# A Certified Verifier for a Fragment of Separation Logic*

Nicolas Marti† Reynald Affeldt‡

## Abstract

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 [4]. There exist several implementations of verifiers for separation logic [9, 10, 12], 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 [5, 9] 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 [9, 12], 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.

*This is a revised version of a paper presented at the 9th JSSST Workshop on Programming and Programming Languages (`http://www.sato.kuis.kyoto-u.ac.jp/ppl2007`).

†Department of Computer Science, University of Tokyo

‡Research Center for Information Security (RCIS), National Institute of Advanced Industrial Science and Technology (AIST)

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 [9, 12] 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:

```
nat_ind : ∀ (P : nat → Prop),
  P O → (∀ n, P n → P (S n)) → ∀ 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 `le` (noted $\leq$):

```
Inductive le (n : nat) : nat → Prop :=
  le_n : n ≤ n | le_S : ∀ m, n ≤ m → n ≤ S m.
```

The constructors of `le` can be seen as axioms whose application yields the proof of lemmas. For example, the proof-term `(le_S O O (le_n O))` of type `O ≤ S O` 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*. 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 ≤ 30.
  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 :=
  match x with
  | O => true
  | S x' => match y with
     O => false | S y' => leq_nat x' y' end
  end.
Lemma leq_nat_correct : ∀ x y,
  leq_nat x y = true → x ≤ y.
  ...
```

(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 ≤ 30.
  apply leq_nat_correct; auto.
Qed.
```

[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.

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.

# 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 [11]. 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 [13].

## 3.1 The Programming Language

Separation logic is an extention 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 [13]):

```
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 progam, and `s'` the resulting state (states are encoded with the `option` type whose `None` constructor represents error states). See [11] 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. 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 Q : assert) : assert :=
  fun s h => ∃ h1, ∃ h2,
    h1 # h2 ∧ h = h1 +++ h2 ∧ P s h1 ∧ Q s h2.
Notation "P ** Q" := (con P Q).
```

`P -* Q` holds when `Q` holds on the current heap extended with any (disjoint) heap for which `P` holds:

```
Definition imp (P Q : assert) : assert :=
  fun s h =>
    ∀ h', h # h' ∧ P s h' → Q s (h +++ h').
Notation "P -* Q" := (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 $\{P\}c\{Q\}$ is 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 Q:assert) : Prop :=
  ∀ s h,
    (P s h  →  ¬ Some (s, h) -- c --> None) ∧
    (∀ s' h', P s h →
      Some (s, h) -- c --> Some (s', h') →
        Q s' h').
```

The axiomatic semantics is defined as an inductive predicate whose constructors rephrase Reynolds axioms [4]:

```
Inductive semax : assert → cmd → assert → Prop :=
 ...
Notation "{{ P }} c {{ Q }}" := (semax P c Q).
```

We formally proved that `semax` is sound and complete w.r.t. `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 [5], 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 [12]. 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 [9].

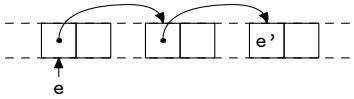
### 4.1 Syntax and Informal Semantics

Formulas of our fragment represent states symbolically. To represent a store symbolically, we use the language of boolean expressions `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:

```
Inductive Sigma : Set :=
| emp : Sigma
| singl : expr → expr → Sigma
| cell : expr → Sigma
| star : Sigma → Sigma → Sigma
| lst : expr → expr → Sigma.
```

Simply put, this syntax represents the connectives defined in Sect. 3.2: `emp` represents the empty heap like the homonym connective defined by shallow encoding; `singl` is syntax for `mapsto`; `cell e` represents a singleton heap whose contents is unknown; `star h h'` is the syntactic separating conjunction (Coq notation: `h**h'`; this is the same notation as the "semantic" separating conjunction in Sect. 3.2; in informal arguments, we will write $\star$ 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 `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:




To summarize, the syntax of our assertion language `assrt` is defined as a product of `expr_b` and `Sigma`:

```
Definition Pi := expr_b.
Definition assrt := Pi * Sigma.
```

In informal arguments, we will write $\langle \pi, \sigma \rangle$ 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, what 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 `Sigma -> store.s -> heap.h -> Prop` where `store.s -> heap.h -> Prop` is precisely the type `assert` of formulas in our shallow encoding:

```
Fixpoint Sigma_interp (a : Sigma) : assert :=
 match a with
  | emp => sep.emp
  | singl e1 e2 => fun s h =>
    (e1 |-> e2) s h ∧ eval e1 s ≠ 0
  | cell e => fun s h =>
    (∃ v, (e |-> int_e v) s h) ∧ eval e s ≠ 0
  | s1 ** s2 =>
    Sigma_interp s1 ** Sigma_interp s2
  | lst e1 e2 => Lst e1 e2
 end.
```

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

```
Inductive Lst : expr → expr → assert :=
| Lst_end: ∀ e e' s h,
  eval e s = eval e' s → sep.emp s h →
  Lst e e' s h
| Lst_next: ∀ e e' e'' data s h h1 h2,
  h1 # h2 → h = h1 +++ h2 →
  eval e s ≠ eval e' s →
  eval e s ≠ 0 →
  eval (e +e nat_e 1) s ≠ 0 →
  (e |-> e'' ** (e +e nat_e 1 |-> data)) s h1 →
  Lst e'' e' s h2 →
  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`):

```
Definition assrt_interp (a : assrt) : assert :=
  match a with
  (pi, sigm) => fun s h =>
    eval_pi pi s = true ∧ Sigma_interp sigm s h
 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 $\langle \pi_1, \sigma_1 \rangle \vee \ldots \vee \langle \pi_n, \sigma_n \rangle$ 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 entailment and without loss of expressivity. Indeed, all formulas belonging to a fragment with disjunction inside `Sigma` have a counterpart in our fragment. For instance, $\langle \pi, \sigma_1 \star (\sigma_2 \vee \sigma_3) \star \sigma_4 \rangle$ would have the same semantics as $\langle \pi, \sigma_1 \star \sigma_2 \star \sigma_4 \rangle \vee \langle \pi, \sigma_1 \star \sigma_3 \star \sigma_4 \rangle$. We encode disjunctions of assertions by lists:

```
Definition Assrt := list assrt.
```

Like for `assrt`, the semantics of `Assrt` is defined as a satisfiability relation, that is simply the disjunction of the satisfiability relations of the `assrt` disjuncts (function `Assrt_interp`, of type `Assrt -> assert`).

## 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 `assrt` (files `frag_list_entail.v` and `expr_b_dp.v` in [13]).

### 5.1 Entailment Proof System

Our proof system enables derivation of entailments of type `assrt -> assrt -> 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):

```
Goal entail
 (true_b, list e e' ** e' |-> e'' **
          cell (e'+1) ** list e'' 0)
 (true_b, list e 0).
   unfold e, e', e''; Entail.
Qed.
```

We have proved formally that the `entail` proof system is correct, i.e., that only valid entailments can be derived:

```
Lemma entail_soundness : ∀ P Q, entail P Q →
  assrt_interp P ==> assrt_interp Q.
```

We think that the `entail` proof system is also complete because it contains the rules of the proof system of [5], which is complete. An important point for this proof system to be complete is that it makes explicit the arithmetic constraints that are deducible from the `Sigma` formulas. There are two kinds of such constraints: (1) by definition of `cell` and `singl`, all cells locations are strictly positive integers (e.g., rule `singl_not_null`), and (2) cells on both sides of `star` have pairwise different locations (e.g., rule `star_neq`).

### 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 `elim_common_subheap` (Fig. 3), that tries to eliminate, subheap by subheap, the lhs `sig1 ** remainder` from the rhs `sig2`.

$$\frac{\langle \pi_1, \sigma_2 \star \sigma_1 \rangle \vdash \langle \pi, \sigma \rangle}{\langle \pi_1, \sigma_1 \star \sigma_2 \rangle \vdash \langle \pi, \sigma \rangle}\ \texttt{coml} \qquad \frac{\langle \pi_1 \wedge e_1 \neq 0, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}\ \texttt{singl\_not\_null}$$

$$\frac{\pi_1 \to \pi_2}{\langle \pi_1, \texttt{emp} \rangle \vdash \langle \pi_2, \texttt{emp} \rangle}\ \texttt{tauto} \qquad \frac{\langle \pi_1 \wedge e_1 \neq e_3, \sigma_1 \star e_1 \mapsto e_2 \star e_3 \mapsto e_4 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \star e_3 \mapsto e_4 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}\ \texttt{star\_neq}$$

$$\frac{\neg\, \pi_1}{\langle \pi_1, \texttt{emp} \rangle \vdash \langle \pi_2, \texttt{emp} \rangle}\ \texttt{incons} \qquad \frac{\pi_1 \to e_1 = e_3 \quad \pi_2 \to e_2 = e_4 \quad \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star \texttt{lst}\, e_1\, e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_3\, e4 \rangle}\ \texttt{lstsamelst}$$

$$\frac{\pi_1 \to e_1 = e_3 \quad \pi_1 \to e_2 = e_4 \quad \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star e_3 \mapsto e_4 \rangle}\ \texttt{star\_elim} \qquad \frac{\pi_1 \to e_1 = e_3 \quad \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \rangle}{\langle \pi_1, \sigma_1 \star \texttt{cell}\, e_1 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{cell}\, e_3 \rangle}\ \texttt{star\_elim''}$$

$$\frac{\begin{array}{c}\pi_1 \to e_1 = e_3 \\ \langle \pi_1, \sigma_1 \star \texttt{cell}\, e_4 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_2\, e_4 \rangle\end{array}}{\langle \pi_1, \sigma_1 \star \texttt{cell}\, e_4 \star \texttt{lst}\, e_1\, e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_3\, e_4 \rangle}\ \texttt{lstelim} \qquad \frac{\begin{array}{c}\pi_1 \to e_1 = e_3 \quad e_1 \neq e_4 \quad e_1 \neq 0 \\ \langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{cell}\, e_1 + 1 \star \texttt{lst}\, e_2\, e_4 \rangle\end{array}}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\, e_3\, e_4 \rangle}\ \texttt{lstelim'''}$$

Figure 1: Excerpt of the `entail` Proof System

$$\cfrac{\cfrac{\cfrac{\cfrac{\texttt{true\_b} \to \texttt{true\_b}}{\langle \texttt{true\_b}, \texttt{emp} \rangle \vdash \langle \texttt{true\_b}, \texttt{emp} \rangle}\ \texttt{tauto}}{\langle \texttt{true\_b}, \texttt{lst e'' 0} \rangle \vdash \langle \texttt{true\_b}, \texttt{lst e'' 0} \rangle}\ \texttt{lstsamelst}}{\langle \texttt{true\_b}, \texttt{cell e'+1 ** lst e'' 0} \rangle \vdash \langle \texttt{true\_b}, \texttt{cell e'+1 ** lst e'' 0} \rangle}\ \texttt{star\_elim''}}{\langle \texttt{true\_b}, \texttt{e'} \mapsto \texttt{e''} \texttt{ ** cell e'+1 ** lst e'' 0} \rangle \vdash \langle \texttt{true\_b}, \texttt{lst e' 0} \rangle}\ \texttt{lstelim'''}}{\langle \texttt{true\_b}, \texttt{lst e e' ** e'} \mapsto \texttt{e'' ** cell e'+1 ** lst e'' 0} \rangle \vdash \langle \texttt{true\_b}, \texttt{lst e 0} \rangle}\ \texttt{lstelim}$$

Figure 2: Example of Entailment: List Composition

This elimination is performed by the function `elim_common_cell` (Fig. 3), that 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:

```
elim_common_cell true_b (cell e'+1)
  (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 `assrt`). 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 rule `tauto` of 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

```
Fixpoint elim_common_subheap
 (pi : Pi) (sig1 sig2 remainder : Sigma)
 : option (Sigma * Sigma) :=
 match sig1 with
  | sig11 ** sig12 =>
    match elim_common_subheap pi sig11 sig2
          (sig12 ** remainder) with
     | None => None
     | Some (sig11', sig12') =>
        Some (remove_empty_heap pi
              (sig11' ** sig12), sig12')
    end
  | _ => elim_common_cell pi sig1 remainder sig2
 end.
```

```
Fixpoint elim_common_cell
 (pi : Pi) (sig remainder sig' : Sigma)
 : option (Sigma * Sigma) :=
 match sig' with
  ...
  | _ => ...
   match (sig, sig') with
    ...
    (* this case corresponds to the application
       of the rule lstelim''' *)
    | (singl e1 e2, lst e3 e4) =>
      if andb (expr_b_dp (pi =b> (e1 == e3)))
        (andb (expr_b_dp (pi =b> (e1 =/= e4)))
        (expr_b_dp (pi =b> (e1 =/= nat_e 0))))
      then Some
        (emp, (cell (e1 +e nat_e 1)) ** (lst e2 e4))
      else None
    ...
   end
 end.
```

Figure 3: Elimination of Common Subheaps

on Fourier-Motzkin variable elimination (this is actually the function `expr_b_dp` that already appears in Fig. 3).

This reasoning is implemented by the function `assrt_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 `assrt_entail_fun` to verify entailments of assertions (type `assrt`). Here, we explain how to use this function to verify entailments of disjunctions of assertions (type `Assrt`).

**Elimination of Disjunctions in the Lhs** To eliminate disjunctions in the lhs of the entailment we use the rule `elim_lhs_disj` (Fig. 4, function `Assrt_entail_Assrt_fun` in file `frag_list_entail.v`). Thanks to this rule, we can decompose an entailment between `Assrt` formulas into a list of entailments between an `assrt` formula (on the lhs) and an `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 `elim_rhs_disj1` (Fig. 4, function `orassrt_impl_Assrt1` in file `frag_list_entail.v`). But this rule is not sufficient, as illustrated by the following counter-example:

$$\langle \texttt{true\_b}, \sigma \rangle \vdash \langle \texttt{y} = 0, \sigma \rangle \vee \langle \texttt{y} \neq 0, \sigma \rangle$$

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_disj2` (Fig. 4, functions `orpi` and `orassrt_impl_Assrt2` in file `frag_list_entail.v`).

$$\frac{\bigwedge_i \left(\langle \pi_i, \sigma_i \rangle \vdash A\right)}{\left(\bigvee_i \langle \pi_i, \sigma_i \rangle\right) \vdash A}\ \texttt{elim\_lhs\_disj}$$

$$\frac{\bigvee_i \left(\langle \pi, \sigma \rangle \vdash \langle \pi_i, \sigma_i \rangle\right)}{\langle \pi, \sigma \rangle \vdash \left(\bigvee_i \langle \pi_i, \sigma_i \rangle\right)}\ \texttt{elim\_rhs\_disj1}$$

$$\frac{\pi \rightarrow \left(\bigvee_i \pi_i\right) \quad \bigwedge_i \left(\langle \pi \wedge \pi_i, \sigma \rangle \vdash \langle \texttt{true\_b}, \sigma_i \rangle\right)}{\langle \pi, \sigma \rangle \vdash \left(\bigvee_i \langle \pi_i, \sigma_i \rangle\right)}\ \texttt{elim\_rhs\_disj2}$$

Figure 4: Entailment of Disjunctions of Assertions

We are now equipped to explain the function `entail_fun`, that proves the validity of entailments. It takes as input an `assrt` 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:assrt) (A:Assrt) (l:list (assrt * assrt))
  : result (list (assrt * assrt)) := ...
```

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 l,
  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 [13]).

## 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 [4]:

```
Fixpoint wpAssrt_interp (a: wpAssrt) : assert :=
 match a with
| wpElt a1 => Assrt_interp a1
| wpSubst l L =>
  subst_lst2update_store l (wpAssrt_interp L)
| wpLookup x e L => (fun s h => ∃ e0,
  (e |-> e0 ** (e |-> e0 -*
   update_store2 x e 0 (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_pi 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 `assrt -> 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.
```

$$\frac{\langle \pi_1, \sigma_1 \rangle \vdash \langle \pi_2[v_n/x_n] \cdots [v_1/x_1], \sigma_2[v_n/x_n] \cdots [v_1/x_1] \rangle}{\{\pi_1, \sigma_1\} x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n \{\pi_2, \sigma_2\}} \texttt{ subst}$$

$$\frac{\pi_1 \rightarrow v_1 = e_1 \quad \{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x_1 \leftarrow e_2; c\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x_1 \leftarrow\!\!* \; v_1; c\{\pi_2, \sigma_2\}} \texttt{ lookup}$$

$$\frac{\begin{array}{c}\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x' \leftarrow e_2; x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; x \leftarrow x'; c\{\pi_2, \sigma_2\} \\ \pi_1 \rightarrow e_1 = e[v_n/x_n] \cdots [v_1/x_1] \quad \text{fresh } x'\end{array}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; x \leftarrow\!\!* \; e; c\{\pi_2, \sigma_2\}} \texttt{ subst\_lookup}$$

$$\frac{\pi_1 \rightarrow v_1 = e_1 \quad \{\pi_1, \sigma_1 \star e_1 \mapsto v_2\} c\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} v_1 \mathrel{*\!\!\leftarrow} v_2; c\{\pi_2, \sigma_2\}} \texttt{ mutation}$$

$$\frac{\{\pi_1, \sigma_1\} e[v_n/x_n] \cdots [v_1/x_1] \mathrel{*\!\!\leftarrow} e'[v_n/x_n] \cdots [v_1/x_1]; x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; c\{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1\} x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; e \mathrel{*\!\!\leftarrow} e'; c\{\pi_2, \sigma_2\}} \texttt{ subst\_mutation}$$

$$\frac{\{\pi_1 \wedge b, \sigma_1\} c_1 \{\pi_2, \sigma_2\} \qquad \{\pi_1 \wedge \neg b, \sigma_1\} c_2 \{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1\} \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \{\pi_2, \sigma_2\}} \texttt{ if}$$

$$\frac{\{\pi_1, \sigma_1\} \texttt{if } b[v_n/x_n] \cdots [v_1/x_1] \texttt{ then } x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; c_1 \texttt{ else } x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; c_2 \{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1\} x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \{\pi_2, \sigma_2\}} \texttt{ subst\_if}$$

Figure 5: Excerpt of the `tritra` Proof System

### 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_dp`. This operation is implemented by the function `tritra_step` of type `Pi -> Sigma -> wpAssrt -> option (list ((Pi * Sigma) * 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:

```
Fixpoint triple_transformation
  (P : Assrt) (Q : wpAssrt) { struct P }
  : option (list ((Pi * Sigma) * wpAssrt)) := ...

Lemma triple_transformation_correct: ∀ P Q,
  triple_transformation P Q = Some nil  ->
  Assrt_interp P ==> wpAssrt_interp Q.
```

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 [13]).

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 $\{P\}c_1;\texttt{while}_I\ b\ \texttt{do}\ c;c_2\{Q\}$ where $I$ is an invariant. To prove this triple, it is sufficient to prove the three triples $\{P\}c_1\{I\}$, $\{I \wedge b\}c\{I\}$, and $\{I \wedge \neg b\}c_2\{Q\}$. Applying this idea repeatedly turns a separation logic triple into a set of loop-free triples, as implemented by the following function:

```
Fixpoint vcg (c:cmd') (Q:wpAssrt) { struct c }
  : option (wpAssrt * (list (Assrt * wpAssrt)))
  := ...
```

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 `{{·}}·{{·}}`; `Assrt_interp` and `wpAssrt_interp` were defined respectively in Sections 4.3 and 6.1:

```
Lemma vcg_correct : ∀ c Q Q' l,
 vcg c Q = Some (Q', l) →
 (∀ A L, In (A, L) l →
   Assrt_interp A ==> wpAssrt_interp L) →
 {{ 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)) :=
  match vcg c (wpElt Q) with
    | None => None
    | Some (Q', l) =>
      match triple_transformation P Q' with
        | Some l' =>
          match 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: ∀ 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 [9, 10, 12]. 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 [9]). Although decidability results exist [3, 8, 7], the separating implication is actually seldom used in specifications of algorithms (one notable exception is [2]). 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 [9] extended with pointer arithmetic (file `LSF.v` in [13]). 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 occurences 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[2]. 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.

[2] As there is no universal quantification in our assertion language, the behavior of `init(n)` is specified for only one arbitrary value, and the programs and pre/postconditions are computed by Coq functions.

| Program | `tritra` | forward reasoning |
|---|---|---|
| `swap` | 16 | 20 (+19%) |
| `init (5)` | 46 | 69 (+33%) |
| `init (10)` | 138 | 225 (+38%) |
| `init (15)` | 195 | 320 (+39%) |
| `3max` | 9.0 | 7.9 (−14%) |

Table 1: Size of Proof-terms files in kbytes

## 9.2 The Extracted OCaml Verifier

Thanks to the extraction facility of Coq, we can extract the verification function `bigtoe_fun` (and its underlying functions and data structures) in the OCaml language. The certified verifier is in file `extracted.ml` in [13]. 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 `Assrt`, 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 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 [13]). The second version uses a non-certified decision procedure based on the Cooper algorithm [14]. 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 [6]). Table 2 summarizes the measurements (hardware: Pentium IV 2.4GHz with 1GB of RAM).

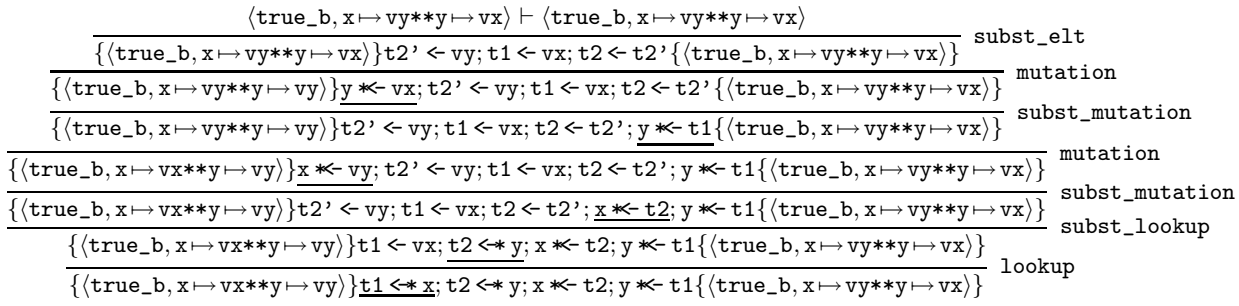



Figure 6: Swap of Cells using our Proof System

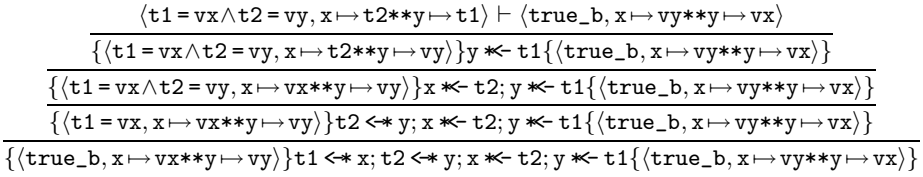



Figure 7: Swap of Cells using Forward Reasoning

| Program | FMVE | Cooper |
|---|---|---|
| Reverse list | 0.240 s | 0.111 s |
| List traversal | 0.160 s | 0.085 s |
| List append | 147.593 s | 0.660 s |
| Insert head | 0.009 s | 0.108 s |
| Insert tail | unknown | 2.580 s |

Table 2: Execution Time

Here follows a brief description of the benchmark programs: `Reverse list` is an in-place reversal of a list as the one described in [4], `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 [13].

# 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 [9]. 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 [10]. 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 [12]. This verifier uses folding/unfolding of data-structures definitions to prove entailments. A prototype has been developed and used for verification of several functions

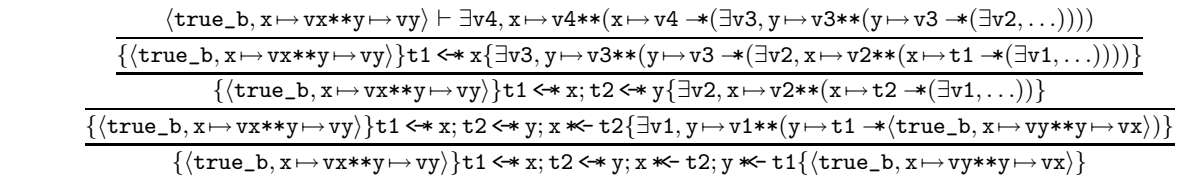



Figure 8: Swap of Cells using Backward Reasoning

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

[1] The LogiCal Project, INRIA. The Coq Proof Assistant. `http://coq.inria.fr`.

[2] H. Yang. 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).*

[3] C. Calcagno, H. Yang and P. W. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *21st Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2001)*, LNCS vol. 2001, p. 108–119, Springer.

[4] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symp. on Logic in Computer Science (LICS 2002)*, p. 55–74.

[5] J. Berdine, C. Calcagno and P. W. O'Hearn. A Decidable Fragment of Separation Logic. In *24th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, LNCS vol. 3328, p. 97–109, Springer.

[6] A. Chaieb and T. Nipkow. Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic. In *12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, LNCS vol. 3835, p. 367–380, Springer.

[7] D. Galmiche and D. Méry. Characterizing Provability in BI's Pointer Logic through Resource Graphs. In *12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, LNCS vol. 3835, p. 459–473, Springer.

[8] C. Calcagno, P. Gardner and M. Hague. From Separation Logic to First-Order Logic. In *8th Int. Conf. on Foundations of Software Science and Computational Structures (FOSSACS 2005)*, LNCS vol. 3441, p. 395–409, Springer.

[9] J. Berdine, C. Calcagno and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *3rd Asian Symp. on Programming Languages and Systems (APLAS 2005)*, LNCS vol. 3780, p. 52–68, Springer.

[10] C. Calcagno, D. Distefano, P. W. O'Hearn and H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *13th Int. Symp. on Static Analysis (SAS 2006)*, LNCS vol. 4134, p. 182–203, Springer.

[11] N. Marti, R. Affeldt and A. Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th Int. Conf. on Formal Engineering Methods (ICFEM 2006)*, LNCS vol. 4260, p. 400–419, Springer.

[12] H. H. Nguyen, C. David, S. Qin and W. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *8th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, LNCS vol. 4349, Springer.

[13] R. Affeldt and N. Marti. Separation Logic in Coq. `http://www.nongnu.org/seplog/`.

[14] J. Harrison. Cooper's Algorithm for Presburger Arithmetic. `http://www.cl.cam.ac.uk/~jrh13/atp`.