A Certified Verifier for a Fragment of Separation Logic

Nicolas Marti* Reynald Affeldt†

Abstract

Separation logic is an extension of Hoare logic that is convenient 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 propose 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 derive a certified, stand-alone, and efficient verifier for separation logic.

1 Introduction

Separation logic is an extension of Hoare logic that has proved convenient to verify imperative programs with pointers and mutable data structures [4]. There exist several implementations of verifiers for separation logic [9, 10, 12]. However, 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 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 difficult 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.

Our contribution is to develop and verify in the Coq proof assistant [1] a 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, this verifier can also be used as a certified, stand-alone, and efficient verifier.

*Department of Computer Science, University of Tokyo

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

In this paper, we explain the implementation and the verification of our verifier in the Coq proof assistant. The verifier is organized in several phases for each of which we provide a correctness lemma proved in Coq. Though our verifier is not as versatile as existing prototypes, we believe that it provides a good evaluation of the effort required by formal verification and it provides a good basis for further extensions.

The goal of our verifier is to prove automatically separation logic triples $\{P\}c\{Q\}$, where c is a command, P and Q are assertions of separation logic. For the assertions, we cannot use the full separation logic language because 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 handle 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.

Our verifier is organized in three successive phases:
1. *Verification conditions generator*: The input separation logic 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.

The rest of this paper essentially consists of the successive explanation of each phase. In Sect. 5, we explain the entailment phase. In Sect. 6, we explain the triple transformation phase. In Sect. 7, we explain the verification conditions generator. The resulting tactic 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 (by comparison with the forward reasoning approach) and performance benchmarks for the derived stand-alone OCaml verified. Sect. 10 is dedicated to comparison with related work. We conclude in Sect. 11.

2 Background: Tactics in Coq

There are two ways to verify goals in the Coq proof assistant: apply successively lemmas until the goal assertion is proved, or develop and certify a decision procedure as a Coq function. The latter method is referred to as *reflection*. Its main benefit is that generated proof-terms are small, because the tactic amounts to computing the return value of a function. (Another advantage is that it allows for formal proof in Coq of the completeness of the decision procedure but we are not concerned with this aspect in this paper.) Let us illustrate this with an example. The most direct way to prove an inequality over naturals in Coq is to use Coq native tactics:

```
Goal 18 <= 30.
repeat constructor.
Qed.
```

Intuitively, `repeat constructor` is a tactic to solve inequalities over naturals. It has the defects to generate large proof-term (this can be checked with `Show Proof`) and to forbid meta-reasoning (there is no way to prove inside Coq the completeness of this tactic). 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 : forall x y,
 leq_nat x y = true -> x <= y.
```

Then, our goal can be proved by applying the correctness lemma:

```
Goal 18 <= 30.
apply leq_nat_correct; auto.
Qed.
```

Decision procedures implemented by reflection lead to short proof-terms and are amenable to completeness proofs. The downside is a more difficult 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 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` (respectively `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 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).

3.2 The Assertion Language

The assertion language is defined on top of the propositional language of Coq (the `Prop` type). Precisely, an assertion is defined as a function from states to Coq propositions:

```
Definition assert := store.s -> heap.h -> Prop.
```

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 :=
  forall 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: `-*`).

```
Definition emp : assert := fun s h => h=heap.emp.

Definition mapsto e1 e2 : assert :=
fun s h => exists p, val2loc (eval e1 s) = p /\
  h = heap.singleton p (eval e2 s).
Notation "e1 '|->' e2" := (mapsto e1 e2).
```

`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 con (P Q : assert) : assert :=
fun s h => exists h1, exists h2,
  h1 # h2 /\ h = h1 +++ h2 /\ P s h1 /\ Q s h2.
Notation "P ** Q" := (con P Q).
```

`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.

The separating implication `P -* Q` holds when `Q` holds on the current heap extended with any heap for which `P` holds. This connective is essential for reasoning because it captures logically the notion of destructive update. But it is seldom used in specifications because its semantics is not intuitive.

3.3 Separation Logic Triples

The semantics of separation logic triples $\{P\}c\{Q\}$ is defined as follows: considering the program c, for every initial state such that the precondition P holds, (1) the execution ends in a valid state, and (2) the postcondition Q holds for every final state. Put formally in Coq:

```
Definition semax' (c:cmd) (P Q :assert): Prop :=
 forall s h,
  (P s h -> ~exec (Some (s, h)) c None) /\
  (forall s' h', P s h ->
   exec (Some (s, h)) c (Some (s', h')) ->
    Q s' h').
```

where `exec` is a big-step operational semantics.

The axiomatic semantics is defined as a Coq inductive type 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. Basically, this is 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].

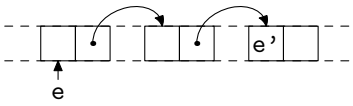
4.1 Syntax and Informal Semantics

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'`; in informal arguments, we will write $\star$ for the separating conjunction). In particular, `Sigma` does not contain the separating implication of separation logic. The novely w.r.t. the shallow encoding of Sect. 3.2 is 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 := prod Pi Sigma.
```

In informal arguments, we will write $\langle \pi, \sigma \rangle$ for assertions.

In this section, contrary to Sect. 3.2, we have defined the syntax of formulas directly in Coq. In the next section, we define their semantics by a satisfiability relation. This technique of encoding is called *deep encoding* and is typical of tactics by reflection. Indeed, the latter need to "parse" the assertion language to prove the validity of formulas, what is difficult to do when the syntax is not an inductive type.

4.2 Formal Semantics

The formal semantics of `Sigma` is defined by a satisfiability relation between (syntactic) formulas and states. We define this relation by a function `Sigma_interp` of type `Sigma -> store.s -> heap.h -> Prop`. Since `store.s -> heap.h -> Prop` is precisely the type `assert` of formulas in our shallow encoding, we can use the connective defined by shallow encoding to define the semantics of `Sigma` formulas:

```
Fixpoint Sigma_interp (a : Sigma) : assert :=
 match a with
  | singl e1 e2 =>
    fun s h => (e1 |-> e2) s h /\ eval e1 s <> 0
  | cell e => fun s h => (exists v,
    (e |-> int_e v) s h) /\ eval e s <> 0
  | emp => sep.emp (* the module sep contains
      the shallow encoding *)
  | star s1 s2 => Sigma_interp s1 **
    Sigma_interp s2
  | lst e1 e2 => Lst e1 e2
 end.
```

where `Lst` is an inductive type of the appropriate type defining singly-linked lists:

```
Inductive Lst : expr -> expr -> assert :=
| Lst_end: forall e1 e2 s h,
  eval e1 s = eval e2 s ->
  sep.emp s h ->
  Lst e1 e2 s h
| Lst_suiv: forall e1 e2 e3 e4 s h h1 h2,
  h1 # h2 -> h = h1 +++ h2 ->
  eval e1 s <> eval e3 s ->
  eval e1 s <> 0 ->
  eval (e1 +e nat_e 1) s <> 0 ->
  (e1 |-> e2 ** (e1 +e nat_e 1 |-> e4)) s h1 ->
  Lst e2 e3 s h2 ->
  Lst e1 e3 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. 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`).

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.

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 that prove that validity for the entailment between two formulas of type `assrt`. This section has been written so as to be self-explanatory; more details can be found in the files `frag_list_entail.v` and `expr_b_dp.v` in [13].

5.1 Entailment Proof System

The proof system we provide is a set of rules to derive entailments of the form `assrt -> assrt -> Prop` such that the lhs (left hand side) semantically implies the rhs (right hand side). The proof system in Coq takes the form of a Coq inductive type `entail`. An excerpt is displayed in Fig. 2. To ease reading, it is reproduced informally in Fig. 1. Most rules are fairly intuitive. For example, we can take a look at the constructor `entail_coml`, that captures the fact that the separating conjunction is commutative on the left of implication.

$$\frac{\texttt{neg\_b}\ \pi_1}{\langle \pi_1, \texttt{emp}\rangle \vdash \langle \pi_2, \texttt{emp}\rangle}\ \texttt{entail\_incons}$$

$$\frac{\pi_1 \rightarrow \pi_2}{\langle \pi_1, \texttt{emp}\rangle \vdash \langle \pi_2, \texttt{emp}\rangle}\ \texttt{entail\_tauto}$$

$$\frac{\langle \pi_1, \sigma_2 \star \sigma_1\rangle \vdash \langle \pi, \sigma\rangle}{\langle \pi_1, \sigma_1 \star \sigma_2\rangle \vdash \langle \pi, \sigma\rangle}\ \texttt{entail\_coml}$$

$$\frac{\Pi_1 \rightarrow e_1 = e_3 \wedge e_1 \neq e_4 \wedge e_1 \neq 0 \quad \langle \pi_1, \sigma_1\rangle \vdash \langle \pi_2, \sigma_2 \star e_1{+}1 \mapsto \_ \star \texttt{lst}\ e_2\ e_4\rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2\rangle \vdash \langle \pi_2, \sigma_2 \star \texttt{lst}\ e_3\ e_4\rangle}\ \texttt{entail\_lstelim'''}$$

$$\frac{\langle \pi_1 \wedge e_1 \neq e_3, \sigma_1 \star e_1 \mapsto e_2 \star e_3 \mapsto e_4\rangle \vdash \langle \pi_2, \sigma_2\rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2 \star e_3 \mapsto e_4\rangle \vdash \langle \pi_2, \sigma_2\rangle}\ \texttt{entail\_sepcon\_neq}$$

$$\frac{\langle \pi_1 \wedge e_1 \neq 0, \sigma_1 \star e_1 \mapsto e_2\rangle \vdash \langle \pi_2, \sigma_2\rangle}{\langle \pi_1, \sigma_1 \star e_1 \mapsto e_2\rangle \vdash \langle \pi_2, \sigma_2\rangle}\ \texttt{entail\_singl\_not\_null}$$

Figure 1: `entail` Proof System (informal excerpt)

We have implemented a tactic (`Entail`, not extractable) that iteratively applies the constructors of `entail` to solve entailments. Here follows an example of such a goal; Fig. 3 provides an informal account of the proof built underneath:

```
Inductive entail : assrt -> assrt -> Prop :=
| entail_incons : forall pi1 pi2 sig1 sig2,
  (forall s, eval_pi pi1 s = true -> False) ->
  entail (pi1, sig1) (pi2, sig2)
| entail_tauto : forall pi1 pi2,
  (forall s, eval_pi pi1 s = true ->
       eval_pi pi2 s = true) ->
  entail (pi1, emp) (pi2, emp)
| entail_coml : forall pi1 sig1 sig2 L,
  entail (pi1, sig2 ** sig1) L ->
  entail (pi1, sig1 ** sig2) L
| entail_lstelim''' :
  forall pi1 sig1 pi2 sig2 e1 e2 e3 e4,
  (forall s, eval_pi pi1 s = true ->
       eval_pi (e1 == e3) s = true) ->
  (forall s, eval_pi pi1 s = true ->
       eval_pi (e1 =/= e4) s = true) ->
  (forall s, eval_pi pi1 s = true ->
       eval_pi (e1 =/= int_e 0%Z) s = true) ->
  entail (pi1, sig1) (pi2, sig2 **
    ((cell (e1 +e nat_e 1)) ** (lst e2 e4))) ->
  entail (pi1, sig1 ** (singl e1 e2))
    (pi2, sig2 ** (lst e3 e4))
| entail_star_elim :
  forall pi1 pi2 sig1 sig2 e1 e2 e3 e4,
  (forall s, eval_pi pi1 s = true ->
       eval_pi (e1 == e3) s = true) ->
  (forall s, eval_pi pi1 s = true ->
       eval_pi (e2 == e4) s = true) ->
  entail (pi1, sig1) (pi2, sig2) ->
  entail (pi1, sig1 ** (singl e1 e2))
    (pi2, sig2 ** (singl e3 e4))
| entail_sepcon_neq :
  forall pi1 sig1 pi2 sig2 e1 e2 e3 e4,
  entail (pi1 &&& (e1 =/= e3), sig1 **
    (singl e1 e2 ** singl e3 e4)) (pi2, sig2) ->
  entail (pi1, sig1 **
    ((singl e1 e2) ** (singl e3 e4))) (pi2, sig2)
| entail_singl_not_null :
  forall pi1 sig1 pi2 sig2 e1 e2,
  entail (pi1 &&& (e1 =/= (nat_e 0)),
    sig1 ** (singl e1 e2)) (pi2, sig2) ->
  entail (pi1, sig1 ** (singl e1 e2)) (pi2, sig2)
| ...
```

Figure 2: `entail` Proof System (Coq excerpt)

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

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

```
Lemma entail_soundness : forall P Q,
 entail P Q ->
  assrt_interp P ==> assrt_interp Q.
```

We think that the `entail` relation is also complete. An important point for this relation 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 `entail_singl_not_null`), and (2) cells on both sides of `star` have pairwise different locations (e.g., rule `entail_sepcon_neq`).

5.2 Entailment Verification Procedure

In this section, we explain a Coq function (`entail_dp`) that proves entailments. Because it is implemented as a certified function, it can be used as a tactic by reflection. It implements a reasoning similar to the proof system in the previous section but this is no redundant work: we will actually use the `Entail` tactic to prove the correctness of the `entail_dp` function.

5.2.1 Implication Between Heaps

In this section, we explain a function `Sigma_impl` to prove the validity of implication between two abstract heaps. This function iteratively calls the function `elim_common_subheap` (Fig. 4), that tries to eliminate, subheap by subheap, the lhs `star sig1 remainder` from the rhs `sig2`. This elimination is performed by the function `elim_common_cell` (Fig. 5), that tries to remove the subheap `sig` from both `star sig sig1` and `sig2`. It is essentially a case-analysis on both heaps leading to the application of an `entail` rule. For instance, Fig. 5 shows the case for which the rule `entail_lstelim'''` is applied.

5.2.2 Entailments Between Assertions

In the previous section, we explained a function `Sigma_impl` to prove the validity of the implication between two abstract heaps. In this section, 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 `entail_incons`. 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 `entail_tauto`. 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 (function `expr_b_dp`).

The above reasoning is implemented by the function `assrt_entail_dp` that extends beforehand the lhs of the entailment with arithmetic constraints, as de-

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

Figure 4: Elimination of Common Subheaps

```
Fixpoint elim_common_cell (pi : Pi) (sig sig1
  sig2 : Sigma) : option (Sigma * Sigma) :=
 match sig2 with
  ...
  | _ => match (sig, sig2) with
  ...
(* this case correspond to the application
   of the constructor entail_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, star
          (cell (e1 +e nat_e 1)) (lst e2 e4))
  else None
         end
 end.
```

Figure 5: Elimintation of Common Cells

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{true\_b} \to \texttt{true\_b}}{\langle \texttt{true\_b}, \texttt{emp}\rangle \vdash \langle \texttt{true\_b}, \texttt{emp}\rangle}\ \texttt{entail\_tauto}}{\langle \texttt{true\_b}, \texttt{lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{lst e'' 0}\rangle}\ \texttt{entail\_lstsamelst}}{\langle \texttt{true\_b}, \texttt{cell e'+1**lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{cell e'+1**lst e'' 0}\rangle}\ \texttt{entail\_star\_elim'''}}{\langle \texttt{true\_b}, \texttt{e'} \mapsto \texttt{e''**cell e'+1**lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{lst e' 0}\rangle}\ \texttt{entail\_elim'''}}{\langle \texttt{true\_b}, \texttt{lst e e'**e'} \mapsto \texttt{e''**cell e'+1**lst e'' 0}\rangle \vdash \langle \texttt{true\_b}, \texttt{lst e 0}\rangle}\ \texttt{entail\_lstelim}$$

Figure 3: Example of Entailment—List Composition

scribed a the end of Sect. 5.1.

5.2.3 Entailments Between Disjunctions

In the previous section, we explained a function `assrt_entail_dp` to verify entailments of assertions (type `assrt`). In this section, we explain how to use this function to verify entailments of disjunctions of assertions (type `Assrt`). For this purpose, we use the set of rules of Fig. 6. Let us explain these rules in details.

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

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

$$\dfrac{\begin{array}{c}\pi \to \left(\bigvee_i \pi_i\right) \\ \bigwedge_i (\langle \pi \wedge \pi_i, \sigma\rangle \vdash \langle \texttt{true\_b}, \sigma_i\rangle)\end{array}}{\langle \pi, \sigma\rangle \vdash \left(\bigvee_i \langle \pi_i, \sigma_i\rangle\right)}\ \texttt{elim\_rhs\_disj2}$$

Figure 6: Entailment of Disjunctions of Assertions

Elimination of Disjunctions in the Lhs To eliminate disjunctions in the lhs of the entailment we use the rule `elim_lhs_disj`. Thanks to this rule, we can decompose entailment between `Assrt` formulas into a list of entailments between an `assrt` formula (on the lhs) and an `Assrt` formula (on the rhs) (see function `Assrt_entail_Assrt_dp` in [13]).

Elimination of Disjunctions in the Rhs The elimination of disjunctions in the rhs of the entailment is more delicate. It is possible to use the rule `elim_rhs_disj1` (see function `orassrt_impl_Assrt1` in [13]). But this rule is not complete, as shown 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 of loop invariants. To resolve them, we use the rule `elim_rhs_disj2` (see functions `orpi` and `orassrt_impl_Assrt2` in [13]). In practice, the combined use of these two rules is sufficient.

The `entail_dp` function is the function that proves the validity of entailments. It takes as input an `assrt` and an `Assrt`, uses the rules from Fig. 6 to eliminate the disjunctions in the rhs, and calls `assrt_entail_dp` from the previous section:

```
Definition entail_dp
  (a:assrt) (A:Assrt) (l:list (assrt * assrt))
  : result (list (assrt * assrt)) := ...
```

The `entail_dp` function returns an option type (constructor `Good` if everything is proved). The proof of correctness of `entail_dp` amounts to the following lemma:

```
Lemma entail_dp_correct: forall A a l,
  entail_dp a A l = Good ->
  assrt_interp a ==> Assrt_interp A.
```

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 separation logic triple into such an entailment. This section has been written so as to be self-explanatory; additional details can be found in the 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 => exists e0,
  (e |-> e0 ** (e |-> e0 -*
   update_store2 x e 0 (wpAssrt_interp L))) s h)
| wpMutation e1 e2 L => (fun s h => exists 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. Fig. 7 contains an informal excerpt of the proof system; for reference, the formal Coq counterpart appears in Fig. 8.

The two rules `tritra_lookup` and `tritra_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 `tritra_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:

$$\frac{\pi_1 \to v_1 = e_1 \qquad \{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x_1 \leftarrow e_2; c \{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x_1 \leftarrow *v_1; c \{\pi_2, \sigma_2\}}\ \texttt{tritra\_lookup}$$

$$\frac{\pi_1 \to v_1 = e_1 \qquad \{\pi_1, \sigma_1 \star e_1 \mapsto v_2\} c \{\pi_2, \sigma_2\}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} v_1 * \leftarrow v_2; c \{\pi_2, \sigma_2\}}\ \texttt{tritra\_mutation}$$

$$\frac{\begin{array}{c}\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} \\ x' \leftarrow e_2; x_1 \leftarrow v_1; \ldots; x_n \leftarrow v_n; x \leftarrow *e; c \\ \{\pi_2, \sigma_2\} \\ \pi_1 \to e_1 = e[v_n/x_n] \ldots [v_1/x_1]\end{array}}{\{\pi_1, \sigma_1 \star e_1 \mapsto e_2\} x_1 \leftarrow v_1; \ldots; x_n \leftarrow v_n; x \leftarrow *e; c \{\pi_2, \sigma_2\}}\ \texttt{tritra\_subst\_lookup}$$

Figure 7: `tritra` Proof System (informal excerpt)

```
Lemma tritra_soundness :
 forall 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 $P \to Q'$ (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_dp` on resulting subgoals:

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

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

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

```
Inductive tritra : assrt -> wpAssrt -> Prop :=
| tritra_lookup : forall pi1 sig1 e1 e2 e x L,
  (forall s, eval_pi pi1 s = true ->
   eval_pi (e1 == e) s = true) ->
  tritra (pi1, sig1 ** (singl e1 e2))
   (wpSubst ((x,e2)::nil) L) ->
  tritra (pi1, sig1 ** (singl e1 e2))
   (wpLookup x e L)
| tritra_mutation :
  forall pi1 sig1 e1 e2 e3 e4 L,
  (forall s, eval_pi pi1 s = true ->
   eval_pi (e1 == e3) s = true) ->
  tritra (pi1, sig1 ** (singl e1 e4)) L ->
  tritra (pi1, sig1 ** (singl e1 e2))
    (wpMutation e3 e4 L)
| tritra_subst_lookup :
  forall pi1 sig1 e1 e2 e x x' l L,
  (forall s, eval_pi pi1 s = true ->
   eval_pi (e1 == (subst_e_lst l e)) s = true)
  -> fresh_lst x' l -> fresh_wpAssrt x' L
  -> fresh_e x' (var_e x) ->
  tritra (pi1, sig1 ** (singl e1 e2))
   (wpSubst ((x,var_e x')::l++((x',e2)::nil)) L)
  -> tritra (pi1, sig1 ** (singl e1 e2))
   (wpSubst l (wpLookup x e L))
| ...
```

Figure 8: `tritra` Proof System (Coq excerpt)

loops are annotated with invariants into a list of loop-free triples. Here again, this section has been written so as to be self-explanatory; additional details can be found in the file `frag_list_vcg.v` in [13].

The generation of loop-free triple 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 $\{P\}c_1;\texttt{while}_I\ b\ \texttt{do}\ c;c_2\{Q\}$ (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 recursively turns a separation logic triple into a set of loop-free triples, as implemented by the function `vcg`:

```
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).

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 $\{\{\cdot\}\}\cdot\{\{\cdot\}\}$; `Assrt_interp` and `wpAssrt_interp` were defined respectively in Sections 4.3 and 6.1.

```
Lemma vcg_correct : forall c Q Q' l,
 vcg c Q = Some (Q', l) ->
 (forall A L, In (A, L) l ->
   Assrt_interp A ==> wpAssrt_interp L) ->
 {{ wpAssrt_interp Q' }}
 proj_cmd c
 {{ wpAssrt_interp Q }}.
```

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_dp (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_dp_correct: forall P Q c,
  bigtoe_dp 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_dp`.

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 (the only exception is [2]). Yet, forward reasoning has the disadvantage of adding a conjunctive clause, with possibly a fresh variable, for each variable modification. 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. 9, 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 10 and 11. 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. 9. 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/post-condition. 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.

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_dp` (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 (see files `lexer.mll` and `grammar.mly`). The resulting verifier can handle three kind of goals: (1) arithmetic formulas (for which all variable 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 (environment: Pentium IV 2.4GHz with 1GB of RAM).

Here follows a brief description of the benchmark

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

programs: `Reverse list` is an in-place reverseal of a list as the one described in [4], `List traversal` is a program that iteratively explores each element of a list, `List append` merges 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 decision procedure for separation logic with user-defined data-structure has been proposed in [12]. This decision procedure 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 a certified implementation 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 triples.

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

References

[1] Various contributors. The Coq Proof Assistant. `http://coq.inria.fr`.

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

[3] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2001)*, volume 2001 of Lecture Notes in Computer Science, p. 108–119, Springer-Verlag.

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

[5] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A Decidable Fragment of Separation Logic. In *24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, volume 3328 of Lecture Notes in Computer Science, p. 97–109, Springer-Verlag.

[6] Amine Chaieb and Tobias Nipkow. Verifying and Reflecting Quantifier Elimination for Presburger

Arithmetic. In *12th Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of Lecture Notes in Computer Science, p. 367–380, Springer-Verlag.

[7] Didier Galmiche and Dominique Méry. Characterizing Provability in BI's Pointer Logic through Resource Graphs. In *12th Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of Lecture Notes in Computer Science, p. 459–473, Springer-Verlag.

[8] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From Separation Logic to First-Order Logic. In *8th Conference on Foundations of Software Science and Computational Structures (FOSSACS 2005)*, volume 3441 of Lecture Notes in Computer Science, p. 395–409, Springer-Verlag.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In *3rd Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of Lecture Notes in Computer Science, p. 52–68, Springer-Verlag.

[10] Cristiano Calcagno, Dino Distefano, Peter O'Hearn and Hongseok Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *13th Symposium on Static Analysis (SAS 2006)*, volume 4134 of Lecture Notes in Computer Science, p. 182–203, Springer-Verlag.

[11] Nicolas Marti, Reynald Affeldt and Akinori Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of Lecture Notes in Computer Science, p. 400–419, Springer-Verlag.

[12] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *8th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, volume 4349 of Lecture Notes in Computer Science, Springer-Verlag.

[13] Reynald Affeldt and Nicolas Marti. Separation Logic in Coq. Online CVS: `http://savannah.nongnu.org/projects/seplog`. Bigtoe Web interface: `http://web.yl.is.s.u-tokyo.ac.jp/~nicolas/bigtoe`.

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

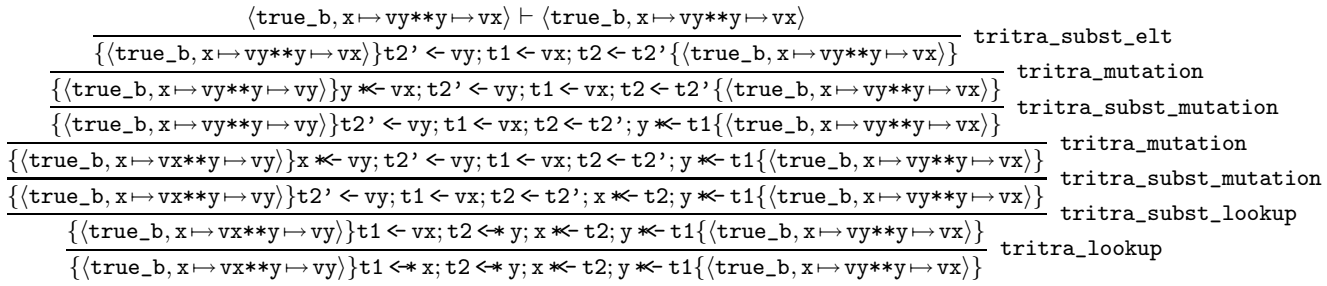


Figure 9: Swap of Cells using our Proof System (see Figures 10 and 11 for the same example using forward and backward reasoning)

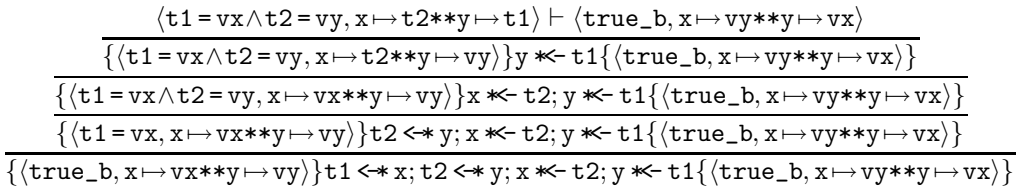


Figure 10: Swap of Cells using Forward Reasoning

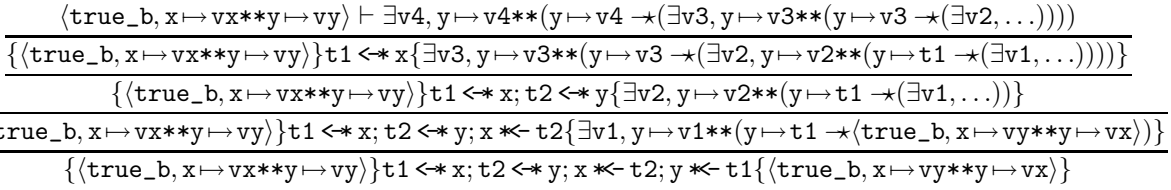


Figure 11: Swap of Cells using Backward Reasoning