Backwards and Forwards with Separation Logic

Callum Bannister[1,2], Peter Höfner[1,2], and Gerwin Klein[1,2]

[1] Data61, CSIRO, Sydney, Australia
[2] Comp. Sci. and Engineering, University of New South Wales, Sydney, Australia

Abstract. The use of Hoare logic in combination with weakest preconditions and strongest postconditions is a standard tool for program verification, known as backward and forward reasoning. In this paper we extend these techniques to allow backward and forward reasoning for separation logic. While the former is derived directly from the standard operators of separation logic, the latter uses a new one. We implement our framework in the interactive proof assistant Isabelle/HOL, and enable automation with several interactive proof tactics.

1 Introduction

The use of Hoare logic [21,19] in combination with weakest preconditions [16] and strongest postconditions [19] is a standard tool for program verification, known as backward and forward reasoning. These techniques are supported by numerous tools, e.g. [37,36,6,1,33].

Although backward reasoning with weakest preconditions is more common in practice, there are several applications where forward reasoning is more convenient, for example, for programs where the precondition is 'trivial', and the postcondition either too complex or unknown. Moreover, "calculating strongest postconditions by symbolic execution provides a smooth transition from testing to full verification: by weakening the initial precondition one can make the verification cover more initial states". [20]

Hoare logic lacks expressiveness for mutable heap data structures. To overcome this deficiency, based on work of Burstall [8], Reynolds, O'Hearn and others developed separation logic for reasoning about mutable data structures [38,40]. Separation logic allows for local reasoning by splitting memory into two halves: the part the program interacts with, and the part which remains untouched, called the *frame*.

The contribution of this paper is two-fold:

(i) Generic techniques for *backward and forward reasoning in separation logic.* A kind of backward reasoning was already established by Reynolds [40]. Although he states that for each command rules for backward reasoning "can be given", he only lists rules for the assignment of variables (called mutation in [40]), and for the deallocation of memory. Reynolds does not present a general framework that can transform any given Hoare triple specification – enriched with a frame – into a rule that is ready to be used for backward reasoning. We present such a general framework. Since it is based on separation

algebras [10] it not only applies to the standard heap model of separation logic, but to all instances of this algebra.
Using similar algebraic techniques we also derive a generic technique for forward reasoning in separation logic. To achieve this we introduce a new operator, separating coimplication, which algebraically completes the set of the standard operators of separating conjunction, separating implication, and septraction. To the best of our knowledge, we are the first who provide a technique for strongest postconditions in separation logic.

(ii) *Proof tactics* for the developed techniques in Isabelle/HOL [37].
To increase automation for both backward and forward reasoning we mechanise this framework in the interactive proof assistant Isabelle/HOL and provide automated proof tactics. In particular, we provide tactics that make it manageable to interactively reason about the separating implication, which is widely considered unwieldy [5,32].

To show feasibility of our techniques we not only present standard examples such as list reversal, but also look at a larger case study: a formally verified initialiser for component-based systems built on the formally verified seL4 microkernel [28]. A proof of this initialiser using 'standard' manual separation logic reasoning can be found in the literature [7]. Redoing parts of this proof illustrates the strength of our tactics, gives an indication of how much automation they achieve, and shows by how much they reduce manual proof effort.

2 Notation

In this section we present the notation of Isabelle/HOL [37] that deviate from standard mathematical notation.

We denote the space of total functions by $\Rightarrow$, and write type variables as `'a`, `'b`, etc. The `option` type

```
datatype 'a option = None | Some 'a
```

adjoins a new element `None` to type `'a`. Hence `'a option` models partial functions.

Separation logic assertions typically are functions from the state to bool, i.e., `'s ⇒ bool`. We lift the standard logical connectives $\wedge$, $\vee$, $\neg$, and $\longrightarrow$ point-wise to the function space in the spirit of standard separation logic, e.g. `(P ⟹ Q) = (∀s. P s ⟶ Q s)`.

For the example programs in this paper, we use a *deterministic* state monad. Since we are interested in distinguishing failed executions we add a flag in the style of other monadic Hoare logic frameworks [14]. This means, a program execution has the type `'s ⇒ 'r × 's × bool`, i.e., a function that takes a state and returns as result a new state, and a flag indicating whether the execution was successful (`true`) or not (`false`). Sequential composition, denoted by $\ggg$, is defined as

```
f ⋙= g ≡
λs. let (r', s', c) = f s; (r'', s'', c') = g r' s'
    in (r'', s'', c ∧ c')
```

Since our theory is based on abstract separation algebra (see below), we can change the underlying monad without problems. In particular we use both a *nondeterministic* state monad and an error monad for our case study.
For larger programs we use `do`-notation for sequential composition, e.g.

```
do { x ← f; g x; h x }
```

3 Hoare Logic and Separation Logic

Hoare logic or *Floyd-Hoare logic* [21,19] is the standard logic for program analysis, based on the eponymous *Hoare triple*: $\{\!|P|\!\}\ \mathtt{m}\ \{\!|Q|\!\}$ (originally denoted by `P {m} Q`), where `P` and `Q` are assertions, called pre- and postcondition respectively, and `m` is a program or command.

Initially, Hoare logic considered *partial correctness* only [21], ignoring termination. In our monadic context, where we identify non-termination and failed execution, this translates to

$$\{\!|P|\!\}\ \mathtt{m}\ \{\!|Q|\!\} \equiv \forall \mathtt{h}.\ \mathtt{P}\ \mathtt{h} \longrightarrow (\mathsf{let}\ (\mathtt{r'},\ \mathtt{h'},\ \mathtt{c}) = \mathtt{m}\ \mathtt{h}\ \mathsf{in}\ \mathtt{c} \longrightarrow \mathtt{Q}\ \mathtt{r'}\ \mathtt{h'})$$

If the precondition `P` holds before the execution of `m`, *and* `m` terminates successfully (the flag `c` is true) then the postcondition `Q` holds afterwards. Successful termination needs to be proven separately. If `m` fails to terminate successfully under `P`, i.e., by non-termination or other program failure, then any postcondition forms a valid Hoare triple.

Total correctness combines termination with correctness.

$$\{\!|P|\!\}\ \mathtt{m}\ \{\!|Q|\!\}_t \equiv \forall \mathtt{h}.\ \mathtt{P}\ \mathtt{h} \longrightarrow (\mathsf{let}\ (\mathtt{r'},\ \mathtt{h'},\ \mathtt{c}) = \mathtt{m}\ \mathtt{h}\ \mathsf{in}\ \mathtt{Q}\ \mathtt{r'}\ \mathtt{h'} \wedge \mathtt{c})$$

For total correctness, whenever `P` holds, `m` *will* terminate successfully, and the result satisfies `Q`.

Example 1. Assume the function `delete_ptr p`, which clears the allocated memory pointed to by `p`, and fails if `p` does not point to any location at all or to an address outside the current heap.

Let `emp` be the empty heap. Then the triple $\{\!|\mathtt{p} \mapsto _|\!\}$ `delete_ptr p` $\{\!|\mathtt{emp}|\!\}$ describes the situation where the heap has a single location `p`, and is otherwise empty.[3] After succesful termination the heap is empty.

However, this specification is limiting since it only allows one particular heap configuration as precondition. Consider two further scenarios, namely heap configurations where `p` does not point to any location in the heap (e.g. the empty heap), and heap configurations with additional memory.

In the first scenario, `delete_ptr p` fails. Hence $\{\!|\mathtt{emp}|\!\}$ `delete_ptr p` $\{\!|Q|\!\}$ would hold under partial correctness for any `Q`, but not under total correctness. In the second scenario, with additional memory, that additional memory remains unchanged during the execution of `delete_ptr p`. This is the case separation logic deals with. □

[3] We will explain the heap model in detail later in this section.

Separation logic (SL) (e.g. [40]) extends Hoare logic by assertions to express separation between memory regions, which allows reasoning about mutable data structures. It is built around *separating conjunction* `_*_`, which asserts that a heap can be split into two disjoint parts where its two argument predicates hold.

The usual convention in SL is to require that even in partial correctness the program is *abort*-free, in particular for pointer access. The semantics of our slightly more traditional setting does not distinguish between non-termination and failure. Hence partial correctness will not guarantee pointer safety, while total correctness will.

A standard ingredient of SL is the *frame rule*

$$\frac{\{\!|\mathtt{P}|\!\}\ \mathtt{m}\ \{\!|\mathtt{Q}|\!\}}{\{\!|\mathtt{P} * \mathtt{R}|\!\}\ \mathtt{m}\ \{\!|\mathtt{Q} * \mathtt{R}|\!\}}$$

The rule asserts that a program `m` that executes correctly in a state with a small heap satisfying its precondition `P`, with postcondition `Q`, can also execute in any state with a larger heap (satisfying `P * R`) and that the execution will not affect the additional part of the state. Traditionally, it requires as side condition that no variable occurring free in `R` is modified by `m`. In our shallow monadic setting, no such variables exist and hence no side condition is required. We differ from tradition by proving that the frame rule holds for particular specifications rather than over the program syntax as a whole. This allows us to talk about programs that are not *strictly* local, but may be local with regards to a particular precondition. When local specifications are given for the primitive operations of a program, it is easy to compose them to show the locality of larger programs.

SL can be built upon separation algebras, which are commutative partial monoids [10]. Such algebras offer a binary operation `+` and a neutral element `0`, such that whenever `x + y` is defined, `+` is commutative and associative, and `x + 0 = x`. Our automation framework is built upon an existing Isabelle/HOL framework [29,30], which uses a total function `+` together with another commutative operation `##` that weakly distributes over `+` [29], and expresses the aforementioned disjointness.

Using these operations, separating conjunction is defined as

$$\mathtt{P} * \mathtt{Q} \equiv \lambda \mathtt{h}.\ \exists \mathtt{h}_1\ \mathtt{h}_2.\ \mathtt{h}_1\ \#\#\ \mathtt{h}_2 \wedge \mathtt{h} = \mathtt{h}_1 + \mathtt{h}_2 \wedge \mathtt{P}\ \mathtt{h}_1 \wedge \mathtt{Q}\ \mathtt{h}_2 \tag{1}$$

which implies associativity and commutativity of `*`.

The standard model of SL, and separation algebra, uses heaps. The term $(\mathtt{p} \mapsto \mathtt{v})\ \mathtt{h}$ indicates that the pointer `p` on heap `h` is allocated and points to value `v`. The term $\mathtt{p} \mapsto _$ indicates an arbitrary value at location `p`.

A heap is a partial function from addresses (pointers) to values. The operation $\mathtt{h}_1\ \#\#\ \mathtt{h}_2$ checks whether the domains of $\mathtt{h}_1$ and $\mathtt{h}_2$ are disjoint. When $\mathtt{h}_1\ \#\#\ \mathtt{h}_2$ evaluates to true, $\mathtt{h}_1 + \mathtt{h}_2$ 'merges' the heaps by forming their union. The formal definitions are straightforward and omitted here.

In separation algebras, the operations `##` and `+` define a partial order, which formalises subheaps:

$$\mathtt{h}_1 \preceq \mathtt{h} \equiv \exists \mathtt{h}_2.\ \mathtt{h}_1\ \#\#\ \mathtt{h}_2 \wedge \mathtt{h}_1 + \mathtt{h}_2 = \mathtt{h}$$

SL usually leads to simple proofs of pointer manipulation for data structures. Classical examples of such data structures are singly- and doubly-linked lists, trees, as well as directed acyclic graphs (DAGs) [39,22].

Separating implication $\mathtt{P} \longrightarrow\!\!* \; \mathtt{Q}$, also called *magic wand*, is another operator of SL. When applied to a heap $\mathtt{h}$ it asserts that extending $\mathtt{h}$ by a disjoint heap, satisfying $\mathtt{P}$, guarantees that $\mathtt{Q}$ holds on the combined heap:

$$\mathtt{P} \longrightarrow\!\!* \; \mathtt{Q} \equiv \lambda \mathtt{h}.\ \forall \mathtt{h}_1.\ \mathtt{h}\ \#\#\ \mathtt{h}_1 \wedge \mathtt{P}\ \mathtt{h}_1 \longrightarrow \mathtt{Q}\ (\mathtt{h} + \mathtt{h}_1) \tag{2}$$

Ishtiaq and O'Hearn use this operator for reasoning in the presence of sharing [25].

The operations $*$ and $\longrightarrow\!\!*$ are lower and upper adjoints of a Galois connection, e.g. [15]. This relationship implies useful rules, like currying $(\mathtt{P} * \mathtt{Q} \Longrightarrow \mathtt{R}) \Longrightarrow (\mathtt{P} \Longrightarrow \mathtt{Q} \longrightarrow\!\!* \; \mathtt{R})$, decurrying $(\mathtt{P} \Longrightarrow \mathtt{Q} \longrightarrow\!\!* \; \mathtt{R}) \Longrightarrow (\mathtt{P} * \mathtt{Q} \Longrightarrow \mathtt{R})$, and modus ponens $\mathtt{Q} * (\mathtt{Q} \longrightarrow\!\!* \; \mathtt{P}) \Longrightarrow \mathtt{P}$. As we will see, separating implication is useful for backward reasoning.

The literature uses another 'basic' operator of SL, *septraction* [43]:

$$\mathtt{P} -\!\!\circledast\ \mathtt{Q} \equiv \lambda \mathtt{h}.\ \exists \mathtt{h}_1.\ \mathtt{h}_1\ \#\#\ \mathtt{h} \wedge \mathtt{P}\ \mathtt{h}_1 \wedge \mathtt{Q}\ (\mathtt{h}_1 + \mathtt{h}) \tag{3}$$

It is the dual of separating implication, i.e., $\mathtt{P} -\!\!\circledast\ \mathtt{Q} = \neg(\mathtt{P} \longrightarrow\!\!* \; \neg\mathtt{Q})$, and expresses that the heap can be extended with a state satisfying $\mathtt{P}$, so that the extended state satisfies $\mathtt{Q}$. Septraction plays a role in combining SL with rely-guarantee reasoning [43], and for shared data structures such as DAGs [22].

4 Separating Coimplication

While separating conjunction, separating implication, and septraction, as well as their relationships to each other are well studied and understood, one operation is missing in SL.

We define *separating coimplication*, denoted by $\leadsto\!*$, as

$$\mathtt{P} \leadsto\!* \; \mathtt{Q} \equiv \lambda \mathtt{h}.\ \forall \mathtt{h}_1\ \mathtt{h}_2.\ \mathtt{h}_1\ \#\#\ \mathtt{h}_2 \wedge \mathtt{h} = \mathtt{h}_1 + \mathtt{h}_2 \wedge \mathtt{P}\ \mathtt{h}_1 \longrightarrow \mathtt{Q}\ \mathtt{h}_2 \tag{4}$$

It states that whenever there is a subheap $\mathtt{h}_1$ satisfying $\mathtt{P}$ then the remaining heap satisfies $\mathtt{Q}$. To the best of our knowledge, we are the first to define this operator and explore its properties.

It is the dual of separating conjunction, i.e., $\mathtt{P} \leadsto\!* \; \mathtt{Q} = \neg(\mathtt{P} * \neg\mathtt{Q})$, which is the same relationship as the one between separating implication and septraction.

Special instances of $\leadsto\!*$ (in the form of doubly negated conjunction) appear in the literature: the *dangling operator* of Vafeiadis and Parkinson [43] uses subterms of the form $\neg(\mathtt{p} \mapsto _ * \mathtt{True})$, which equals $\mathtt{p} \mapsto _ \leadsto\!* \; \mathtt{False}$, and the *subtraction operator* by Calcagno et al. [9], used for comparing bi-abduction solutions, uses terms of the form $\mathtt{P} \leadsto\!* \; \mathtt{emp}$. These occurrences indicate that separating coimplication is an important, yet unexplored operator for SL. As we will show, it is also the crucial ingredient to set up forward reasoning for SL.

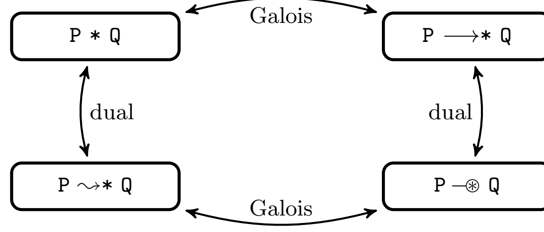

Fig. 1. Relationship between operators of separation logic

Separating coimplication forms a Galois connection with septraction. Therefore, many useful theorems follow from abstract algebraic reasoning. For example, similar to the rules stated above for $*$ and $\longrightarrow*$, we get rules for currying, decurrying and cancellation:

$$\frac{\texttt{P} \mathbin{-\!\circledast} \texttt{Q} \Longrightarrow \texttt{R}}{\texttt{Q} \Longrightarrow \texttt{P} \rightsquigarrow* \texttt{R}}\ (\textsc{curry}) \qquad \frac{\texttt{Q} \Longrightarrow \texttt{P} \rightsquigarrow* \texttt{R}}{\texttt{P} \mathbin{-\!\circledast} \texttt{Q} \Longrightarrow \texttt{R}}\ (\textsc{decurry}) \qquad \frac{\texttt{Q} \mathbin{-\!\circledast} (\texttt{Q} \rightsquigarrow* \texttt{P})}{\texttt{P}}\ (\textsc{canc})$$

It follows that separating coimplication is isotone in one, and antitone in the other argument:

$$\frac{\texttt{P'} \Longrightarrow \texttt{P}}{\texttt{P} \rightsquigarrow* \texttt{Q} \Longrightarrow \texttt{P'} \rightsquigarrow* \texttt{Q}} \qquad \frac{\texttt{Q} \Longrightarrow \texttt{Q'}}{\texttt{P} \rightsquigarrow* \texttt{Q} \Longrightarrow \texttt{P} \rightsquigarrow* \texttt{Q'}}$$

Separating coimplication is not only interesting because it completes the set of 'nicely' connected operators for SL (see Fig. 1), it is also useful to characterise specific heap configurations. For example, $(\texttt{P} \rightsquigarrow* \texttt{False})\ \texttt{h}$ states that no subheap of h satisfies P: $\texttt{P} \rightsquigarrow* \texttt{False} = \lambda \texttt{h}.\ \forall \texttt{h}_1.\ \texttt{h}_1 \preceq \texttt{h} \longrightarrow \neg\ \texttt{P}\ \texttt{h}_1$.

While properties concerning $\rightsquigarrow*$ and $-\!\circledast$ mostly follow from the Galois connection, some need to be derived 'manually':

$$\frac{\texttt{P} \rightsquigarrow* \texttt{Q} \qquad \texttt{P} * \texttt{R}}{\texttt{P} * (\texttt{Q} \land \texttt{R})} \qquad \frac{\texttt{P} \longrightarrow* (\texttt{R} \land (\texttt{P} \rightsquigarrow* \texttt{False}))}{\neg\texttt{P} * (\texttt{P} \longrightarrow* \neg\texttt{R})}$$

The first rule states that whenever a heap satisfies $\texttt{P} \rightsquigarrow* \texttt{Q}$, and can be split into two subheaps satisfying P and R, respectively, then the subheap satisfying R has to satisfy Q as well. The second rule connects separating implication and coimplication directly and states that if adding a heap satisfying P yields a heap with no subheap containing P, then the underlying heap cannot satisfy P.

SL considers different classes of assertions [40]; each of them plays an important role in SL, and usually gives additional information about the heap. For example, a *precise assertion* characterises a unique heap portion (when such a portion exists), i.e.,

$$\texttt{precise P} \equiv \forall \texttt{h}\ \texttt{h}_1\ \texttt{h}_2.\ \texttt{h}_1 \preceq \texttt{h} \land \texttt{P}\ \texttt{h}_1 \land \texttt{h}_2 \preceq \texttt{h} \land \texttt{P}\ \texttt{h}_2 \longrightarrow \texttt{h}_1 = \texttt{h}_2$$

P is precise iff the distributivity $\forall \texttt{Q R}.\ ((\texttt{Q} \land \texttt{R}) * \texttt{P}) = (\texttt{Q} * \texttt{P} \land \texttt{R} * \texttt{P})$ holds. [15] Separating coimplication yields a nicer characterisation:

$$\texttt{precise P} = \forall \texttt{R}.\ \texttt{P} * \texttt{R} \Longrightarrow (\texttt{P} \rightsquigarrow* \texttt{R}) \qquad (5)$$

On the one hand this equivalence eliminates one of the ∀-quantifiers, which simplifies reasoning; on the other hand it directly relates separating conjunction with coimplication, stating that if P and R hold on a heap, and one pulls out an arbitrary subheap satisfying P, the remaining heap must satisfy R. Obviously, this relationship between * and $\leadsto$* does not hold in general since separating coimplication may pull out the 'wrong' subheap satisfying P.

As a consequence, using (CANC), we immediately get

$$\frac{\texttt{precise P}}{\texttt{P} \multimap\!\!\circledast \texttt{ (P * R)} \Longrightarrow \texttt{R}} \tag{6}$$

Our Isabelle files [3] contain many more properties of separating coimplication. The most important use of separating coimplication, however, is its application in forward reasoning, as we will demonstrate in Sect. 6.

Example 2. Using separating coimplication we can fully specify `delete_ptr p` in a way that matches intuition: ⦃p ↦ _ $\leadsto$* R⦄ delete_ptr p ⦃R⦄. This rule states that the final state should satisfy R, when the pointer is deleted, and the pointer existed in the first place. □

5 Walking Backwards

Backward reasoning [16] or *reasoning in weakest-precondition style* proceeds backwards from a given postcondition Q and a given program m by determining the *weakest* precondition wp(m,Q) such that ⦃wp(m,Q)⦄ m ⦃Q⦄ is a valid Hoare triple.

Backward reasoning is well established for formal programming languages, using classical logics. For example the weakest precondition wp(m_1;m_2,Q) for a sequential program equals wp(m_1,wp(m_2,Q)); the full set goes back to Dijkstra [16]. Using these equations, backward reasoning in Hoare logic is straightforward.

Avoiding Frame Calculations. In SL, however, it comes at a price, since reasoning has to work on Hoare triples of the form ⦃P * R⦄ m ⦃Q * R⦄ and has to consider the frame. Whenever an arbitrary postcondition X is given, one needs to split it up into the actual postcondition Q needed for reasoning about m, and the (untouched) frame R. That means for given X and Q one has to *calculate the frame* R such that X = Q * R. Frame calculations are often challenging in applications since X can be arbitrary complex. The same holds for a given precondition.

Example 3. Let `copy_ptr p p' = do { x ← get_ptr p; set_ptr p' x }` be the program that copies the value at pointer p to the value at pointer p'. Its natural specification is ⦃p ↦ x * p' ↦ _⦄ copy_ptr p p' ⦃p ↦ x * p' ↦ x⦄. The specification we use is

$$\forall \texttt{R.}\ \{\!|\texttt{p} \mapsto \texttt{x * p'} \mapsto \texttt{_ * R}|\!\}\ \texttt{copy_ptr p p'}\ \{\!|\texttt{p} \mapsto \texttt{x * p'} \mapsto \texttt{x * R}|\!\} \tag{7}$$

In a larger program, the postcondition at the call site of copy_ptr will be more complex than Q = p ↦ x * p' ↦ x. Say it is ⦃p' ↦ v * a ↦ _ * p ↦ v * R'⦄, for some heap R'. To determine the precondition, using Rule (7), the postcondition needs to be in the form Q * R. One has to calculate the frame R = a ↦ _ * R'. □

Phrasing specifications in the form $\{\!|P * R|\!\}\ m\ \{\!|Q * R|\!\}$ (similar to Rule (7)) state that the frame rule holds for program m, i.e., that m only consists of local actions with respect to P. In the monadic setting, where not all programs are necessarily local, we find this form more convenient than a predicate on the programs and a separate frame rule. That also means that our Isabelle/HOL framework does not rely on the frame rule.

In the previous example the frame calculation uses only associativity and commutativity of *, but in general such calculations can be arbitrarily complex. A solution to this problem follows directly from the Galois connection and 'rewrites' the pre- and postcondition.

$$(\forall R.\ \{\!|P * R|\!\}\ m\ \{\!|Q * R|\!\}) = (\forall X.\ \{\!|P * (Q \longrightarrow\!* X)|\!\}\ m\ \{\!|X|\!\}) \tag{8}$$

The left-hand side coincides with the form we use to specify our programs. The right-hand side has the advantage that it works for any postcondition X; no explicit calculation of the frame is needed for the postcondition. Since $Q \longrightarrow\!* X$ is the *weakest* [9] choice of frame, the calculation happens implicitly and automatically in the precondition. This is a generalisation of what occurs in Reynolds' work [40] for specific operations.

Since Rule (8) generates Hoare triples that can be applied to arbitrary postconditions, we can use these rules directly to perform backward reasoning in the sense of Dijkstra [16]. That means that our calculations are similar to the classical ones for reasoning with weakest preconditions, e.g. $wp(m_1, wp(m_2, Q))$. As a consequence our framework can generate preconditions fully automatically. As in the classical setting, applying the rules of Hoare logic is now separated from reasoning about the content of the program and the proof engineer can focus their effort on the part that requires creativity.

Example 4. Using Equivalence (8), the specification for copy_ptr (7) becomes

$$\{\!|\exists x.\ p \mapsto x * p' \mapsto _ * (p \mapsto x * p' \mapsto x \longrightarrow\!* X)|\!\}\ \texttt{copy_ptr}\ p\ p'\ \{\!|X|\!\} \qquad \square$$

Simplifying Preconditions. As mentioned above, Equivalence (8) allows us to perform backward reasoning and to generate preconditions. However, the generated formulas will often be large and hence automation for simplifying generated preconditions is necessary. We provide such simplification tactics.

Both the right-hand side of (8) and the previous example show that generated preconditions contain interleavings of * and $\longrightarrow\!*$. A simplifier suitable for our framework has to deal with such interleavings, in particular it should be able to handle formulas of the type $P * (Q \longrightarrow\!* R)$, for any P, Q and R. Two rules that are indispensable here are cancellation and currying, as introduced in Sect. 3:

$$\frac{R \Longrightarrow R'}{P * R \Longrightarrow P * R'} \qquad \frac{P * Q \Longrightarrow R}{P \Longrightarrow Q \longrightarrow\!* R} \tag{9}$$

Currently, not many solvers support the separating implication operator [5,32]. Some automatic solvers for separating implication exist for formulas over a restricted set of predicates [23]. Since we are aiming at a general framework for arbitrary specifications, we do not want to restrict the expressiveness of pre-

and postconditions, and hence we cannot restrict our framework to such subsets. Moreover, we cannot hope to develop fully automatic solvers for the problem at hand at all, since it is undecidable for arbitrary pre- and postconditions [11].

We provide proof tactics for Isabelle/HOL that can simplify formulas of the form `P * (Q ⟶* R)`, for any `P`, `Q` and `R`, and hence can be used in the setting of backward reasoning. Although we cannot expect full automation, the simplification achieved by the tactics is significant, as we will show. Our tactics can make partial progress without fully solving the goal. As experience shows for standard proof methods in Isabelle, this is the most useful kind, e.g. the method `simp`, which rewrites the current goal and leaves a normal form, is much more frequently used than methods such as `blast` or `force` that either have to fully solve the goal or fail, but cannot make intermediate progress available to the user. What we provide is a simplifier, not an entailment solver or semi-solver.

Our framework [3] offers support for backward reasoning in SL, and builds on top of an existing library [30], which is based on separation algebras. This brings the advantage that abstract rules, such as `Q * (Q ⟶* P) ⟹ P`, which are indispensable for handling interleaving of `*` and `⟶*` are immediately available. Since the framework is independent of the concrete heap model, we can apply the tool to a wide range of problem domains. As usual, the tactics enable the user to give guidance to complete proofs where other methods fail, and to substantially reduce proof effort.

– The tactic `sep_wp` performs weakest-precondition reasoning on monads and automatically transforms specification Hoare triples provided as arguments into weakest-precondition format, using Equivalence (8). In addition to the transformations already described, it can also handle further combinations, e.g. with classical Hoare logic, or instances where the separation logic only operates on parts of the monad state. We integrate `sep_wp` into the existing tactic `wp` [14] of the seL4 proofs, which implements classical weakest-precondition reasoning with additional features such as structured decomposition of postconditions. The user sees a tactic that can handle both, SL and non-SL goals, gracefully.
– We develop the tactic `sep_mp` to support reasoning about separating implication, and `sep_lift` to support the currying rule of Sect. 3, eliminating separating implication. These are both integrated into the existing `sep_cancel` [30] method, for reducing formulas by means of cancellation rules.

Detailed Example. To illustrate backward reasoning in SL in more detail, we show the correctness of the program `swap_ptr p p'` that swaps the values `p` and `p'` point. Pointer programs are built from four basic operations that manipulate the heap: `new_ptr` allocates memory for a pointer, `delete_ptr` removes a pointer from the heap, `set_ptr` assigns a value, and `get_ptr` reads a value, respectively. Their specifications are as follows:

```
{|R|} new_ptr {|λrv. rv ↦ _ * R|}
{|p ↦ _ * R|} delete_ptr p {|R|}
{|p ↦ _ * R|} set_ptr p v {|p ↦ v * R|}
{|∃x. p ↦ x * R x|} get_ptr p {|λrv. p ↦ rv * R rv|}
```

```
p ↦ v * p' ↦ v' * R ⟹
∀x. x ↦ _ ⟶*
   (∃pv. p ↦ pv *
       (p ↦ pv  ⟶* x ↦ _ *
        (x ↦ pv  ⟶*
         (∃pv'. p' ↦ pv' *
             (p' ↦ pv'  ⟶* p ↦ _ *
              (p ↦ pv'  ⟶*
               (∃y. x ↦ y *
                   (x ↦ y  ⟶* p' ↦ _ *
                    (p' ↦ y  ⟶* x ↦ _ * p ↦ v' * p' ↦ v * R)))))))))
```

Fig. 2. Backward reasoning: generated proof goal for `swap_ptr`

As before we use specifications with frames, avoiding the use of the frame rule.

Recall that in our monadic setting the postcondition `R` is a predicate over two parameters: the return value `rv` of the function, and the state `s` after termination. When there is no return value (e.g. for `set_ptr`) we omit the first parameter.

Using Rule (8), or the tactic `wp` (which includes `sep_wp`), we transform these specifications into a form to be used in backward reasoning (for partial and total correctness), except `delete_ptr`, which already has the appropriate form.

```
        {|∀x. x ↦ _ ⟶* X x|} new_ptr {|X|}
   {|p ↦ _ * (p ↦ v ⟶* X)|} set_ptr p v {|X|}
{|∃x. p ↦ x * (p ↦ x ⟶* X x)|} get_ptr p {|X|}
```

The program `swap_ptr`, which involves all heap operations, is given as

```
swap_ptr p p' = do {
  np ← new_ptr;
  copy_ptr p np;
  copy_ptr p' p;
  copy_ptr np p';
  delete_ptr np
}
```

where `copy_ptr p p' = do { x ← get_ptr p; set_ptr p' x }`, as before. We use the specifications of the basic operations to prove the specification

```
{|p ↦ v * p' ↦ v' * R|} swap_ptr p p' {|p ↦ v' * p' ↦ v * R|}
```

Using equational reasoning of the form $\texttt{wp(m}_1\texttt{;m}_2\texttt{,Q) = wp(m}_1\texttt{,wp(m}_2\texttt{,Q))}$, and starting from the (given) postcondition our framework automatically derives a precondition `pre`. In case the given precondition `p ↦ v * p' ↦ v' * R` implies `pre`, the specification of `swap_ptr` holds. The proof goal is depicted in Fig. 2.

Our tactics simplify this lengthy, unreadable formula, where major simplifications are based on the aforementioned rules (Eqs. (9)).

The tactic `sep_cancel` is able to simplify the generated goal automatically, but gets stuck at existential quantifiers. Although resolving existential quantifiers cannot be fully automated in general, our framework handles many common situations. The left-hand side of Fig. 3 shows an intermediate step illustrating the state before resolving the last existential quantifier. One of the assumptions

```
p ↦ v' * p' ↦ v' * x ↦ v * R ⟹
∃y. x ↦ y *
   (x ↦ y ⟶* p' ↦ _ *
    (p' ↦ y ⟶* x ↦ _ * p ↦ v' *
     p' ↦ v * R))
```

```
x ↦ v * p ↦ v' * p' ↦ v * R ⟹
x ↦ _ * p ↦ v' * p' ↦ v * R
```

Fig. 3. Matching existential quantifier and eliminating ⟶* for `swap_ptr`

is `x ↦ v` and hence the obvious choice for `y` is `v`. Here, Isabelle's simple existential introduction rule is sufficient to allow `sep_cancel` to perform the match without input. The tactic `sep_cancel` can then solve the proof goal fully automatically; for completeness we show a state where all occurrences of ⟶* have been eliminated.

Case Study: System Initialisation. Boyton et al. [7] present a formally verified initialiser for component-based systems built on the seL4 kernel. The safety and security of software systems depends heavily on their initialisation; if initialisation is broken all bets are off with regards to the system's behaviour. The previous proofs (about $15,000$ lines of proof script) were brittle, often manual, and involved frequent specification of the frame. Despite an early form of `sep_cancel` the authors note that "higher-level automation such as frame computation/matching would have improved productivity" [7].

In contrast to our earlier examples, the initialiser proofs operate on a *non-deterministic* state monad, as well as the *non-deterministic error* state monad. Our tactics required only the addition of two trivial interface lemmas to adapt to this change of computation model, illustrating the genericity of our approach.

Substituting the previous mechanisation with our framework[4] we substantially reduce the proof effort: for commands specified in SL, the calculation of the weakest precondition is automatic, without any significant user interaction. Additionally, we find that calculating the frame indirectly via resolution of separating implications is significantly easier to automate, as the separating implication is the *weakest* choice of solution for in-place frame calculation. The general undecidability of separating implication did not pose a problem.

Figure 4 presents a sample of the entire proof script for an seL4 API function to give an indication of the improvements. For brevity Fig. 4 shortens some of the names in the proof. The separation algebra in this statement lets `*` be used inside larger heap objects, such as specifying the capabilities stored inside a Thread Control Block (TCB) object using `↦c`. The lemma models which seL4 capabilities are available to the user after a `restart` operation.

The left-hand side of Fig. 4 shows the original proof. Each application of an SL specification rule required first a weakening of the postcondition to bring it into the expected form, and often a manual specification of the frame. Not only is this cumbersome and laborious for the proof engineer, it was highly brittle – any change of the functionality or specification requires a new proof.

The right-hand side shows the simplified proof. It shortens eighteen lines of proof script to three, without noticeable increase in prover time. By removing

[4] Updated proofs at `https://github.com/seL4/l4v/tree/seL4-7.0.0/sys-init`.

```
lemma restart_null_wp:
 ⦃(tcb, pop_slot) ↦c NullCap * (tcb, reply_slot) ↦c _ * R⦄
  restart tcb
 ⦃(tcb, reply_slot) ↦c (MRCap tcb) * (tcb, pop_slot) ↦c RCap * R⦄
```

```
apply (clarsimp simp:restart_def)
apply (wp)
apply (rule hoare_strengthen_post)
apply (rule set_cap_wp[where R=
       (tcb, reply_slot) ↦c MRCap tcb * R])
apply (sep_cancel)+
apply (rule hoare_strengthen_post)
apply (rule set_cap_wp[where
            R=(tcb, pop_slot) ↦c _ * R])
apply (sep_cancel)+
apply (rule hoare_strengthen_post)
apply (rule ipc_cancel_ret[where
            R=(tcb, reply_slot) ↦c _ * R])
apply (sep_cancel)+
apply (wp)
apply (clarsimp)
apply (intro conjI impI)
apply (drule opt_cap_sep_imp)
apply (clarsimp)
apply (drule opt_cap_sep_imp)
apply (clarsimp)
done
```

```
apply (clarsimp simp:restart_def)
apply (wp sep_wp:set_cap_wp ipc_cancel_ret)
apply (sep_cancel | simp | safe)+
done
```

Fig. 4. Reducing user steps by a factor of 6 in system initialisation proofs

the manual term specification, the tactics also make the proof more robust to changes – we can rewrite parts of the code, while leaving the proof unchanged.

Another strength is that our tactics are incremental, i.e., we can use them alongside others. In our example we use `safe` and `simp`. This design allows us to attack arbitrary formulas of SL.

6 Walking Forwards

Forward reasoning uses strongest postconditions [19]. It proceeds forwards from a given precondition `P` and a given program `m` by calculating the strongest postcondition `sp(m,P)`. Although backward reasoning with weakest preconditions is more common in practice, there are several applications where forward reasoning is more convenient, for example, for programs where the precondition is 'trivial', and the postcondition either too complex or unknown.

Usually forward reasoning focuses on partial correctness. Recall that we admit memory failures in partial correctness. In larger proofs it is convenient to show absence of failure separately, e.g. during a refinement proof [14], and assume it in multiple partial-correctness proofs, thereby avoiding proof duplication.

To enable forward reasoning for SL it is desirable to transform a Hoare triple $\{\!|\texttt{P * R}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * R}|\!\}$ into the form $\{\!|\texttt{X}|\!\}\ \texttt{m}\ \{\!|\texttt{post}|\!\}$, similar to Equivalence (8).

Avoiding Frame Calculations. In [22] Hobor and Villard present the rule FWRamify:

$$\frac{\forall \texttt{F}.\ \{\!|\texttt{P * F}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * F}|\!\} \qquad \texttt{R} \Longrightarrow \texttt{P * True} \qquad \texttt{Q * (P} \mathbin{-\!\circledast} \texttt{R)} \Longrightarrow \texttt{R'}}{\{\!|\texttt{R}|\!\}\ \texttt{m}\ \{\!|\texttt{R'}|\!\}}$$

At first glance this rule looks like the frame calculation could be avoided, since the conclusion talks about arbitrary preconditions R. It is a 'complification' of what we can more simply write as $(\forall \texttt{F}.\ \{\!|\texttt{P * F}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * F}|\!\}) \wedge (\texttt{R} \Longrightarrow \texttt{P * True}) \Longrightarrow \{\!|\texttt{R}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * (P} \mathbin{-\!\circledast} \texttt{R)}|\!\}$, which states that a terminating program m, specified by P * F and Q * F, will end up in a state satisfying $\texttt{Q * (P} \mathbin{-\!\circledast} \texttt{R)}$ if R contains a subheap satisfying P, which is characterised by $\texttt{R} \Longrightarrow \texttt{P * True}$.

The reason FWRamify *cannot* be used to avoid the frame calculation is the subheap-test $\texttt{R} \Longrightarrow \texttt{P * True}$, which includes a frame calculation itself, and is as hard to check as reasoning via weakening of the precondition.

In general it seems impossible to transform triples $\{\!|\texttt{P * R}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * R}|\!\}$ into strongest-postcondition form, without introducing additional proof burden, similar to FWRamify. As discussed in Sect. 4, the term $\texttt{P} \leadsto\!\texttt{*}\ \texttt{R}$ states that R holds, whenever P is removed from the heap – removal is only feasible if P exists.

Separating coimplication implies an equivalence similar to (8):

$$(\forall \texttt{R}.\ \{\!|\texttt{P} \leadsto\!\texttt{*}\ \texttt{R}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * R}|\!\}) = (\forall \texttt{X}.\ \{\!|\texttt{X}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * (P} \mathbin{-\!\circledast} \texttt{X)}|\!\}) \tag{10}$$

which we can use for forward reasoning. It is based on 'reverse modus ponens', $\texttt{X} \Longrightarrow \texttt{P} \leadsto\!\texttt{*}\ \texttt{(P} \mathbin{-\!\circledast} \texttt{X)}$, which follows directly from the Galois connection. Intuitively, the postcondition is calculated from the heap satisfying X by subtracting the part satisfying the precondition P and replacing it with a heap satisfying Q.

In practice, specifications $\{\!|\texttt{P * R}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * R}|\!\}$ can almost always be rewritten into $\{\!|\texttt{P} \leadsto\!\texttt{*}\ \texttt{R}|\!\}\ \texttt{m}\ \{\!|\texttt{Q * R}|\!\}$, especially if P is precise.

For example, the precondition of $\{\!|\texttt{p} \mapsto \texttt{_} \leadsto\!\texttt{*}\ \texttt{R}|\!\}\ \texttt{set_ptr p v}\ \{\!|\texttt{p} \mapsto \texttt{v * R}|\!\}$ assumes the hypothetical case that if we had the required resource ($\texttt{p} \mapsto \texttt{_}$), we would have a predicate R corresponding to the rest of the heap. In the postcondition, the resource does exist and is assigned to the correct value v.

Example 5. Using the specifications of the heap operations and Equivalence (10) yields the following Hoare triples (for partial correctness).

$$\begin{array}{c}
\{\!|\texttt{X}|\!\}\ \texttt{new_ptr}\ \{\!|\lambda \texttt{rv. rv} \mapsto \texttt{_ * X}|\!\} \\
\{\!|\texttt{X}|\!\}\ \texttt{delete_ptr p}\ \{\!|\texttt{p} \mapsto \texttt{_} \mathbin{-\!\circledast} \texttt{X}|\!\} \\
\{\!|\texttt{X}|\!\}\ \texttt{set_ptr p v}\ \{\!|\texttt{p} \mapsto \texttt{v * (p} \mapsto \texttt{_} \mathbin{-\!\circledast} \texttt{X)}|\!\} \\
\{\!|\texttt{X}|\!\}\ \texttt{get_ptr p}\ \{\!|\lambda \texttt{rv. p} \mapsto \texttt{rv * (p} \mapsto \texttt{rv} \mathbin{-\!\circledast} \texttt{X)}|\!\}
\end{array}$$

□

Simplifying Postconditions. Since Equivalences (8) and (10) have the same shape, we can develop a framework for forward reasoning following the lines of backward reasoning. As for backward reasoning, forward reasoning generates

```
∃np. np ↦ _ −⊛
      (∃x. p' ↦ x *
          (p' ↦ _ −⊛ np ↦ x *
           (np ↦ x −⊛
            (∃x. p ↦ x *
                (p ↦ _ −⊛ p' ↦ x *
                 (p' ↦ x −⊛
                  (∃x. np ↦ x *
                      (np ↦ _ −⊛ p ↦ x *
                       (p ↦ x −⊛ np ↦ _ * p ↦ v * p' ↦ v' * R)))))))))
⟹ p ↦ v' * p' ↦ v * R
```

Fig. 5. Forward reasoning: generated proof goal for `swap_ptr`

lengthy postconditions that need simplification. This time we have to simplify interleavings of $*$ and $-\circledast$.

Three laws are important for resolving interleavings of $*$ and $-\circledast$:

$$\frac{\mathtt{P} \Longrightarrow \mathtt{Q} \longrightarrow\!* \ \mathtt{R}}{\mathtt{P} * \mathtt{Q} \Longrightarrow \mathtt{R}} \qquad \frac{\mathtt{Q} \Longrightarrow \mathtt{P} \leadsto\!* \ \mathtt{R}}{\mathtt{P} -\!\circledast \ \mathtt{Q} \Longrightarrow \mathtt{R}} \qquad \frac{\mathtt{precise\ P}}{\mathtt{P} * \mathtt{R} \Longrightarrow \mathtt{P} \leadsto\!* \ \mathtt{R}}$$

The former two allow us to move subformulas from the antecedent to the consequent, while the latter one is a cancellation law. Depending on which term is precise different cancellation rules are needed.

We develop the following tactics for forward reasoning:

- `sep_invert` provides an 'inversion' simplification strategy, based on the aforementioned laws. It transforms interleavings of $*$ and $-\circledast$ into $\longrightarrow\!*$ and $\leadsto\!*$.
- `septract_cancel` simplifies $\leadsto\!*$ by means of cancellation rules.
- `sep_forward` integrates `septract_cancel` and `sep_cancel`, alongside a few other simple methods, to provide a simplification strategy for most formulas reasoning forwards.
- `sep_forward_solve` inverts, and then attempts to use `sep_forward` to fully solve the goal.

Figure 5 depicts the generated proof goal for `swap_ptr`. The first ten lines show the generated postcondition, whereas the last one is the given one. With the help of the developed tactics, our framework proves `swap_ptr`. As before instantiation of existential quantifiers is sometimes needed, and handled automatically for common cases.

Benchmark. One of the standard SL benchmarks is in-place list reversal:

```
list_rev p = do {
  (hd_ptr, rev) ← whileLoop (λ(hd_ptr, rev) s. hd_ptr ≠ NULL)
                    (λ(hd_ptr, rev). do {
                          next_ptr ← get_ptr hd_ptr;
                          set_ptr hd_ptr rev;
                          return (next_ptr, hd_ptr)
                        })
                   (p, NULL);
  return rev
}
```

For this example, the predicate `list` that relates pointers to abstract lists is defined in the standard, relational recursive way [35]. We used `septract_cancel` to verify the Hoare triple

```
⦃list p ps * R⦄ list_rev p ⦃λrv. list rv (rev ps) * R⦄
```

We only had to interact with our framework in a non-trivial way by adding the invariant that the list pointed to by the previous pointer is already reversed.

Case Study: System Initialisation. To investigate the robustness of our tactics in a real-world proof scenario, we again turn to the proof of system initialisation showcased earlier. We completed a portion of the proof, comprising of twenty function specifications, to demonstrate that a forward approach could achieve the same gains as our backward one, providing a degree of assurance that either approach could be taken without incurring costs.

As with weakest precondition, we are able to provide tactics enabling concise, highly automatic proofs. We give an example using the error monad, where the statement has a second postcondition `⦃P⦄ m ⦃Q⦄,⦃E⦄`. When the code throws an exception, `E` must hold. Leaving `E` free means no exception will occur. The following lemma models the result of invoking the seL4 API call `move` on two capabilities:

```
lemma invoke_cnode_move_cap:
  ⦃dest ↦c _ * src ↦c cap * R⦄
    invoke_cnode (MoveCall cap' src dest)
  ⦃dest ↦c cap' * src ↦c NullCap * R⦄,⦃E⦄
  apply (simp add:validE_def)
  apply (clarsimp simp:invoke_cnode_def liftE_bindE validE_def[symmetric])
  apply (sp sp:move_cap_sp)
  apply (sep_forward_solve)
  done
```

Most of the effort was in constructing the strongest-postcondition framework akin to `wp`. This is generic and can be used outside of our SL framework. The only work required to adapt our tactics to the proof was specialising the strongest postcondition Hoare triple transformation to the monads used, which was trivial.

7 Related Work

Separata [24,23] is an Isabelle framework for separation logic based on a labelled sequent calculus. It is a general reasoning tool for separation logic, and supports separating conjunction, separating implication, and septraction. While it can prove a number of formulas that our tactics cannot, none of these formulas appear in our verification tasks using backward and forward reasoning, and they are unlikely to show up in these styles, owing to the highly regular shape of the generated formulas of our framework. Conversely, Separata was not able to solve the weakest-precondition formulas produced in our proof body. Our framework

can integrate solvers such as Separata for the generated pre- and postconditions; hence we see these tools as additional support for our framework.

Other frameworks for reasoning in separation logic in an interactive setting such as the Coq-based VeriSmall [2] or CFML [12] stand in a similar relationship. One of the main strengths of our framework is its generality, which should allow it to be easily combined with other frameworks.

Many other tools such as Space Invader [18], Verifast [26], HOLfoot [42] offer a framework for forward reasoning within separation logic, based on variations of symbolic execution. Since they do not provide support for separating implication, they also do not perform backward reasoning in weakest-precondition style as presented. The few tools that do support separating implication are automatic full solvers [34] and do not provide a user-guided interactive framework.

The only approach using strongest postcondition we are aware of is the rule FWRamify by Hobor and Villard [22], which its authors find too difficult to use in practice. Using separating coimplication, we do not find septraction to be fundamentally more difficult than the existing well-known SL fragments.

Many existing interactive tools perform frame inference [4,41,13,1], which is the way SL was presented by O'Hearn et al. [38]. We take a different approach, and automatically divide logical reasoning from program text achieving the same separation of concerns standard Hoare logic enjoys. Our form of frame calculation is deferred to the purely logical part of the problem, where we can provide an interactive proof tactic for calculating the frame incrementally as needed.

Iris Proof Mode was developed in Coq by Krebbers et al. [31], on top of the Iris framework for Higher Order Concurrent Separation Logic [27]. They provide support for separating conjunction and implication, and use separating implication for performing weakest-precondition reasoning. As Iris is based on *affine* separation logic, where resource leaks are not reasoned about, it is unclear whether their tactics can be adopted in our *linear* setting.

Our framework operates entirely on the level of the abstract separation algebra, leaving it to the user to provide facts about model-dependent predicates, such as points-to predicates. This makes the tool highly adaptable. As we presented earlier, we have used it for ordinary heap models, for fine-grained 'partial' objects, as well as multiple different monad formalisations.

In the field of static analysis, bi-abduction is a promising technique in specification derivation, employed notably in the Infer tool [9], as well as in attempts to detect memory leaks automatically in Java programs [17]. Since the frame calculations happen in place, instead of separating logic from program as we do, it would be interesting to employ our framework to this space.

8 Summary

We have presented a methodology for backward and forward reasoning in separation logic. To support proof automation we have implemented our theoretical results in a framework for automation-assisted interactive proofs in Isabelle/HOL.

The more traditional backward reasoning works for both partial and total correctness. It makes use of the standard separating implication rule for weakest preconditions, which often counts as unwieldy. We, however, provide an interactive tactic that successfully resolves the separating implications we have encountered in sizeable practical applications.

The forward reasoning framework makes use of a new operator, the separating coimplication, which forms a nice algebraic completion of the existing operators of separating conjunction, separating implication, and septraction. The framework relies on the fact that specifications can be (re)written into the form `⦃P ⤳* R⦄ m ⦃Q * R⦄`. This is always possible when `P` is precise. While we suspect that this weaker specification will usually be true for partial correctness, we leave the general case for future work.

We have demonstrated our new proof tactics in a case study for both forward and backward reasoning. For backward reasoning, we have achieved substantial improvements, reducing the number of user proof steps by a factor of up to six. For forward reasoning, we have taken a portion of the same proof and completed it with our strongest-postcondition framework, achieving similar gains. We believe this gives empirical grounds that users can decide which style of reasoning is suitable for their problem domain, without incurring costs in mechanisation.

References

1. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) European Symposium on Programming (ESOP'11). Lecture Notes in Computer Science, vol. 6602, pp. 1–17. Springer (2011). https://doi.org/10.1007/978-3-642-19718-5_1
2. Appel, A.W.: VeriSmall: Verified smallfoot shape analysis. In: Jouannaud, J.P., Shao, Z. (eds.) Certified Programs and Proofs (CPP'11). Lecture Notes in Computer Science, vol. 7086, pp. 231–246. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_18
3. Bannister, C., Höfner, P., Klein, G.: Forward and backward reasoning in separation logic. Isabelle theories (2018), https://github.com/sel4proj/Jormungand/tree/ITP18
4. Bengtson, J., Jensen, J.B., Birkedal, L.: Charge! – A framework for higher-order separation logic in Coq. In: Beringer, L., Felty, A.P. (eds.) Interactive Theorem Proving (ITP'12). Lecture Notes in Computer Science, vol. 7406, pp. 315–331. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_21
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) Programming Languages and Systems (APLAS'05). Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005). https://doi.org/10.1007/11575467_5
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
7. Boyton, A., Andronick, J., Bannister, C., Fernandez, M., Gao, X., Greenaway, D., Klein, G., Lewis, C., Sewell, T.: Formally verified system initialisation. In: Groves, L., Sun, J. (eds.) Formal Methods and Software Engineering (ICFEM'13). Lecture

Notes in Computer Science, vol. 8144, pp. 70–85. Springer (2013). https://doi.org/10.1007/978-3-642-41202-8_6
8. Burstal, R.: Some techniques for proving correctness of programs which alter data structures. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 7, pp. 23–50. Edinburgh University Press (1972)
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
10. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Logic in Computer Science (LICS'07). pp. 366–378. IEEE (2007). https://doi.org/10.1109/LICS.2007.30
11. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: Programming Languages and Systems (APLAS'01). pp. 289–300 (2001)
12. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) International Conference on Functional Programming (ICFP'11). pp. 418–430. ACM (2011). https://doi.org/10.1145/2034773.2034828
13. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: Hall, M.W., Padua, D.A. (eds.) Programming Language Design and Implementation (PLDI'11). pp. 234–245. ACM (2011). https://doi.org/10.1145/1993498.1993526
14. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Mohamed, O.A., A. Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics (TPHOLs'08). Lecture Notes in Computer Science, vol. 5170, pp. 167–182. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_16
15. Dang, H.H., Höfner, P., Möller, B.: Algebraic separation logic. J. Logic & Algebraic Programming **80**(6), 221–247 (2011). https://doi.org/10.1016/j.jlap.2011.04.003
16. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
17. Distefano, D., Filipovic, I.: Memory leaks detection in Java by bi-abductive inference. In: Rosenblum, D.S., Taentzer, G. (eds.) Fundamental Approaches to Software Engineering (FASE'10). Lecture Notes in Computer Science, vol. 6013, pp. 278–292. Springer (2010). https://doi.org/10.1007/978-3-642-12029-9_20
18. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06). Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer (2006). https://doi.org/10.1007/11691372_19
19. Floyd, R.W.: Assigning meanings to programs. Mathematical Aspects of Computer Science **19**, 19–32 (1967)
20. Gordon, M., Collavizza, H.: Forward with Hoare. In: Roscoe, A., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare. pp. 101–121. Springer (2010). https://doi.org/10.1007/978-1-84882-912-1_5
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
22. Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: Giacobazzi, R., Cousot, R. (eds.) Principles of Programming Languages (POPL'13). pp. 523–536. ACM (2013). https://doi.org/10.1145/2429069.2429131
23. Hóu, Z., Goré, R., Tiu, A.: Automated theorem proving for assertions in separation logic with all connectives. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction (CADE'15). Lecture Notes in Computer Science, vol. 9195, pp. 501–516. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_34

24. Hóu, Z., Sanan, D., Tiu, A., Liu, Y.: Proof tactics for assertions in separation logic. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving (ITP'17). Lecture Notes in Computer Science, vol. 10499, pp. 285–303. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_19
25. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Principles of Programming Languages (POPL'01). vol. 36, pp. 14–26. ACM (2001). https://doi.org/10.1145/373243.375719
26. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) Programming Languages and Systems (APLAS'10). Lecture Notes in Computer Science, vol. 6461, pp. 304–311. Springer (2010). https://doi.org/10.1007/978-3-642-17164-2_21
27. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Principles of Programming Languages (POPL'15). pp. 637–650. ACM (2015). https://doi.org/10.1145/2676726.2676980
28. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. Transaction of Computer Systems **32**(1), 2:1–2:70 (2014). https://doi.org/10.1145/2560537
29. Klein, G., Kolanski, R., Boyton, A.: Mechanised separation algebra. In: Beringer, L., Felty, A.P. (eds.) Interactive Theorem Proving (ITP'12). Lecture Notes in Computer Science, vol. 7406, pp. 332–337. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_22
30. Klein, G., Kolanski, R., Boyton, A.: Separation algebra. Archive of Formal Proofs (2012), `http://isa-afp.org/entries/Separation_Algebra.shtml`, Formal proof development
31. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Castagna, G., Gordon, A.D. (eds.) Principles of Programming Languages (POPL'17). pp. 205–217. ACM (2017). https://doi.org/10.1145/3009837.3009855
32. Lee, W., Park, S.: A proof system for separation logic with magic wand. In: Jagannathan, S., Sewell, P. (eds.) Principles of Programming Languages (POPL'14). pp. 477–490. ACM (2014). https://doi.org/10.1145/2535838.2535871
33. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
34. Maclean, E., Ireland, A., Grov, G.: Proof automation for functional correctness in separation logic. Journal of Logic and Computation **26**(2), 641–675 (2016). https://doi.org/10.1093/logcom/exu032
35. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. In: Baader, F. (ed.) Automated Deduction (CADE'03). Lecture Notes in Computer Science, vol. 2741, pp. 121–135. Springer (2003). https://doi.org/10.1007/978-3-540-45085-6_10
36. Nipkow, T.: Hoare logics in Isabelle/HOL. In: Schwichtenberg, H., Steinbrüggen, R. (eds.) Proof and System-Reliability. pp. 341–367. Springer (2002). https://doi.org/10.1007/978-94-010-0413-8_11
37. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9

38. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic (CSL'01). Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001). https://doi.org/10.1007/3-540-44802-0_1
39. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS'02). pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
40. Reynolds, J.C.: An introduction to separation logic. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 22, pp. 285–310. IOS Press (2009). https://doi.org/10.3233/978-1-58603-976-9-285
41. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Grove, D., Blackburn, S. (eds.) Programming Language Design and Implementation (PLDI'15). pp. 77–87. ACM (2015). https://doi.org/10.1145/2737924.2737964
42. Tuerk, T.: A Separation Logic Framework for HOL. Ph.D. thesis, University of Cambridge, UK (2011)
43. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) Concurrency Theory (CONCUR'07). Lecture Notes in Computer Science, vol. 4703, pp. 256–271. Springer (2007). https://doi.org/10.1007/978-3-540-74407-8_18