}\ x := y\ \{\exists z.\, \phi[x := z] \wedge (y[x := z] \doteq x)\}$ where z is fresh,

(lookup) $\vdash \{\phi \wedge (y \hookrightarrow -)\}\ x := [y]\ \{\exists z.\, \chi * \neg(\chi \mathrel{-\!\!*} \neg\phi[x := z])\}$
where $\chi = (y[x := z] \mapsto x)$ and z is fresh,

(mutation) $\vdash \{\phi \wedge (x \hookrightarrow -)\}\ [x] := y\ \{(x \mapsto y) * \neg((x \mapsto -) \mathrel{-\!\!*} \neg\phi)\}$,

(allocation) $\vdash \{\phi\}\ x := \mathbf{new}(y)\ \{\exists z.\, (x \mapsto y[x := z]) * \phi[x := z]\}$,

(deallocation) $\vdash \{\phi \wedge (x \hookrightarrow -)\}\ \mathbf{delete}(x)\ \{\neg((x \mapsto -) \mathrel{-\!\!*} \neg\phi)\}$.

Lemma 4.3.10. *The* forwards *axioms are sound with respect to* **MSL***, and describe the strongest postcondition with respect to the given primitive operations and precondition.*

These forwards axioms also express the strongest postcondition with respect to the given primitive operations and precondition, but note that we have additional assumptions in the precondition that ensures absence of failure.

Finally, we observe that we have admissibility of the frame rule for pure pointer programs in case we take the *backwards* axioms for the primitive operations. In pure pointer programs, there are no other operations than the operations of assignment, lookup, mutation, allocation and deallocation.

Lemma 4.3.11. *The frame rule is admissible in the proof system* $\mathbf{RL}^{-}$ *with the* backwards *axioms, given that the background theory is maximally consistent.*

Proof. Consider a deduction that makes use of the frame rule. The strategy is to 'push upward' the instance of the frame rule to the top of the deduction, i.e. where there is an axiom applied that is either (skip), (halt), (assign), or one of the pointer program operations (lookup), (mutation), (allocation), (deallocation). For these axioms, it can be verified that the conclusion of the frame rule is deducible (from the empty context). We then have to analyze the adaptation rules, and the structural rules.

For the adaptation rule, we illustrate the proof by showing how to push the frame rule up the consequence rule. Consider the following deduction in which the frame rule is applied directly after the consequence rule:

$$\dfrac{\dfrac{\phi' \to \phi \quad \overset{\mathcal{D}}{\{\phi\}\ S\ \{\psi\}} \quad \psi \to \psi'}{\{\phi'\}\ S\ \{\psi'\}}}{\{\phi' * \chi\}\ S\ \{\psi' * \chi\}}$$

It is our induction hypothesis that the frame rule is admissible for shorter deductions, so from deduction $\mathcal{D}$ with conclusion $\{\phi\}\ S\ \{\psi\}$ we obtain deduction $\mathcal{D}'$ with conclusion $\{\phi * \chi\}\ S\ \{\psi * \chi\}$. By then applying the consequence rule we obtain the following deduction:

$$\dfrac{\phi' * \chi \to \phi * \chi \quad \overset{\mathcal{D}'}{\{\phi * \chi\}\ S\ \{\psi * \chi\}} \quad \psi * \chi \to \psi' * \chi}{\{\phi' * \chi\}\ S\ \{\psi' * \chi\}}$$

since we know $\phi' \to \phi$ and $\psi \to \psi'$ are in the background theory and the background theory is maximally consistent, the two remaining premises must be in the background theory too. The remaining adaptation rules are similar.

For the structural rules, we illustrate how the proof goes by looking at sequential composition. Consider that we have a deduction in which the frame rule is applied directly following the sequential composition:

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\{\phi\}\ S_1\ \{\psi\}} \quad \overset{\mathcal{D}_2}{\{\psi\}\ S_2\ \{\chi\}}}{\{\phi\}\ S_1; S_2\ \{\chi\}}}{\{\phi * \xi\}\ S_1; S_2\ \{\chi * \xi\}}$$

By induction hypothesis, we obtain from $\mathcal{D}_1$ and $\mathcal{D}_2$ two deductions $\mathcal{D}_1'$ and $\mathcal{D}_2'$ with respectively conclusions $\{\phi * \xi\}\ S_1\ \{\psi * \xi\}$ and $\{\psi * \xi\}\ S_2\ \{\chi * \xi\}$. Note that the changed variables of the smaller programs are contained in the changed variables of the sequential composition, so the frame rule would be applicable. We then obtain the following deduction:

$$\dfrac{\overset{\mathcal{D}_1'}{\{\phi * \xi\}\ S_1\ \{\psi * \xi\}} \quad \overset{\mathcal{D}_2'}{\{\psi * \xi\}\ S_2\ \{\chi * \xi\}}}{\{\phi * \xi\}\ S_1; S_2\ \{\chi * \xi\}}$$

which finishes this case. The remaining structural cases are similar. For the **while** and **if** rules, one also needs the equivalence $(\phi \wedge \chi) * \psi \equiv (\phi * \psi) \wedge \chi$, which holds since χ is a pure quantifier-free formula. □

Note that it is an open problem whether the proof above may be adapted to the setting of recursive procedures, with or without parameters (see also the Ph.D. thesis of Al Ameen [6]). Although intuitively we can 'push' the frame rule through the assumptions about each procedure call, this may result in an infinite amount of assumptions obtained that way and it is not obvious whether this infinite set of assumptions is compact, in the sense that all consequences can also be derived from a finite subset of these infinite assumptions without using the frame rule.

4.4 Dynamic separation logic

We have now seen different axiomatizations of the primitive operations of all pointer programs. In particular, it can be observed that in the *backwards* and *forwards* sets of axioms, we do not systematically analyze the structure of the given postcondition or precondition. For example, in the mutation axiom of the *backwards* set, the given postcondition is simply *verbatim* part of the weakest precondition. Furthermore, the axioms for (mutation), (allocation) and (deallocation) all increase the complexity of the generated formula as measured by their number of nested separating connectives. Thus, even starting with a first-order formula, the resulting generated formula necessarily is a formula of separation logic. This is contrary to how, e.g., the (assign) axioms work in *backwards* and *forwards*, and the (lookup) axiom works in *backwards*, which perform a substitution to perform a structural analysis of the given postcondition, and do not introduce additional separating connectives.

This raises the questions: are there alternative ways to axiomatize these operations? In particular, is there a way to axiomatize (mutation), (allocation) and (deallocation) so that the structure of the given postcondition is analyzed, akin to a substitution operator? Can we give weakest preconditions and strongest postconditions without increasing the nesting depth of separating connectives?

To answer these questions, we introduce in our assertion language an additional *program modality* for each statement S, which has highest binding priority, denoted as $[S]p$. This extended language is called *dynamic separation logic* (DSL), and as such the syntax of dynamic separation logic becomes:

$$p, q ::= \ldots \mid [S]p$$

We shall use the Roman letters p, q to stand for formulas of DSL, whereas we use the Greek letters ϕ, ψ to stand for formulas of separation logic. Note that in case of the assignment $x := y$ the program modality $[x := y]\phi$ is different from the (capture-avoiding) substitution operator $\phi[x := y]$, since the former is a formula of our extended language, whereas the latter is a meta-operation defined on the separation logic formula ϕ.

We also extend the semantics of separation logic to interpret the additional program modalities. The intended semantics of dynamic separation logic extends the semantics of separation logic by interpreting the modality $[S]p$ as expressing the weakest precondition of statement S with postcondition p:

- ...
- $\langle \mathcal{M}, \mathfrak{H} \rangle, h, \rho \models^{\mathbf{DSL}} [S]p$ iff $(S, h, \rho) \not\longrightarrow\!\!\!\!\!\rightarrow \mathbf{fail}$ and $\langle \mathcal{M}, \mathfrak{H} \rangle, h', \rho' \models p$ for all h', s' such that $(S, h, \rho) \longrightarrow\!\!\!\!\!\rightarrow (\checkmark, h', \rho')$.

Note that to give semantics to formulas of dynamic separation logic, we need to interpret the formulas with respect to a fixed spatial machine model $\langle \mathcal{M}, \mathfrak{H} \rangle$, which itself depends on a memory structure $\mathfrak{H} = (\mathfrak{A}, H)$, with an underlying structure $\mathfrak{A}$ and memory model H, which is the same memory structure with which we evaluate the formula of dynamic separation logic. This also explains the need for the effect condition on spatial machine models, since we need the resulting heap h' in every final configuration to be in the set of heaps H. Since h' is a finite distance from h, and we know that memory models are closed under the operations of heap update and heap clear, we know that h' must be in the memory model too.

Extending Reynolds' logic to also include formulas of dynamic separation logic in its program specifications, it is not difficult to see that we have that

$$\models \{[S]p\}\ S\ \{p\}$$

holds, and $\models \{p\}\ S\ \{q\}$ implies $\models^{\mathbf{DSL}} p \to [S]q$, that is, $[S]q$ indeed expresses the weakest precondition of statement S and postcondition q.

In dynamic logic axioms are introduced to simplify formulas in which modalities occur. We have the following basic equivalences **E1-3** for assignments.

Lemma 4.4.1 (Assignment)**.** *Let the statement S be the assignment $x := y$.*

$$[S]\mathbf{false} \equiv \mathbf{false} \qquad (\mathbf{E1})$$
$$[S](p \circ q) \equiv [S]p \circ [S]q \qquad (\mathbf{E2})$$
$$[S](\forall z p) \equiv \forall z([S]p) \qquad (\mathbf{E3})$$
$$[S]b \equiv b[S] \qquad (\mathbf{E4})$$

In **E2** *we have that $\circ$ stands for the binary connectives $\to$, $*$, $-\!\!*$.*
In **E3** *we assume that z is not equal to x or y.*
In **E4** *we have that b is either $(z_1 \doteq z_2)$, $C(z_1, \ldots, z_n)$, or $(z_1 \hookrightarrow z_2)$.*

The proofs of these equivalences for $[x := y]p$ proceed by a straightforward induction on the structure of p. The base cases of logical equality, predicates, and the weak 'points to' construct are handled by a straightforward extension of the *substitution lemma* of separation logic. In fact, for assignments, by **E4** as base case and **E1-3** for the inductive cases, we have for any formula of separation logic ϕ,

$$[x := y]\phi \equiv \phi[x := y],$$

where the latter is the (capture-avoiding) substitution operator. The reason we present the axioms **E1-4** in the way it is done above, is because we want to analyze what happens when we take different statements in the place of S: do these axioms then still hold?

The above equivalences **E1-3** do not hold in general for the other primitive operations of pointer programs. Let x, y be distinct variables. For example,

$$[x := [y]]\mathbf{false} \equiv \neg(y \hookrightarrow -),$$

showing that lookup fails **E1**. For allocation, we do have

$$[x := \mathbf{new}(y)]\mathbf{false} \equiv \mathbf{false},$$

but $[x := \mathbf{new}(y)](x \neq y)$ is not equivalent to $\neg([x := \mathbf{new}(y)](x \doteq y))$, since

$$[x := \mathbf{new}(y)](x \neq y) \equiv (y \hookrightarrow -),$$

because to end up in a final state where $x \neq y$ we need to have that y is already allocated in the initial state, whereas

$$[x := \mathbf{new}(y)](x \doteq y) \equiv \forall z.\, ((z \not\hookrightarrow -) \rightarrow z \doteq y),$$

which forces the only free location to be y. But then it is also the case that $\neg([x := \mathbf{new}(y)](x \doteq y))$ expresses that $(z \not\hookrightarrow -)$ for some $z \neq y$. So this shows allocation fails **E2**. Similar examples exists for the other primitive operations.

We now introduce new primitive operations, separate from pointer programs, called *pseudo-operations*. These pseudo-operations are not part of pointer programs, but we can give them semantics in the usual way through an operationalization. We have the pseudo-operation

$$\langle x \rangle := e$$

called *heap update*, and

$$\langle x \rangle := \bot$$

called *heap clear*. These pseudo-operations could be described by the following small-step transitions:

$$(\langle x \rangle := y, h, \rho) \longrightarrow (\checkmark, h[\rho(x) := \rho(y)], \rho)$$
$$(\langle x \rangle := \bot, h, \rho) \longrightarrow (\checkmark, h[\rho(x) := \bot], \rho)$$

In contrast to the mutation and deallocation operations, these pseudo-operations do not require that $\rho(x) \in dom(h)$, e.g., if $\rho(x) \notin dom(h)$ then the heap update $\langle x \rangle := y$ extends the domain of the heap, whereas mutation $[x] := y$ leads to failure in that case.

It is now crucial to observe that these are *pseudo*-operations, precisely because they fail the frame condition. As such, these operations can never occur in a spatial machine model, which requires the frame condition to hold for all operations. Before, we could establish the footprint of a pointer program simply by running a

program on a small heap: if the program would lead to failure, then the location that was missing from the initial heap necessarily is in the footprint. However, these pseudo-operations never fail. Hence these cannot be used to determine the footprint of a program.

That the pseudo-operations do not satisfy the frame condition can best be seen by considering why the frame rule (in a hypothetical situation where we let the pseudo-operations in the place of a statement) would become unsound. Clearly, we have that

$$\models \{\mathbf{emp}\}\ \langle x\rangle := y\ \{(x \mapsto y)\}$$

holds. However, the conclusion of the frame rule fails:

$$\not\models \{\mathbf{emp} * (x \mapsto y)\}\ \langle x\rangle := y\ \{(x \mapsto y) * (x \mapsto y)\}$$

because the initial state is satisfiable (there surely is a heap in which only the location x is allocated and has value y), and the execution successfully terminates (the pseudo-operations never lead to failure). But, the final state does not satisfy $(x \mapsto y) * (x \mapsto y)$ because that formula is equivalent to **false**.

Strictly speaking, we thus consider not only the modality $[S]\phi$ where S is given semantics by a spatial machine model, but also consider modalities over these two pseudo-operations: $[\langle x\rangle := y]\phi$ and $[\langle x\rangle := \bot]\phi$.

The above equivalences **E1-3**, with **E2** restricted to the (standard) logical connectives, *do* hold for the *pseudo*-operations.

In the following lemma we give an axiomatization in dynamic separation logic of the primitive operations in terms of simple assignments and these two pseudo-operations. For comparison we also give the standard *backwards* axiomatization [188, 81, 15].

Lemma 4.4.2 (Axioms basic instructions)**.**

$$[x := [e]]p \equiv \exists y((e \hookrightarrow y) \wedge [x := y]p), \tag{E5}$$

$$[[x] := e]p \equiv \begin{cases} (x \hookrightarrow -) \wedge [\langle x\rangle := e]p \\ (x \mapsto -) * ((x \mapsto e) \mathrel{-\!\!*} p) \end{cases} \tag{E6}$$

$$[x := \mathbf{new}(e)]p \equiv \begin{cases} \forall x((x \not\hookrightarrow -) \to [\langle x\rangle := e]p) \\ \forall x((x \mapsto e) \mathrel{-\!\!*} p) \end{cases} \tag{E7}$$

$$[\mathbf{delete}(x)]p \equiv \begin{cases} (x \hookrightarrow -) \wedge [\langle x\rangle := \bot]p \\ (x \mapsto -) * p \end{cases} \tag{E8}$$

We require in the axiom for $x := \mathbf{new}(e)$ that x does not appear in e, for technical convenience.

In the sequel **E5-8** refer to the corresponding dynamic separation logic equivalences. The proofs of these equivalences are straightforward (consist simply of expanding the semantics of the involved modalities) and therefore omitted.

We have the following separation logic axiomatization of the heap update and heap clear pseudo-operations.

$$[\langle x\rangle := e]p \equiv ((x \mapsto -) * ((x \mapsto e) \mathrel{-\!\!*} p)) \vee ((x \not\hookrightarrow -) \wedge ((x \mapsto e) \mathrel{-\!\!*} p))$$
$$[\langle x\rangle := \bot]p \equiv ((x \mapsto -) * p) \vee ((x \not\hookrightarrow -) \wedge p)$$

This axiomatization thus requires a case distinction between whether or not x is allocated.

Note that, letting p be any formula in separation logic ϕ, we have that $[x := y]\phi$ in **E5** reduces to $\phi[x := y]$ by **E1-4**. As such, it is possible to eliminate the modality, in the case of the assignment and lookup instructions.

We now want to eliminate the modalities for the heap update and heap clear instructions compositionally in terms of p, because such an elimination would also allow us to eliminate the modalities of the other instructions. What thus remains for a complete axiomatization is a characterization of $[S]b$, $[S](e \hookrightarrow e')$, $[S](p * q)$, and $[S](p \mathrel{-\!\!*} q)$, where S denotes one of the two pseudo-instructions. Lemma 4.4.3 provides an axiomatization in DSL of a heap update.

Lemma 4.4.3 (Heap update)**.** *We have the following equivalences for the heap update modality.*

$$[\langle x\rangle := e]b \equiv b, \tag{**E9**}$$

$$[\langle x\rangle := e](e' \hookrightarrow e'') \equiv (x = e' \wedge e'' = e) \vee (x \neq e' \wedge e' \hookrightarrow e''), \tag{**E10**}$$

$$[\langle x\rangle := e](p * q) \equiv ([\langle x\rangle := e]p * q') \vee (p' * [\langle x\rangle := e]q), \tag{**E11**}$$

$$[\langle x\rangle := e](p \mathrel{-\!\!*} q) \equiv p' \mathrel{-\!\!*} [\langle x\rangle := e]q, \tag{**E12**}$$

where p' abbreviates $p \wedge (x \not\hookrightarrow -)$ and, similarly, q' abbreviates $q \wedge (x \not\hookrightarrow -)$.

These equivalences we can informally explain as follows. Since the heap update $\langle x\rangle := e$ does not affect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $([\langle x\rangle := e]b) \equiv b$.

Predicting whether $(e' \hookrightarrow e'')$ holds after $\langle x\rangle := e$, we only need to make a distinction between whether x and e' are aliases, that is, whether they denote the same location, which is simply expressed by $x = e'$. If $x = e'$ then $e'' = e$ should hold, otherwise $(e' \hookrightarrow e'')$ (note again, that $\langle x\rangle := e$ does not affect the values of the expressions e, e' and e''). As a basic example, we compute

$$\begin{array}{lcl}
[\langle x\rangle := e](y \hookrightarrow -) & \equiv & (\text{definition } y \hookrightarrow -) \\
[\langle x\rangle := e]\exists z(y \hookrightarrow z) & \equiv & (\textbf{E3}) \\
\exists z[\langle x\rangle := e](y \hookrightarrow z) & \equiv & (\textbf{E10}) \\
\exists z((y = x \wedge e = z) \vee (y \neq x \wedge (y \hookrightarrow z))) & \equiv & (\text{semantics SL}) \\
y \neq x \to (y \hookrightarrow -) & &
\end{array}$$

We use this derived equivalence in the following example:

$$\begin{array}{lcl}
[\langle x\rangle := e](y \mapsto -) & \equiv & (\text{definition } y \mapsto -) \\
[\langle x\rangle := e]((y \hookrightarrow -) \wedge \forall z((z \hookrightarrow -) \to z = y)) & \equiv & (\textbf{E2, E3, E9}) \\
[\langle x\rangle := e](y \hookrightarrow -) \wedge \forall z([\langle x\rangle := e](z \hookrightarrow -) \to z = y) & \equiv & (\text{see above}) \\
(y \neq x \to (y \hookrightarrow -)) \wedge \forall z((z \neq x \to (z \hookrightarrow -)) \to z = y) & \equiv & (\text{semantics SL}) \\
y = x \wedge (\mathbf{emp} \vee (x \mapsto -)) & &
\end{array}$$

Predicting whether $(p * q)$ holds after the heap update $\langle x\rangle := e$, we need to distinguish between whether p or q holds for the sub-heap that contains the

(updated) location x. Since we do not assume that x is already allocated, we instead distinguish between whether p or q holds initially for the sub-heap that does *not* contain the updated location x. As a simple example, we compute

$$
\begin{array}{lcl}
[\langle x\rangle := e](\mathbf{true} * (x \mapsto -)) & \equiv & (\mathbf{E9,E11}) \\
(\mathbf{true} * ((x \mapsto -) \wedge (x \not\mapsto -))) \vee ((x \not\mapsto -) * [\langle x\rangle := e](x \mapsto -) & \equiv & (\text{see above}) \\
(\mathbf{true} * ((x \mapsto -) \wedge (x \not\mapsto -))) \vee ((x \not\mapsto -) * (\mathbf{emp} \vee (x \mapsto -))) & \equiv & (\text{semantics SL}) \\
(\mathbf{true} * \mathbf{false}) \vee ((x \not\mapsto -) * (\mathbf{emp} \vee (x \mapsto -))) & \equiv & (\text{semantics SL}) \\
\mathbf{true} & &
\end{array}
$$

Note that this coincides with the above calculation of $[\langle x\rangle := e](y \hookrightarrow -)$, which also reduces to **true**, instantiating y by x.

The semantics of $(p \mathrel{-\!\!*} q)$ after the heap update $\langle x\rangle := e$ involves universal quantification over all disjoint heaps that do not contain x (because after the heap update x is allocated). Therefore we simply add the condition that x is not allocated to p, and apply the heap update to q. As a very basic example, we compute

$$
\begin{array}{lcl}
[\langle x\rangle := 0]((y \hookrightarrow 1) \mathrel{-\!\!*} (y \hookrightarrow 1)) & \equiv & (\mathbf{E12}) \\
((y \mapsto 1) \wedge (x \not\mapsto -)) \mathrel{-\!\!*} [\langle x\rangle := 0](y \hookrightarrow 1)) & \equiv & (\mathbf{E10}) \\
((y \mapsto 1) \wedge (x \not\mapsto -)) \mathrel{-\!\!*} ((y = x \wedge 0 = 1) \vee (y \neq x \wedge y \hookrightarrow 1)) & \equiv & (\text{semantics SL}) \\
\mathbf{true} & &
\end{array}
$$

Note that $(y \hookrightarrow 1) \mathrel{-\!\!*} (y \hookrightarrow 1) \equiv \mathbf{true}$ and $[\langle x\rangle := 0]\mathbf{true} \equiv \mathbf{true}$.

Proof of Lemma 4.4.3.

E9 $h, s \models [\langle x\rangle := e]b$
iff (semantics heap update modality)
$h[s(x) := s(e)], s \models b$
iff (b does not depend on the heap)
$h, s \models b$

E10 $h, s \models [\langle x\rangle := e](e' \hookrightarrow e'')$
iff (semantics heap update modality)
$h[s(x) := s(e)], s \models e' \hookrightarrow e''$
iff (semantics points-to)
$h[s(x) := s(e)](s(e')) = s(e'')$
iff (definition $h[s(x) := s(e)]$)
if $s(x) = s(e')$ then $s(e) = s(e'')$ else $h(s(e')) = s(e'')$
iff (semantics assertions)
$h, s \models (x = e' \wedge e'' = e) \vee (x \neq e' \wedge e' \hookrightarrow e'')$

E11 $h, s \models [\langle x\rangle := e](p * q)$
iff (semantics heap update modality)
$h[s(x) := s(e)], s \models p * q$.
From here we proceed as follows. By the semantics of separating conjunction,

there exist h_1 and h_2 such that $h[s(x) := s(e)] = h_1 \uplus h_2$, $h_1, s \models p$, and $h_2, s \models q$. Let $s(x) \in dom(h_1)$ (the other case runs similarly). So $h[s(x) := s(e)] = h_1 \uplus h_2$ implies $h_1(s(x)) = s(e)$ and $h = h_1[s(x) := h(x)] \uplus h_2$, By the semantics of the heap update modality, $h_1(s(x)) = s(e)$ and $h_1, s \models p$ implies $h_1[s(x) := h(x)], s \models [\langle x \rangle := e]p$. Since $s(x) \notin dom(h_2)$, we have $h_2, s \models q \wedge x \not\mapsto -$. By the semantics of separation conjunction we conclude that $h, s \models [\langle x \rangle := e]p * q'$ (q' denotes $q \wedge x \not\mapsto -$).

In the other direction, from $h, s \models [\langle x \rangle := e]p * q'$ (the other case runs similarly) we derive that there exist h_1 and h_2 such that $h = h_1 \uplus h_2$, $h_1, s \models [\langle x \rangle := e]p$ and $h_2, s \models q'$. By the semantics of the heap update modality it follows that $h_1[s(x) := s(e)], s \models p$. Since $s(x) \notin dom(h_2)$, we have that $h[s(x) := s(e)] = h_1[s(x) := s(e)] \uplus h_2$, and so $h[s(x) := s(e)], s \models p * q$, that is, $h, s \models [\langle x \rangle := e](p * q)$.

E12 $h, s \models [\langle x \rangle := e](p \mathrel{-\!\!*} q)$
iff (semantics of heap update modality)
$h[s(x) := s(e)], s \models p \mathrel{-\!\!*} q$
iff (semantics separating implication)
for every h' disjoint from $h[s(x) := s(e)]$: if $h', s \models p$ then $h[s(x) := s(e)] \uplus h', s \models q$
iff (since $s(x) \notin dom(h')$)
for every h' disjoint from h: if $h', s \models p \wedge x \not\mapsto -$ then $(h \uplus h')[s(x) := s(e)], s \models q$
iff (semantics of heap update modality)
for every h' disjoint from h: if $h', s \models p \wedge x \not\mapsto -$ then $h \uplus h', s \models [s(x) := s(e)]q$
iff (semantics separating implication)
$h, s \models (p \wedge x \not\mapsto -) \mathrel{-\!\!*} [\langle x \rangle := e]q$. □

The equivalences for the heap clear modality in the following lemma can be informally explained as follows. Since $\langle x \rangle := \bot$ does not affect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $b[\langle x \rangle := \bot] = b$. For $e \hookrightarrow e'$ to hold after executing $\langle x \rangle := \bot$, we must initially have that $x \neq e$ and $e \hookrightarrow e'$. As a simple example, we have that $\forall y, z(y \not\mapsto z)$ characterizes the empty heap. It follows that $[\langle x \rangle := \bot](\forall y, z(y \not\mapsto z))$ is equivalent to $\forall y, z(\neg(y \neq x \wedge y \hookrightarrow z))$. The latter first-order formula is equivalent to $\forall y, z(y = x \vee y \not\mapsto z)$. This assertion thus states that the domain consists at most of the location x, which indeed ensures that after $\langle x \rangle := \bot$ the heap is empty. To ensure that $p * q$ holds after clearing x it suffices to show that the initial heap can be split such that both p and q hold in their respective sub-heaps with x cleared. The semantics of $p \mathrel{-\!\!*} q$ after clearing x involves universal quantification over all disjoint heaps that do may contain x, whereas before executing $\langle x \rangle := \bot$ it involves universal quantification over all disjoint heaps that do *not* contain x, in case x is allocated initially. To formalize in the initial configuration universal quantification over all disjoint heaps we distinguish between all disjoint heaps that do not contain x and *simulate* all disjoint heaps that contain x by interpreting both p and q in $p \mathrel{-\!\!*} q$ in the context of heap updates $\langle x \rangle := y$ with *arbitrary* values y for the

location x.

As a very basic example, consider $[\langle x\rangle := \bot]((x \hookrightarrow 0) \mathrel{-\!\!*} (x \hookrightarrow 0))$, which should be equivalent to **true**. The left conjunct $((x \hookrightarrow 0) \wedge (x \not\hookrightarrow -)) \mathrel{-\!\!*} [\langle x\rangle := \bot](x \hookrightarrow 0))$ of the resulting formula after applying **E16** is equivalent to **true** (because $(x \hookrightarrow 0) \wedge (x \not\hookrightarrow -)$ is equivalent to **false**). We compute the second conjunct (in the application of **E10** we omitted some trivial reasoning steps):

$$\begin{array}{lcl} \forall y([\langle x\rangle := y](x \hookrightarrow 0) \mathrel{-\!\!*} [\langle x\rangle := y](x \hookrightarrow 0) & \equiv & (\textbf{E10}) \\ \forall y(y = 0 \mathrel{-\!\!*} y = 0) & \equiv & (\text{semantics SL}) \\ \textbf{true} & & \end{array}$$

Lemma 4.4.4 (Heap clear)**.** *We have the following equivalences for the heap clear modality.*

$$[\langle x\rangle := \bot]b \equiv b, \tag{\textbf{E13}}$$

$$[\langle x\rangle := \bot](e \hookrightarrow e') \equiv (x \neq e) \wedge (e \hookrightarrow e'), \tag{\textbf{E14}}$$

$$[\langle x\rangle := \bot](p * q) \equiv [\langle x\rangle := \bot]p * [\langle x\rangle := \bot]q, \tag{\textbf{E15}}$$

$$[\langle x\rangle := \bot](p \mathrel{-\!\!*} q) \equiv \begin{array}{l}((p \wedge x \not\hookrightarrow -) \mathrel{-\!\!*} [\langle x\rangle := \bot]q) \wedge \\ \forall y([\langle x\rangle := y]p \mathrel{-\!\!*} [\langle x\rangle := y]q),\end{array} \tag{\textbf{E16}}$$

where y is fresh.

Proof. Here we go.

E13 $[\langle x\rangle := \bot]b \equiv b$. As above, it suffices to observe that the evaluation of b does not depend on the heap.

E14 $h, s \models [\langle x\rangle := \bot](e \hookrightarrow e')$
iff (semantics heap clear modality)
$h[\langle s(x)\rangle := \bot], s \models e \hookrightarrow e'$
iff (semantics points-to)
$s(e) \in dom(h[\langle s(x)\rangle := \bot])$ and $h[\langle s(x)\rangle := \bot](s(e)) = h(s(e)) = s(e')$
iff (semantics assertions)
$h, s \models x \neq e \wedge e \hookrightarrow e'$

E15 $h, s \models [\langle x\rangle := \bot](p * q)$
iff (semantics heap clear modality)
$h[\langle s(x)\rangle := \bot], s \models p * q$
iff (semantics separating conjunction)
$h_1, s \models p$ and $h_2, s \models q$, for some h_1, h_2 such that $h[\langle s(x)\rangle := \bot] = h_1 \uplus h_2$
iff (semantics heap clear modality)
$h_1, s \models [\langle x\rangle := \bot]p$ and $h_2, s \models [\langle x\rangle := \bot]q$, for some h_1, h_2 such that $h = h_1 \uplus h_2$.
Note: $h = h_1 \uplus h_2$ implies $h[\langle s(x)\rangle := \bot] = h_1[\langle s(x)\rangle := \bot] \uplus h_2[\langle s(x)\rangle := \bot]$, and, conversely, $h[\langle s(x)\rangle := \bot] = h_1 \uplus h_2$ implies there exists h_1', h_2' such that $h = h_1' \uplus h_2'$ and $h_1 = h_1'[\langle s(x)\rangle := \bot]$ and $h_2 = h_2'[\langle s(x)\rangle := \bot]$.

E16 $h, s \models [\langle x\rangle := \bot](p \mathrel{-\!\!*} q)$
iff (semantics heap clear modality)
$h[s(x) :=\bot], s \models p \mathrel{-\!\!*} q$.
From here we proceed as follows. First we show that $h, s \models ((p \wedge x \not\leftrightarrow -) \mathrel{-\!\!*} [\langle x\rangle := \bot]q)$ and $h, s \models \forall y([\langle x\rangle := y]p \mathrel{-\!\!*} [\langle x\rangle := y]q)$ implies $h[s(x) :=\bot], s \models p \mathrel{-\!\!*} q$. Let h' be disjoint from $h[s(x) :=\bot]$ and $h', s \models p$. We have to show that $h[s(x) :=\bot] \uplus h', s \models q$. We distinguish the following two cases.

- First, let $s(x) \in dom(h')$. We then introduce $s' = s[y := h'(s(x))]$. We have $h', s' \models p$ (since y does not occur in p), so it follows by the semantics of the heap update modality that $h'[s(x) := \bot], s' \models [\langle x\rangle := y]p$. Since $h'[s(x) := \bot]$ and h are disjoint (which clearly follows from that h' and $h[s(x) := \bot]$ are disjoint), and since $h, s' \models [\langle x\rangle := y]p \mathrel{-\!\!*} [\langle x\rangle := y]q$, we have that $h \uplus (h'[s(x) := \bot]), s' \models [\langle x\rangle := y]q$. Applying again the semantics of the heap update modality, we obtain $(h \uplus (h'[s(x) := \bot]))[s(x) := s'(y)], s' \models q$. We then can conclude this case observing that y does not occur in q and that $h[s(x) := \bot] \uplus h' = (h \uplus (h'[s(x) := \bot]))[s(x) := s'(y)]$.
- Next, let $s(x) \notin dom(h')$. So h' and h are disjoint, and thus (since $h, s \models (p \wedge x \not\leftrightarrow -) \mathrel{-\!\!*} [\langle x\rangle := \bot]q$) we have $h \uplus h', s \models [\langle x\rangle := \bot]q$. From which we derive $(h \uplus h')[s(x) := \bot], s \models q$ by the induction hypothesis. We then can conclude this case by the observation that $h[s(x) := \bot] \uplus h' = (h \uplus h')[s(x) := \bot]$.

Conversely, assuming $h[s(x) := \bot], s \models p \mathrel{-\!\!*} q$, we first show that $h, s \models (p \wedge x \not\leftrightarrow -) \mathrel{-\!\!*} [\langle x\rangle := \bot]q$ and then $h, s \models \forall y([\langle x\rangle := y]p \mathrel{-\!\!*} [\langle x\rangle := y]q)$.

- Let h' be disjoint from h and $h', s \models p \wedge x \not\leftrightarrow -$. We have to show that $h \uplus h', s \models [\langle x\rangle := \bot]q$, that is, $(h \uplus h')[s(x) :=\bot], s \models q$ (by the semantics of the heap clear update). Clearly, $h[s(x) := \bot]$ and h' are disjoint, and so $h[s(x) := \bot] \uplus h', s \models q$ follows from our assumption. We then can conclude this case by the observation that $(h \uplus h')[s(x) := \bot] = h[s(x) := \bot] \uplus h'$, because $s(x) \notin dom(h')$.
- Let h' be disjoint from h and $s' = s[y := n]$, for some n such that $h', s' \models [\langle x\rangle := y]p$. We have to show that $h \uplus h', s' \models [\langle x\rangle := y]q$. By the semantics of the heap update modality it follows that $h'[s(x) := n], s' \models p$, that is, $h'[s(x) := n], s \models p$ (since y does not occur in p). Since $h'[s(x) := n]$ and $h[s(x) := \bot]$ are disjoint, we derive from the assumption $h[s(x) := \bot], s \models p \mathrel{-\!\!*} q$ that $h[s(x) := \bot] \uplus h'[s(x) := n], s \models q$. Again by the semantics of the heap update modality we have that $h \uplus h', s' \models [\langle x\rangle := y]q$ iff $(h \uplus h')[s(x) := n], s' \models q$ (that is, $(h \uplus h')[s(x) := n], s \models q$, because y does not occur in q). We then can conclude this case by the observation that $(h \uplus h')[s(x) := n] = h[s(x) := \bot] \uplus h'[s(x) := n]$. □

We denote by **E** the *rewrite system* obtained from the equivalences **E1-16** by orienting these equivalences from left to right, e.g., equivalence **E1** is turned into

a rewrite rule $[S]\mathbf{false} \Rightarrow \mathbf{false}$. The following theorem states that the rewrite system **E** is complete, that is, confluent and strongly normalizing. Its proof is straightforward (using standard techniques) and therefore omitted.

Theorem 4.4.5 (Completeness of **E**)**.**

- **Normal form.** *Every standard formula of separation logic is in normal form (which means that it cannot be reduced by the rewrite system* **E***).*
- **Local confluence.** *For any two reductions $p \Rightarrow q_1$ and $p \Rightarrow q_2$ (p a formula of DSL) there exists a DSL formula q such that $q_1 \Rightarrow q$ and $q_2 \Rightarrow q$.*
- **Termination.** *There does not exist an infinite chain of reductions $p_1 \Rightarrow p_2 \Rightarrow p_3 \Rightarrow \cdots$.*

We now show an example of the interplay between the modalities for heap update and heap clear. We want to derive

$$\{\forall x((x \not\hookrightarrow -) \rightarrow p)\}\ x := \mathbf{new}(0); \mathbf{delete}(x)\ \{p\}$$

where statement $x := \mathbf{new}(0); \mathbf{delete}(x)$ simulates the so-called random assignment [107]: the program terminates with a value of x that is chosen non-deterministically. First we apply the axiom **E8** for de-allocation to obtain

$$\{(x \hookrightarrow -) \wedge [\langle x \rangle := \bot]p\}\ \mathbf{delete}(x)\ \{p\}.$$

Next, we apply the axiom **E8** for allocation to obtain

$$\begin{gathered}\{\forall x((x \not\hookrightarrow -) \rightarrow [\langle x \rangle := 0]((x \hookrightarrow -) \wedge [\langle x \rangle := \bot]p))\} \\ x := \mathbf{new}(0) \\ \{(x \hookrightarrow -) \wedge [\langle x \rangle := \bot]p\}.\end{gathered}$$

Applying **E10** (after pushing the heap update modality inside), followed by some basic first-order reasoning, we can reduce $[\langle x \rangle := 0](\exists y(x \hookrightarrow y))$ to true. So we obtain

$$\begin{gathered}\{\forall x((x \not\hookrightarrow -) \rightarrow [\langle x \rangle := 0][\langle x \rangle := \bot]p)\} \\ x := \mathbf{new}(0) \\ \{(x \hookrightarrow -) \wedge [\langle x \rangle := \bot]p\}.\end{gathered}$$

In order to proceed we formalize the interplay between the modalities for heap update and heap clear by the following general equivalence:

$$[\langle x \rangle := e][\langle x \rangle := \bot]p \equiv [\langle x \rangle := \bot]p$$

We then complete the proof by applying the sequential composition rule and consequence rule, using the above equivalence and the following axiomatization of the heap clear modality:

$$(x \not\hookrightarrow -) \wedge [\langle x \rangle := \bot]p \equiv (x \not\hookrightarrow -) \wedge p$$

Now it is possible to define meta-operations on formulas of separation logic ϕ.

Definition 4.4.6. We define the meta-operations $\phi[\langle x\rangle := y]$ and $\phi[\langle x\rangle := \bot]$:

$$\phi[\langle x\rangle := y] = [\langle x\rangle := y]\phi,$$

and

$$\phi[\langle x\rangle := \bot] = [\langle x\rangle := \bot]\phi.$$

Note that due to Theorem 4.4.5, we can completely eliminate the modality when it is applied to a formula of separation logic. Hence the resulting formulas are again formulas of separation logic, and no longer in the extended language of DSL.

The above axiomatization can be extended in the standard manner to a program logic for sequential while programs, see [107], which does not require the frame rule, nor any other adaptation rule besides the consequence rule. For recursive programs however one does need more adaptation rules: a further discussion about the use of the frame rule in a relative completeness proof for recursive pointer programs is outside the scope of this thesis, and left for future work.

4.5 Alternative axiomatizations

Based on the heap update and heap clear pseudo-instructions of the previous section, we can give two alternative axiomatizations of Reynolds' logic. It is remarkable that these alternative axiomatizations can be proven to be also the weakest preconditions, respectively strongest postconditions, of the primitive operations of pointer programs.

Definition 4.5.1. The set of *alt-backwards* axioms is:

(assign) $\vdash \{p[x := y]\}\ x := y\ \{p\}$,

(lookup) $\vdash \{\exists z((y \hookrightarrow z) \wedge \phi[x := z])\}\ x := [y]\ \{\phi\}$ where z is fresh,

(mutation) $\vdash \{(x \hookrightarrow -) \wedge \phi[\langle x\rangle := y]\}\ [x] := y\ \{\phi\}$,

(allocation) $\vdash \{\forall x((x \not\hookrightarrow -) \to \phi[\langle x\rangle := y])\}\ x := \mathbf{new}(y)\ \{\phi\}$ where $x \neq y$,

(deallocation) $\vdash \{(x \hookrightarrow -) \wedge \phi[\langle x\rangle := \bot]\}\ \mathbf{delete}(x)\ \{\phi\}$.

These alternative backwards axioms also express the weakest precondition:

Lemma 4.5.2. *The* alt-backwards *axioms are sound with respect to* **MSL***, and describe the weakest precondition with respect to the given primitive operations and postcondition.*

We can also give the set of *alt-forwards* axioms, in the sense that these axioms allow reasoning forwards from a given precondition.

Definition 4.5.3. The set of *alt-forwards* axioms is:

(assign) $\vdash \{\phi\}\ x := y\ \{\exists z(\phi[x := z] \wedge y[x := z] = x)\}$,

(lookup) $\vdash \{(y \hookrightarrow -) \wedge \phi\}\ x := [y]\ \{\exists z(\phi[x := z] \wedge y[x := z] \hookrightarrow x)\}$,

(mutation) $\vdash \{(x \hookrightarrow -) \wedge \phi\}\ [x] := y\ \{(\exists z(\phi[\langle x\rangle := z])) \wedge (x \hookrightarrow y)\}$,

(allocation) $\vdash \{\phi\}\ x := \mathbf{new}(y)\ \{(\exists z(\phi[x := z]))[\langle x\rangle := \bot] \wedge (x \hookrightarrow y)\}$,

(deallocation) $\vdash \{(x \hookrightarrow -) \wedge \phi\}\ \mathbf{delete}(x)\ \{\exists z(\phi[\langle x\rangle := z]) \wedge (x \not\hookrightarrow -)\}$,

where z is fresh.

Lemma 4.5.4. *The* alt-forwards *axioms are sound with respect to* **MSL***, and describe the strongest postcondition with respect to the given primitive operations and precondition.*

Another application of the modality for the heap update and heap clear pseudo-instructions is that we are able to prove the completeness of the local axioms of Reynolds' logic and the frame rule, without employing the separating implication as the invariant formula.

Theorem 4.5.5 (Completeness local axioms)**.** *For any primitive pointer operation S, if $\models \{p\}\ S\ \{q\}$ then $\{\phi\}\ S\ \{\psi\}$ is derivable from the* local *axioms and the frame rule, the consequence rule, and the invariance rule for basic assignment and look-up.*

Proof. Let $\models \{\phi\}\ S\ \{\psi\}$.

Basic assignment By the invariance rule for basic assignments, we first derive

$$\{\mathbf{true} \wedge \exists x(\phi)\}\ x := e\ \{x = e \wedge \exists x(\phi)\}$$

Clearly, ϕ implies $\exists x(\phi)$. Let $h, s \models x = e$, that is, $s(x) = s(e)$, and $h, s[x := n] \models \phi$, for some n. From the assumption $\models \{\phi\}\ x := e\ \{\psi\}$ we then derive $h, s[x := s[x := n](e)] \models \psi$, that is, $h, s \models \psi$ (since $s[x := n](e) = s(e) = s(x)$).

Look-up By the restricted invariance rule, we first derive

$$\{\exists x(\phi) \wedge (e \hookrightarrow -)\}\ x := [e]\ \{\exists x(\phi) \wedge (e \hookrightarrow x)\}$$

Since $\models \{\phi\}\ x := [e]\ \{\psi\}$, we have that ϕ implies $e \hookrightarrow -$, and so ϕ implies $\exists x(\phi) \wedge (e \hookrightarrow -)$. On the other hand, let $h(s(e)) = s(x)$ and $h, s' \models \phi$, where $s' = s[x := n]$, for some n. From the assumption $\models \{\phi\}\ x := [e]\ \{\psi\}$ we then derive $h, s[x := h(s'(e))] \models \psi$, that is, $h, s \models \psi$ (since x does not occur in e and $h(s(e)) = s(x)$, we have that $s[x := h(s'(e))] = s[x := h(s(e))] = s$).

Mutation Let ϕ' denote $\exists y(\phi[\langle x\rangle := y])$. By the frame rule, we first derive

$$\{(x \mapsto -) * \phi'\}\ [x] := e\ \{(x \mapsto e) * \phi'\}$$

Let $h, s \models \phi$. We show that $h, s \models (x \mapsto -) * \phi'$: Since $\models \{\phi\}\ [x] := e\ \{\psi\}$ we have that $s(x) \in dom(h)$. So we can introduce the split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto -$ and $h_2 = h[s(x) :=\bot]$. By the above substitution lemma it then suffices to observe that $h_2, s[y := h(s(x))] \models \phi[\langle x\rangle := y]$ if and only if $h_2[s(x) := h(s(x))], s \models \phi$ (y does not appear in ϕ), that is, $h, s \models \phi$. On the other hand, we have that $(x \mapsto e) * \phi'$ implies ψ: Let $h, s \models (x \mapsto e) * \phi'$. So there exists a split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto e$ and $h_2, s \models \phi'$. Let n be such that $h_2, s[y := n] \models \phi[[x] := y]$. By the above substitution lemma we have that $h_2, s[y := n] \models \phi[\langle x\rangle := y]$ if and only if $h_2[s(x) := n], s \models \phi$ (y does not appear in ϕ). Since $\models \{\phi\}\ [x] := e\ \{\psi\}$ it then follows that $h_2[s(x) := s(e)], s \models \psi$, that is, $h, s \models \psi$ (note that $h = h_2[s(x) := s(e)]$ because $h(s(x)) = s(e)$ and $h_2 = h[s(x) := \bot]$).

Allocation By the frame rule, we first derive

$$\{\mathbf{emp} * \exists x(\phi)\}\ x := \mathbf{new}(e)\ \{(x \mapsto e) * \exists x(\phi)\}$$

Clearly, ϕ implies $\mathbf{emp} * \exists x(\phi)$. On the other hand, let $h, s \models (x \mapsto e) * \exists x(\phi)$. So there exists a split $h = h_1 \uplus h_2$ such that $h_1, s \models x \mapsto e$ and $h_2, s[x := n] \models \phi$, for some n. Since $\models \{\phi\}\ x := \mathbf{new}(e)\ \{\psi\}$, we derive that $h_2[s(x) := s[x := n](e)], s \models \psi$, that is, $h, s \models \psi$ (note that $s(x) \notin dom(h_2)$ and, since x does not appear in e, we have $s[x := n](e) = s(e)$, and thus $h = h_2[s(x) := s(e)]$).

Dispose Let ϕ' denote $(x \not\mapsto -) \wedge \exists y(\phi[\langle x\rangle := y])$. By the frame rule, we first derive

$$\{(x \mapsto -) * \phi'\}\ [x] := \bot\ \{\mathbf{emp} * \phi'\}$$

See above (mutation) for the kind of argument that establishes that ϕ implies $(x \mapsto -) * \phi'$. On the other hand, Let $h, s \models \mathbf{emp} * \phi'$, that is, $h, s \models \phi'$, and so by the above substitution lemma, we have $h[s(x) := n], s \models \phi$, for some n (again, y does not appear in ϕ). Since $\{\phi\}\ [x] := \bot\ \{\psi\}$, we derive $h[s(x) := \bot], s \models \psi$, that is, $h, s \models \psi$, since $h, s \models x \not\mapsto -$. □