A Certified Verifier for a Fragment of Separation Logic

Nicolas Marti, Reynald Affeldt

Separation logic is an extension of Hoare logic to verify imperative programs with pointers and mutable data-structures. Although there exist several implementations of verifiers for separation logic, none of them has actually been itself verified. In this paper, we present a verifier for a fragment of separation logic that is verified inside the Coq proof assistant. This verifier is implemented as a Coq tactic by reflection to verify separation logic triples. Thanks to the extraction facility to OCaml, we can also derive a certified, stand-alone and efficient verifier for separation logic.

1 Introduction

Separation logic is an extension of Hoare logic to verify imperative programs with pointers and mutable data-structures [6]. There exist several implementations of verifiers for separation logic [11][12][14], but they all share a common weak point: they are not themselves verified.

It makes little doubt that a verifier for separation logic can be verified using, say, a proof assistant. The real question is: At which price? Indeed, such verifiers are non-trivial pieces of software. They require manipulation of concepts such as fresh variables, that are notoriously hard to get right in a proof assistant. They also rely on decision procedures for arithmetic that are not necessarily available in a suitable form. This means at least a non-negligible implementation work.

In this paper, our contribution is to develop and verify in the Coq proof assistant [1] a new verifier for a fragment of separation logic. This verifier can be used inside Coq as a tactic to prove separation logic triples. Thanks to the extraction facility of Coq to OCaml, it can also be used as a certified, stand-alone and efficient verifier. Though our verifier is not as versatile as recent verifiers, we believe that our work provides a good evaluation of the effort required by formal verification of verifiers for separation logic.

The goal of our verifier is to prove automatically separation logic triples \{P\}_c\{Q\}, where \(c\) is a command, and \(P\) and \(Q\) are assertions of separation logic. For the assertions, we cannot use the full separation logic language because the validity is undecidable. Instead, we deal with a fragment identified in previous work by other authors [7][11] as a good candidate for automation. We extend this language with Presburger arithmetic so as to be able to handle pointer arithmetic. The only datatypes we provide are singly-linked lists, but the ideas extend to other recursive datatypes such as trees. A formal description of separation logic follows in Sect. 3; our target assertion language is formally explained in Sect. 4.

The basic design idea of our verifier is to turn separation logic triples into logical implications between assertions to be proved automatically. Similarly to related work [11][14], this is implemented in three successive phases:

1. Verification conditions generator: The input
triple is cut into a list of loop-free triples.
2. **Triple transformation**: Every loop-free triple is turned into logical implications between assertions.
3. **Entailment**: Every implication derived from the previous phase is proved valid.

Besides formal verification of these three phases, another originality of our work is the triple transformation phase in itself: we appeal to a new proof system that mixes backward and forward reasoning whereas related work \[11,14\] essentially relies on forward reasoning (the advantages of our approach are discussed in detail in Sect. 9.1). In the rest of this paper, we explain for each phase of our verifier what it does and how we prove it correct: the entailment phase in Sect. 5, the triple transformation phase in Sect. 6, and the verification conditions generator in Sect. 7. The resulting verifier amounts to a simple combination of these three phases, as summarized in Sect. 8. In Sect. 9, we comment on practical aspects: the size of generated proof-terms and performance benchmarks for the derived stand-alone OCaml verifier. Sect. 10 is dedicated to comparison with related work. We conclude in Sect. 11.

### 2 About the Coq Proof Assistant

The Coq proof assistant [1] is a tool for formal verification of software. It provides a typed functional language to write programs and specifications, and tactics to build proofs.

In order for the user to build proofs by induction, Coq automatically proves induction principles for data structures and predicates defined inductively. For example, given the following definition of naturals:

**Inductive nat : Set := O : nat | S : nat → nat.**

Coq automatically proves the well-known induction principle:

\[
\text{nat} \text{\_ind : } \forall (P : \text{nat} \rightarrow \text{Prop}), \quad P \text{\_}0 \rightarrow (\forall n, P n \rightarrow P (S n)) \rightarrow \forall n, P n
\]

(The standard Coq types Set and Prop establish a distinction between data structures and predicates.) Like data structures, predicates also can be defined by induction. For example, the relation “less than” is defined the predicate \(\text{le} \) (noted \(\leq\)):

**Inductive le (n : nat) : nat → Prop :=
\[
\text{le\_n : } n \leq n \mid \text{le\_S} : \forall m, n \leq m \rightarrow n \leq S m.
\]

The constructors of \(\text{le}\) can be seen as axioms whose application yields the proof of lemmas. For example, the proof-term \(\text{le\_S} \text{\_O} \text{\_O} (\text{le\_n O})\) of type \(0 \leq S 0\) is a formal proof of the lemma “0 is less than 1”.

There are basically two ways to prove lemmas in Coq: successive applications of lemmas, or development and verification of a decision procedure as a Coq function. The latter method is known as **reflection**, pioneered in Coq by Boutin [3]. Its main benefit is that generated proof-terms are small, because it amounts to computing the return value of a Coq function [1]. Let us illustrate the difference with an example. One can prove inequalities over naturals with Coq native tactics:

**Lemma foo : 18 \leq 30.**

\[
\text{repeat (apply le\_n || apply le\_S).}
\]

**Qed.**

The keywords **Lemma** and **Qed** start and end a formal proof. The tactic **apply** applies an existing lemma. Tactics can be combined: **repeat** repeats a tactic, “||” executes the tactic on the right or the one on the left in case of failure. This usage of tactics has the defect to generate large proof-terms. To solve such inequalities by reflection, one would first write a Coq function deciding inequalities and prove it correct:

**Fixpoint leq_nat (x y:nat) {struct x} : bool :=
\[
\text{match x with}
\]

\[
\mid O => true
\]

\[
\mid S x' => match y with
\]

\[
\mid O => false | S y' => leq_nat x' y' end
\]

**end.**

**Lemma leq_nat\_correct : \forall x y, leq_nat x y = true \rightarrow x \leq y.**

\[
\text{...}
\]

(The keyword **Fixpoint** defines recursive functions.)

Using the function **leq_nat** and its correctness lemma **leq_nat\_correct**, our lemma can then be proved as follows:

**Lemma foo : 18 \leq 30.**

\[
\text{apply leq\_nat\_correct; auto.}
\]

**Qed.**

Decision procedures implemented by reflection lead to short proof-terms. The downside is a more intricate implementation. This is nonetheless a tactic by reflection that we propose to implement in this paper.

†1 Another advantage of reflection is that it allows for formal proof in Coq of the completeness of decision procedures but we are not concerned with this aspect in this paper.
3 Separation Logic in Coq

The Coq tactic we build in this paper (and from which we will derive our certified verifier) is tailored to verification of separation logic triples as defined in our previous work [13]. In this section, we recall the aspects of our encoding that are necessary to understand the correctness statements in this paper. All proof scripts we refer to can be found online [15].

3.1 The Programming Language

Separation logic is an extension of Hoare logic with a native notion of heap and pointers. In separation logic, the state of a program is a couple of a store (a map from variables to values) and a heap (a map from locations to values). There are two commands to access the heap: lookup (or dereference) and mutation (or destructive update).

The syntax of the programming language in Coq is defined as follows (file axiomatic.v in [15]):

Inductive cmd : Set :=
| assign : var.v → expr → cmd
| lookup : var.v → expr → cmd
| mutation : expr → expr → cmd
| seq : cmd → cmd → cmd
| ifte : expr_b → cmd → cmd → cmd

... Notation "x <- e" := (assign x e).
... Notation "x '<-*' e" := (lookup x e).
Notation "e '*<-' f" := (mutation e f).
Notation "c ; d" := (seq c d).
... The type expr (resp. expr_b) is the type of numerical (resp. boolean) expressions:

Inductive expr : Set :=
| var_e : var.v → expr
| int_e : val → expr
| add_e : expr → expr → expr
| min_e : expr → expr → expr
... Inductive expr_b : Set :=
| true_b : expr_b
| eq_b : expr → expr → expr_b
| neg_b : expr_b → expr_b
| and_b : expr_b → expr_b → expr_b
...

Expressions are evaluated for a given store by the functions eval and eval_b:

Fixpoint eval (e:expr) (s:store.s) : Z := ...
Fixpoint eval_b (e:expr_b) (s:store.s) : bool := ...

Let us explain the operational semantics informally. The assignment x<-e updates the value of the variable x with the result of the evaluation of the expression e in the current state (eval e s, with s the store of the current state). The lookup x<-*e updates the value of the variable x with the value contained inside the cell of location eval e s. The heap mutation e1<-*e2 modifies the cell of location eval e1 s with the value eval e2 s (heap accesses fail if the accessed cell is not in the heap). This operational semantics is formalized as a ternary predicate exec noted s--c-->s’ where s is a starting state, c a program, and s’ the resulting state (states are encoded with the option type whose None constructor represents error states). See [13] for detailed explanations.

3.2 The Assertion Language

The assertion language is defined using the standard Coq predicates. More precisely, an assertion is defined as a function from states to Prop:

Definition assert := store.s → heap.h → Prop.
(The keyword Definition defines macros, and more generally non-recursive functions.)

This technique of encoding is known as shallow encoding. It is a convenient way to encode logical connectives and reason using them; see [2] for a more in-depth discussion about the advantages of shallow encoding and the dichotomy with deep encoding. For example, the classical implication of separation logic can be directly encoded using the classical implication of Coq:

Definition entails (P Q : assert) : Prop :=
∀ s h, P s h → Q s h.
Notation "P => Q" := (entails P Q).

There are four constructs specific to separation logic. The atoms empty (Coq notation: emp) and mapsto (notation: |->), and the connectives separating conjunction (notation: **) and separating implication (notation: -*).

emp holds when the heap is empty:

Definition emp : assert := fun s h ⇒ h=heap.emp.
e1|->e2 holds when the heap is a single cell containing the value eval e2 s and whose location is eval e1 s (val2loc is a cast):

Definition mapsto e1 e2 : assert :=
fun s h ⇒ ∃ p, val2loc (eval e1 s) = p ∧ h = heap.singleton p (eval e2 s).
Notation "e1 '<->' e2" := (mapsto e1 e2).

P**Q holds when we can split the heap into two disjoint heaps (disjointness is noted # and union is noted ++) such that P holds for one of them, and Q holds for the other.
Definition con \((P \mathbin{\ast} Q) : assert\) :=
\[
\text{fun } s \ h \Rightarrow \exists h_1, \exists h_2,
\ h_1 \# h_2 \land h = h_1 \ast h_2 \land P \ h_1 \land Q \ h_2.
\]
Notation "\(P \mathbin{\ast\ast} Q" := (\text{con } P \ Q).

\(\text{P} \rightarrow \text{Q}\) holds when \(Q\) holds on the current heap extended with any (disjoint) heap for which \(P\) holds:
Definition imp \((P : assert) \rightarrow (Q : assert) : assert :=
\[
\text{fun } s \ h \Rightarrow
\forall h', h \# h' \land P \ s \ h' \rightarrow Q \ s \ (h \ast h').
\]
Notation "\(P \rightarrow\rightarrow Q" := (\text{imp } P \ Q).

The separating implication is essential for reasoning because it captures logically the notion of destructive update. We will use it as such to give the semantics of an intermediate language in Sect. 6.1. However, we do not use it in specifications in this paper because it talks about not-yet-allocated memory cells and our target fragment of separation logic does not feature natively dynamic allocation.

### 3.3 Separation Logic Triples

The semantics for partial correctness of triples \(\{P\} c \{Q\}\) defined as follows: considering the program \(c\), for every initial state for which the precondition \(P\) holds, (1) the execution will not raise an error, and (2) the postcondition \(Q\) holds for every final state. Put formally in Coq:

Definition semax' \((c:cmd) (P \mathbin{\mathit{assert}}) : Prop :=
\[
\forall s \ h,
(P \ s \ h \rightarrow \neg \text{Some } (s, h) \rightarrow c \rightarrow \text{None}) \land
(\forall s' h', P \ s \ h \rightarrow
\text{Some } (s, h) \rightarrow c \rightarrow \text{Some } (s', h') \rightarrow Q \ s' \ h').
\]

The axiomatic semantics is defined as an inductive predicate whose constructors rephrase Reynolds axioms [6]:

Inductive semax : assert \rightarrow cmd \rightarrow assert \rightarrow Prop :=
\[
\ldots
\]
Notation "\{\{ P \} \} c \{\{ Q \}\}" := (semax \ P \ c \ Q).

We formally proved that \(\text{semax}\) is sound and complete w.r.t. \(\text{semax}'\), in other words, using Reynolds’ axioms, we can prove any valid separation logic triple.

### 4 Target Fragment of Separation Logic

In this section, we present the fragment of the assertion language of separation logic that our verifier deals with. This is basically the same fragment as [7], where it was chosen as a good candidate for automation because entailment (classical implication of assertions) is decidable. We extend this fragment with Presburger arithmetic to handle pointer arithmetic. Since programs never multiply pointers between each other, we think that this extension suffices to enable most verifications; the same extension is done in [14]. The only datatype we deal with is singly-linked lists. We think that the ideas we develop in this paper for lists extend to other recursive datatypes such as trees, along the same lines as [11].

#### 4.1 Syntax and Informal Semantics

Formulas of our fragment represent states symbolically. To represent a store symbolically, we use the language of boolean expressions \(\text{expr}_b\) introduced in Sect. 3.1. This gives us enough expressiveness to write pointer arithmetic formulas. To represent a heap symbolically, we use the following fragment \(\Sigma\) of the assertion language of separation logic:

\[
\begin{align*}
\text{Inductive } \Sigma & : \text{Set} := \\
& | \text{emp } : \Sigma \\
& | \text{singl } : \text{expr } \rightarrow \text{expr } \rightarrow \Sigma \\
& | \text{cell } : \text{expr } \rightarrow \Sigma \\
& | \text{star } : \Sigma \rightarrow \Sigma \rightarrow \Sigma \\
& | \text{lst } : \text{expr } \rightarrow \text{expr } \rightarrow \Sigma.
\end{align*}
\]

Simply put, this syntax represents the connectives defined in Sect. 3.2: \(\text{emp}\) represents the empty heap like the homonym connective defined by shallow encoding; \(\text{singl}\) is syntax for \text{mapsto}; \(\text{cell } e\) represents a singleton heap whose contents is unknown; \(\text{star } h h'\) is the syntactic separating conjunction (Coq notation: \(h \ast h'\); this is the same notation as the “semantic” separating conjunction in Sect. 3.2; in informal arguments, we will write \(\ast\) for the separating conjunction). Note that \(\Sigma\) does not contain the separating implication of separation logic. Compared to the shallow encoding of Sect. 3.2, we add the formula \(\text{lst } e e'\) that represents a heap that contains a singly-linked list whose head has location \(e\) and whose last element points to \(e'\), as illustrated below:

![Singly-Linked List Diagram](image)

To summarize, the syntax of our assertion language \(\text{assert}\) is defined as a product of \(\text{expr}_b\) and \(\Sigma\):

Definition \(\Pi := \text{expr}_b \times \Sigma\).

Definition \(\text{assert} := (\Pi \times \Sigma)\).

In informal arguments, we will write \((\pi, \sigma)\) for assertions.
4.2 Formal Semantics

In the previous section, we have defined the syntax of formulas in Coq. Their semantics has already been defined in Sect. 3.2 by a shallow encoding. In this section, we make the relation between both with a satisfiability relation. This technique of encoding is called deep encoding and is typical of tactics by reflection. Indeed, the latter needs to “parse” the assertion language to prove the validity of formulas, which is difficult to do when the syntax is not an inductive type.

The formal semantics of Sigma formulas is a satisfiability relation between (syntactic) formulas and states. It is defined by a function Sigma_interp of type store.s -> heap.h -> Prop where store.s -> heap.h -> Prop is precisely the type assert of formulas in our shallow encoding:

\[
\text{Fixpoint Sigma_interp } (a : \Sigma) : \text{assert} := \\
\text{match a with} \\
\text{emp -> sep.emp} \\
\text{singl e1 e2 -> fun s h =>} \\
(e1 \mid \rightarrow e2) \text{ s h } \triangleright \text{eval e1 s } \neq 0 \\
\text{cell e } \rightarrow \text{fun s h =>} \\
(\exists v, (e \mid \rightarrow \text{int}_e v) \text{ s h }) \land \text{eval e s } \neq 0 \\
\text{if s } \star \times \text{ s2 =>} \\
\Sigma_\text{interp s1 } \star \times \Sigma_\text{interp s2} \\
\text{lst e1 e2 } \rightarrow \text{Lst e1 e2} \\
\text{end.}
\]

(the formulas from Sect. 3.2 are encapsulated in a module sep to avoid naming conflicts) where Lst is an inductive type of the appropriate type defining singly-linked lists:

\[
\text{Inductive Lst : expr } \rightarrow \text{expr } \rightarrow \text{expr } \rightarrow \text{assert} := \\
\text{Lst_end: } \forall \epsilon e' \epsilon_1 s h, \\
\text{eval e s } = \text{eval e' s } \rightarrow \text{sep.emp s h } \rightarrow \\
\text{Lst e e' s h} \\
\text{Lst_next: } \forall \epsilon e' e'' \epsilon_1 \epsilon_2 \text{ data s h h1 h2,} \\
\epsilon_1 \# \epsilon_2 \rightarrow \epsilon_1 \# h1 \rightarrow h2 \rightarrow \\
\text{eval e s } \neq \text{eval e' s } \rightarrow \\
\text{eval e s } \neq 0 \\
\text{eval (e } \times e \text{ nat_e 1) s } \neq 0 \\
\epsilon_1 \mid \rightarrow e'' \star (e \times e \text{ nat_e 1 } \mid \rightarrow \text{data}) \text{ s h1 } \rightarrow \\
\text{Lst e' e'' s h} \\
\text{Lst e e' s h.}
\]

The semantics of our fragment is finally defined as the conjunction of the satisfiability relations of its two components (expr.pi is syntactically equal to expr_b):

\[
\text{Definition assert_interp } (a : \text{assrt}) : \text{assert} := \\
\text{match a with} \\
\text{pi, sigm } \Rightarrow \text{fun s h =>} \\
\text{eval_pi pi s } = \text{true } \land \text{Sigma_interp sigm s h} \\
\text{end.}
\]

4.3 Disjunctions of Assertions

In fact, we further need to extend our assertion language to represent disjunctions of assertions. Intuitively, this is because loop invariants are usually written as disjunctions. In informal arguments, we will write \(\langle \pi_1, \sigma_1 \rangle \lor \ldots \lor \langle \pi_n, \sigma_n \rangle\) for disjunctions of assertions. Adding this disjunction on top of the fragment allows to handle disjunction for the separation logic part, without multiplying the set of rules necessary to prove disjunction and without loss of expressivity. Indeed, all formulas belonging to a fragment with entailment and without loss of expressivity.

Here follows an example of such a goal (see Fig. 2 for an informal account of the proof built underneath):

\[
\text{Goal entail} \\
(\text{true}_b, \text{list e' } \star \times e' \mid \rightarrow e'' \star \times \text{cell (e'+1) } \star \times \text{list e'' } \lor 0)
\]

5 Entailment

In this section, we present a proof system for entailments of assertions defined in the previous section. Using this proof system, we implement a Coq tactic and a function to prove validity for entailment between two formulas of type assert (files frag_list_entail.v and expr_b_dp.v in [15]).

5.1 Entailment Proof System

Our proof system enables derivation of entailments of type assert -> assert -> Prop such that the left hand side (lhs) semantically implies the right hand side (rhs). In Coq, this proof system takes the form of an inductive predicate entail. An excerpt in informal notation is displayed in Fig. 1. Most rules are fairly intuitive. For example, we can take a look at the rule coml, that captures the fact that the separating conjunction is commutative on the left of implication.

We have implemented a tactic (Entail, not extractable) that iteratively applies the rules of entail to solve entailments. Here follows an example of such a goal (see Fig. 2 for an informal account of the proof built underneath):

\[
\text{Goal entail} \\
(\text{true}_b, \text{list e' } \star \times e' \mid \rightarrow e'' \star \times \text{cell (e'+1) } \star \times \text{list e'' } \lor 0)
\]
5.2 Entailment Verification Procedure

In this section, we explain the Coq function entail_fun that proves entailments. Because we verify it, this function can be used as a tactic by reflection. It implements a reasoning similar to the entail proof system but this is no redundant work: we will actually use the Entail tactic to prove the correctness of entail_fun.

5.2.1 Implication Between Heaps

The first building block of the entail_fun function is a function Sigma_impl that proves the validity of implications between two abstract heaps. This function iteratively calls the function entail_soundness (§7), which tries to eliminate, subheap by subheap, the lhs sig1 ** remainder from the rhs sig2. This elimination is performed by the function elim_common_cell (Fig. 3), which tries to remove the subheap sig from both sig ** remainder and sig'. It is essentially a case-analysis on both heaps leading to the application of an entail rule. For example, Fig. 3 shows the case for which the rule lstelim''' of the entail proof system applies.

In fact, Fig. 2 also provides an illustration of what is achieved by the function Sigma_impl. The intermediate abstract heaps happen to be the successive results of elimination of common subheaps by elim_common_cell. For example, here is the result of the third call:

\( \text{elim_common_cell true_b (cell e'+1)} \)
\( \text{(lst e'' 0) (cell e'+1 ** lst e''0) = Some (lst e'' 0, lst e'' 0)} \)

5.2.2 Entailments Between Assertions

Above, we explained a function Sigma_impl to prove the validity of the implication between two abstract heaps. Here, we explain how to use this function to verify entailments of assertions.

There are two ways of proving entailments between assertions (type assert). The first way is to prove that the lhs is contradictory (i.e., it implies False); this corresponds to the application of the rule incons of the entail proof system. The second way is to prove the implication between the abstract heaps on both hand sides (using Sigma_impl) and to prove the implication between the abstract stores; this corresponds to the application of the
The entail proof system. In order to prove the implication between abstract stores, we need a function to decide Presburger arithmetic; for this purpose, we have certified in Coq a decision procedure based on Fourier-Motzkin variable elimination (this is actually the function \textit{expr\_bdp} that already appears in Fig. 3).

This reasoning is implemented by the function \textit{assert\_entail\_fun} that extends beforehand the lhs of the entailment with arithmetic constraints, as described at the end of Sect. 5.1.

5.2.3 Entailments Between Disjunctions

Above, we explained a function \textit{assert\_entail\_fun} to verify entailments of assertions (type \textit{assert}). Here, we explain how to use this function to verify entailments of disjunctions of assertions (type \textit{Assrt}).

Elimination of Disjunctions in the Lhs

To eliminate disjunctions in the lhs of the entailment we use the rule \textit{elim\_lhs\_disj} (Fig. 4, function \textit{Assrt\_entail\_Assrt\_fun} in file \textit{frag\_list\_entail.v}). Thanks to this rule, we can decompose an entailment between \textit{assert} formulas into a list of entailments between an \textit{assert} formula (on the lhs) and an \textit{Assrt} formula (on the rhs).

Elimination of Disjunctions in the Rhs

The elimination of disjunctions in the rhs of the entailment is more subtle. It is possible to use the rule \textit{elim\_rhs\_disj1} (Fig. 4, function \textit{orassrt\_impl\_Assrt1} in file \textit{frag\_list\_entail.v}). But this rule is not sufficient, as illustrated by the
following counter-example:

\[ \langle \text{true.b, } \sigma \rangle \vdash (y = 0, \sigma) \lor (y \neq 0, \sigma) \]

Such rhs are however important because they are typical of loop invariants. Indeed, a loop invariant usually consists of a disjunction of all possible outcomes of the loop condition, and each disjunct can only be proved under some hypothesis about this outcome. To handle these situations, we use the rule elim_rhs_dizj2 (Fig. 4, functions orpi and orassrt_impl_Assrt2 in file frag_list_entail.v).

We are now equipped to explain the function entail_fun, that proves the validity of entailments. It takes as input an assert and an Assrt, uses the rules from Fig. 4 to eliminate the disjunctions in the rhs, and finally calls assrt_entail_fun:

Definition entail_fun
(a:assert) (A:Assrt) (l:list (assert * assert))
: result (list (assert * assert)) := ...

It returns an option type (constructor Good if everything is proved). The proof of correctness of entail_fun boils down to the following lemma:

Lemma entail_fun_correct: ∀ a a 1, entail_fun a A l = Good → assrt_interp a ==> Assrt_interp A.

We do not think that the entail_fun function is a complete decision procedure because of the rules for entailments between disjunctions. However, it is already useful in practice, as illustrated by the various non-trivial examples in Sect. 9.

6 Triple Transformation

In the previous section, we saw how to solve entailments of assertions of separation logic. In this section, we explain how to transform a loop-free triple into such an entailment (file frag_list_triple.v in [15]).

6.1 Language for Weakest-preconditions

Before explaining the triple transformation, we need to introduce the type wpAssrt. This type represents the weakest precondition of a program with respect to its postcondition:

Inductive wpAssrt : Set :=
| wpElt: Assrt → wpAssrt
| wpSubst: list (var.v * expr) → wpAssrt → wpAssrt
| wpLookup: var.v → expr → wpAssrt → wpAssrt
| wpMutation: expr → expr → wpAssrt → wpAssrt
| wpIf: Pi → wpAssrt → wpAssrt → wpAssrt.

The constructor wpElt represents a postcondition with no program. The wpSubst constructor represents the weakest precondition of a sequence of assignments whose postcondition is itself some weakest precondition, etc.

The interpretation of this language is computed by a weakest precondition generator using backward separation logic axioms from [6]:

Fixpoint wpAssrt_interp (a: wpAssrt) : assert :=
match a with
| wpElt al => Assrt_interp al
| wpSubst l L => subst_lst2update_store l (wpAssrt_interp L)
| wpLookup x e L => (fun s h => ∃ e0, (e |-> e0 ** (e |-> e0 -* wpAssrt_interp L)) s h)
| wpMutation e1 e2 L => (fun s h => ∃ e0, (e1 |-> e0 ** (e1 |-> e2 -* wpAssrt_interp L)) s h)
| wpIf b L1 L2 => (fun s h => (eval_b b s = true → wpAssrt_interp L1 s h) ∧ (eval_b b s = false → wpAssrt_interp L2 s h))
end.

6.2 Triple Transformation Proof System

Now that we have explained wpAssrt, we can explain the role of the tritra proof system. It has type assert → wpAssrt → Prop. Intuitively, the two parameters form a triple of separation logic: the first parameter is a assertion of separation logic (a precondition) and the second parameter is a weakest precondition, or equivalently a program with a postcondition. The constructors of the tritra proof system represent elementary triple transformations. An excerpt in informal notation is displayed in Fig. 5.

The two rules lookup and mutation are intuitive because the lookup (resp. mutation) is the leading command of the program. When lookups and mutations are preceded by assignments, the transformation rules must take care of captures of variables, as exemplified by the rule subst_lookup. Despite these technical difficulties (in particular, the usage of fresh variables), we managed to prove the soundness of this proof system inside Coq:

Lemma tritra_soundness : ∀ P Q, tritra P Q → assrt_interp P ==> wpAssrt_interp Q.

6.3 Triple Transformation Procedure

Equipped with the tritra proof system, we can transform any valid triple \( \{ P \} c (Q) \) into a couple \((P, Q')\) where \( Q' \) is a wpAssrt of the form wpElt.
The implication $P \rightarrow Q'$ (or equivalently the entailment $P \vdash Q'$) can then be solved by entail_fun. This operation is implemented by the function tritra_step of type $P \rightarrow S \rightarrow wpAssrt \rightarrow option ((P \times S) \times wpAssrt))$ that tries to apply tritra rules (at the price of some rewriting of the precondition) so as to return a list of subgoals.

The function that implements the whole triple transformation phase is triple_transformation: it recursively calls tritra_step and then entail_fun on resulting subgoals:

\begin{verbatim}
Fixpoint triple_transformation (P : Assert) (Q : wpAssrt) { struct P } :
  option (list ((P \times S) \times wpAssrt)) := ...

Lemma triple_transformation_correct: \forall P Q, tritra_transformation P Q = Some nil \rightarrow
  Assert_inter P \equiv wpAssrt_inter Q.
\end{verbatim}

The triple transformation is complete as long as the intermediate arithmetic goals it generates fall into Presburger arithmetic, which is likely in practice because pointers are never multiplied between each other. The fact that the triple transformation is complete simply comes from the fact the rules of the tritra proof system cover all possible programs.

### 7 Verification Conditions Generator

In the previous section, we explained how to prove loop-free separation logic triples. In this section, we explain how to turn a separation logic triple whose loops are annotated with invariants into a list of loop-free triples (file frag_list_vcg.v in [15]).

The generation of loop-free triples from a separation logic triple is the role of the verification conditions generator. The main idea of this operation can be explained as follows. Suppose we are given a triple $\{P\}c_1;\text{while } b\text{ do } c_2\{Q\}$ where $I$ is an invariant. To prove this triple, it is sufficient to prove the three triples $\{P\}c_1\{I\}$, $\{I \land b\}c_1\{I\}$, and $\{I \land \neg b\}c_2\{Q\}$. Applying this idea repeatedly turns a separation logic triple into a set of loop-free triples, as implemented by the following function:

\begin{verbatim}
Fixpoint vcg (c : cmd') (Q : wpAssrt) { struct c } :
  option (wpAssrt \times (list (Assert \times wpAssrt))) := ...
\end{verbatim}

In addition to a list of subgoals, vcg returns the weakest precondition of the program (this is the first projection of the return value in the type above).

The verification of vcg amounts to check that, under the hypothesis that subgoals can be verified, the returned condition is indeed a weakest precondition. Recall from Sect. 3.3 that separation logic triples are noted $\{\cdot\};\{\cdot\}; Assert_inter$ and $wpAssrt_inter$ were defined respectively in Sec-
tions 4.3 and 6.1:

Lemma vcg_correct : \forall c Q Q' l,
v cg c Q = Some (Q', l) \rightarrow
(\forall A L, In (A, L) l \rightarrow
Assrt_interp A \Rightarrow wpAssrt_interp L) \rightarrow
{{ wpAssrt_interp Q' }}
proj_cmd c
{{ wpAssrt_interp Q }}.

The verification condition generator is complete, as it consists in applying the Reynolds axioms for sequence and loop, which have been proved complete formally inside of Coq (see Sect. 3.3).

8 Put It All Together

The resulting verification procedure is a Coq function that takes as input a command c (annotated with loop invariants), a precondition P, and a postcondition Q. First, it calls vcg to compute a set of sufficient subgoals. Then, it calls triple_transformation for all these subgoals. If all of them can be proved, it returns Some nil. Otherwise, it returns the list of unsolved subgoals for information:

Definition bigtoe_fun (c: cmd') (P Q: Assrt)
: option (list ((Pi * Sigma) * wpAssrt)) :=
macht vcg c (wpElt Q) with
| None => None
| Some (Q', l) =>
macht triple_transformation P Q' with
| Some l' =>
macht triple_transformations l with
| Some l'' => Some (l' ++ l'')
| None => None
end
| None => None
end
end.

The correctness of this tactic amounts to prove that, if it returns Some nil, then the corresponding separation logic triple holds:

Lemma bigtoe_fun_correct: \forall P Q c,
bigtoe_fun c P Q = Some nil =>
{{ Assrt_interp P }}
proj_cmd c
{{ Assrt_interp Q }}.

Now, in our formal proofs of Hoare triples, we can apply this lemma to delegate the proof to the computation of the function bigtoe_fun.

9 Experimental Measurements

In this section, we present a comparison between our approach and backward/forward reasoning, as well as a benchmark for our verifier.

9.1 Comparison with Backward and Forward Reasoning

All previous work on automatic verification of separation logic triples use forward reasoning [11] [12] [14]. The main reason is that backward reasoning (using a standard weakest precondition generator for separation logic) produces postconditions with separating implications for which there exists no automatic prover (as pointed out in [11]). Although decidability results exist [5] [10] [9], the separating implication is actually seldom used in specifications of algorithms (one notable exception is [4]). However, forward reasoning has the disadvantage of adding, for each variable modification, a conjunctive clause with possibly a fresh variable. This is not desirable in practice because decision procedures for Presburger arithmetic have an exponential complexity w.r.t. the number of clauses and variables. Our approach based on the proof system tritra can be shown experimentally to produce less clauses.

In Fig. 6, we illustrate transformation steps for a program swapping the values of two cells, using our approach. The transformations produced by forward and backward reasoning are displayed in Figures 7 and 8. We can observe that tritra does not add new connectives or variables, contrary to both backward and forward reasoning. (For the latter, no fresh variables have been introduced, because the variables modified by the program do not appear in the precondition.)

In order to measure more precisely differences between our approach and forward reasoning, we have implemented, inside of Coq, a proof system similar to [11] extended with pointer arithmetic (file LSF.v in [15]). We proved interactively several separation logic triples, and compared the size of the compiled proofs terms produced by both approaches. This comparison was done on three different programs. swap is the separation logic triple whose transformation is illustrated in Fig. 6. The init(n) program is a loop that initializes a given field for n contiguous occurrences of a data-structure. This program makes use of pointer arithmetic, as the loop iteratively increments the value of the pointer to the current data-structure, while the data-structures locations are specified by a multiple of the data-
structure’s size in the pre/postconditions. Finally, max3 is a program that returns the maximum value of three variables. The results are presented in Table 1, where the percentages correspond to the overhead of forward reasoning. We can conclude that our approach produces smaller proof-terms, because the underlying arithmetic decision procedure (here, the Coq omega) applies less lemmas to prove the goals.

### 9.2 The Extracted OCaml Verifier

Thanks to the extraction facility of Coq, we can extract the verification function biginte_fun (and its underlying functions and data structures) in the OCaml language. The certified verifier is in file extracted.ml in [15]. We use OCaml-yacc to parse the input language (files lexer.mll and grammar.mly). The resulting verifier can handle three kind of goals: (1) arithmetic formulas (for which all variables are universally quantified), (2) entailments between assertions of Assert, and (3) separation logic triples. As the verification functions return a list of unsolved subgoals, the verifier is able to print these subgoals to help for the debugging of program specifications.

We measure the performance of the OCaml verifier. The first version uses a decision procedure for arithmetic based on variable elimination using the
Table 2 Execution Time

<table>
<thead>
<tr>
<th>Program</th>
<th>FMVE</th>
<th>Cooper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse list</td>
<td>0.240 s</td>
<td>0.111 s</td>
</tr>
<tr>
<td>List traversal</td>
<td>0.160 s</td>
<td>0.085 s</td>
</tr>
<tr>
<td>List append</td>
<td>147.593 s</td>
<td>0.660 s</td>
</tr>
<tr>
<td>Insert head</td>
<td>0.009 s</td>
<td>0.108 s</td>
</tr>
<tr>
<td>Insert tail</td>
<td>unknown</td>
<td>2.580 s</td>
</tr>
</tbody>
</table>

Fourier-Motzkin theorems (FMVE). This is a decision procedure by reflection that we have implemented for our verifier (the omega tactic of Coq cannot be used because it is not implemented by reflection). Of course, this decision procedure has also been verified in Coq (file expr_b_dp.v in [15]). The second version uses a non-certified decision procedure based on the Cooper algorithm [16]. The reason why we provide this second version is that our decision procedure for arithmetic, though necessary for use inside of Coq, is not optimized enough to solve large arithmetic subgoals. A certified implementation of a more efficient decision procedure (such as the Cooper algorithm) is among our future work (Chaieb and Nipkow already did this work in the Isabelle proof assistant [8]). Table 2 summarizes the measurements (hardware: Pentium IV 2.4GHz with 1GB of RAM).

Here follows a brief description of the benchmark programs: Reverse list is an in-place reversal of a list as the one described in [6], List traversal is a program that iteratively explores each element of a list, List append appends two lists, and Insert head (resp. Insert tail) inserts an element at the head (resp. tail) of a list.

The extracted verifier using the Cooper algorithm is available for download and testing through a Web interface, see [15].

10 Related Work

Our main contribution w.r.t. related work is to provide a certified automatic verifier for separation logic triples.

Berdine et al. have developed Smallfoot, a tool for checking separation logic specifications [11]. It uses symbolic, forward execution to produce verification conditions, and a decision procedure to prove them. Although Smallfoot is automatic (even for recursive and concurrent procedures), the assertion language does not allow to deal with pointer arithmetic.

Calcagno et al. have proposed an extension of Smallfoot to verify automatically memory allocators [12]. More precisely, the assertion language is extended with: arithmetic, advanced data-structures (lists with variable-size arrays), and abstract interpretation, allowing to compute automatically loop invariants. A prototype has been developed and used on several examples, such as the Kernighan allocator.

A verifier for separation logic with user-defined data-structure has been proposed in [14]. This verifier uses folding/unfolding of data-structures definitions to prove entailments. A prototype has been developed and used for verification of several functions with advanced invariants.

We believe that the algorithms implemented in these last two work are so complex that verification in Coq would be an order of magnitude harder than the work presented in this paper.

11 Conclusion

In this paper, we presented a verification procedure for a fragment of separation logic together with its verification in the Coq proof assistant. This verification procedure can be used both as a Coq tactic by reflection and as a stand-alone, certified and efficient verifier thanks to Coq extraction in OCaml. Our verifier is in many ways comparable to Smallfoot, the first automatic verifier for separation logic triples. Thus, we think that our work gives a good idea of the effort required to certify a state-of-the-art verifier for separation logic.

As for future work, we are interested in extending our fragment with commands for allocation of fresh memory and arrays.

References

[4] Yang, H.: An example of local reasoning in
BI pointer logic: the Schorr-Waite graph marking algorithm, in 1st Work. on Semantics, Program Analysis, and Computing Environments For Memory Management (SPACE 2001).


