Combining Coq and Automated Theorem Prover to Certify Separation Logic Assertions

Zhong Zhuang[1]

University of Science and Technology of China
dyzz@mail.ustc.edu.cn,
home page: http://kyhcs.ustc.edu.cn/~zz

Abstract. We present a method of combining Coq proof assistant and automated theorem prover to certify separation logic assertions and show how this method will reduce proof size and speed up the process of our certifying compiler. The automated separation logic prover is integrated with a linear arithmetic prover, proof search is led by rewriting rules to handle predicates on heap. This prover will produce a separately checkable Coq proof term in the end of any successful session, and we take advantage of the "proof by reflection" technique to translate a core fragment of separation logic proofs of the automated prover into compact Coq representations.

1 Introduction

Separation logic[28] is an extension of Hoare logic[18] which is modular and much convenient to verify properties with pointers and mutable data structures of imperative programming languages.

We are currently developing a certifying compiler,namely "CComp"([22]), which allows user to give specification with a fragment of separation logic and automatically produce low-level code and the whole proof of the program. Generating proof of certifying compiler is crucial in *proof-carrying code*(PCC[25]) framework. However, in our previous work, the biggest part of the proof is from our built-in automated separation logic prover and the time and proof size complexity of separation logic increase by $O(n^2)$ for worst cases according to user provided specification. This makes it rather un-scalable.

In this paper, we present a method to deal with this difficulty. The key idea is to change the way to produce some part of the proof with "proof by reflection" technique and combine with a modified automated prover to maintain the ability to deal with predicates on heap. This makes a compact representation of the original proof and reduce time complexity but is still as powerful as the automated prover.

The main contributions of this paper are:

- We develop a tactic for a targeted fragment of separation logic in Coq with "proof by reflection" technique which makes possible to translate some proofs of the automated prover in a much compact representation. With this tactic, we can reduce the complexity to $O(n)$.

– In order to benefit from automated prover in Coq, we modify the automated prover to combine it with the reflexive tactic and provide a practical tactic for separation logic in Coq's top level. With this combination, we can prove complicated separation logic formula in Coq with the help of automated prover.

The rest of this paper is organized as follows. Sec 2 briefly introduces the certifying compiler "CComp", Coq proof assistant and built-in prover in "CComp". Sec 3 presents the built-in prover and explain why we need to combine with Coq. Sec 4 describes our implementation of a reflexive tactic for separation logic in Coq. Sec 5 gives details of how to combine these two tools to provide a practical separation logic tactic. We will give results and compare to related work in the remain sections.

2 CComp, Coq and Automated Theorem Prover

2.1 CComp

CComp[22] is a certifying compiler for a subset of the C programming language with explicit memory allocation and deallocation. This compiler intends to deal with data structures such as singly-linked lists, doubly-linked lists and trees. And the safety policy is much stronger than type and memory safety. We have designed a program logic combining a constrained first-order logic and a fragment of separation logic for the source language. We use a verification-condition-based method, and the generated verification conditions (VC for short) are proved by the built-in automated theorem prover. The low-level verification framework follows Hoare-style verification methods. The x86 assembly code, its specification and proofs are generated automatically based on a variant of SCAP[16]. The low-level proofs are constructed using predefined Coq tactics and templates, and partly by reusing the source-level VC proofs.

This paper is motivated by improving performance of the built-in automated theorem prover in CComp.

2.2 Coq Proof Assistant

Coq[7] is a proof assistant based on *Calculus of Inductive Constructions* and combines high-order logic and richly-typed functional programming language. Coq provides interactive proof, decision procedures and tactic[15] language to build proofs. And it is possible to connect Coq with external theorem prover.

In this paper, we have formalized separation logic, developed a reflexive tactic a targeted separation logic and add a tactic to Coq's top level to prove VC from CComp.

2.3 Automated Prover

The built-in automated prover receives VC from CComp, and will produce Coq checkable proof objects if the VC is valid. One sub-prover is for linear integer

arithmetic based on decision procedure Simplex. Its capability is comparable to the Coq tactic omega, but the size of proof generated by our prover is much smaller. Another sub-prover is for the separation logic fragment which also produces Coq proof terms. However, the proof term generated by our prover is quite large[22]. In this paper, we use "proof by reflection" technique to represent some part of the proof term in a more compact form in Coq and combine the automated prover with Coq to build a practical tactic for separation logic.

3 Automated Prover for Separation Logic

3.1 Overview

We integrate into CComp a built-in automated theorem prover with proof-term output. The input formula is in the form of an implication $A_1 \Rightarrow A_2$ which is generated by VCGen. The prover is designed to produce proof-terms when formula is valid. The prover is made up of a linear arithmetic prover and a separation logic prover.

The basic formula consists of two parts: pure formula and spatial formula. The pure formula and spatial formula accepted by the prover are defined by the following syntax (Pred is the name of built-in predicate, such as lseg, tree, and etc.):

$$\begin{aligned}
\text{binop} &::= + \quad - \quad * \quad / \\
\text{comparison} &::= = \quad \neq \quad > \quad < \quad \geq \quad \leq \\
\text{expression(E)} &::= \text{variable} \mid \text{E } \textit{binop} \text{ E} \\
\text{pure-term(P)} &::= \text{E } \textit{comparison} \text{ E} \\
\text{spatial-term(S)} &::= \text{E} \mapsto \text{E} \mid \text{Pred(E,}\ldots\text{,E)} \\
\Pi &::= \textbf{true} \mid \text{P} \mid \Pi \wedge \Pi \\
\Sigma &::= \textbf{emp} \mid \text{S} \mid \Sigma * \Sigma
\end{aligned}$$

The assertion language used in our prover is quite low-level. For example, the heap pointed by a pointer will be partitioned to several separated sub-heaps. Each sub-heap is asserted by the pointer plus the offset. E.g.,

$$p \mapsto data : 0, next : NULL$$

will be presented as:

$$p + 0 \mapsto 0 * p + 4 \mapsto 0$$

3.2 Separation Logic Prover

The separation logic prover in our compiler aims to prove the formula in the form of $\exists x_1, x_2, \ldots, x_n, \Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$ and generate machine-check-able proof objects(currently, we choose plain Coq proof term). Π is the pure formula not involving objects in heap and Σ is the spatial formula indicating the allocated

heap. $x_1, x_2, \ldots, x_n$ are existential variables in Π and Σ (we will make them implicit in the following text).

Currently, we support built-in predicates in spatial formula including *list-seg, list, double-linked-list-seg, double-linked-list, tree*. *emp* indicates that the heap is empty. You can refer to [29] for more detail.

The prover is inspired by Smallfoot[6] and [12]. A formula is processed by prover in following step:

Simplify Send each pair of expressions to our pure prover and get all the equality relations. Then rewriting is performed based on these relations.
For example:

$$p + 1 + 1 \mapsto v \vdash p + 2 \mapsto v$$

$p + 1 + 1$ and $p + 2$ are equal, and we will rewrite this formula to

$$p + 2 \mapsto v \vdash p + 2 \mapsto v$$

Unfold Try to apply following rules until no rules can be applied. This step tries to unfold predefined predicates. Here, we only present rules about linked list.

Lseg-Right unfold Lseg from right

$$\frac{\Delta * lseg(E_0, E_1) \vdash \Delta'}{\Delta * lseg(E, E_0) \vdash \Delta' * lseg(E, E_1)}$$

LSeg-Left unfold Lseg from left

$$\frac{\Delta * lseg(E_1, E_0) \vdash \Delta'}{\Delta * lseg(E, E_0) \vdash \Delta' * E \mapsto v * E + 4 \mapsto E_1} \Delta \wedge \Delta' \vdash E \neq E_0$$

List unfold List

$$\frac{\Delta * lseg(E, nil) \vdash \Delta'}{\Delta * list(E) \vdash \Delta'} \Delta \wedge \Delta' \vdash E \neq nil$$

All the guardians here, e.g. $E \neq E_0$ in rule **LSeg-Left**, are determined by our pure prover.

Fold Do the opposite to Unfold rules, these rules try to fold predicates in formula.

Clean Up Remove predicates that are equal in premise and conclusion.

It is worth to note the principle of applying unfold/fold rules[27], cause it leads to infinite loop without caution. We apply unfold/fold rules

- unfold predicates in premise
- fold predicates in conclusion if no more rules can be applied

After recursively calling these steps, we will finally get a formula in these forms:

- $emp \vdash emp$ This is what we want, and we can conclude the formula can be proved.
- $F \vdash emp$ This is a special state when we are doing frame inference, F is the *frame* we need.
- otherwise, prover will give an error message to show the formula can not be proved.

To get machine check-able proof, we develop a separation logic library in Coq proof assistant which includes memory model, formalized heap, separation logic definition, pre-defined data structures and all the related lemmas, see details in Sec 4.2. Prover will record each step it takes during proof search, and we directly build a Coq-checkable proof term.

For example, we have a Coq lemma :

$$frame_rule : forall \quad F, P, Q, \quad P ==> Q \;\; -> \quad (P * F) ==> (Q * F)$$

and we recorded *frame F prem conc;*... during proof search , then the proof term will look like:

$$frame_rule \;\; F \;\; P \;\; Q \;\; _$$

P and Q can be inference from current premise and conclusion, _ is a proof hole and need to be filled with proof object generated by the remain records of prover.

3.3 Pure Prover

The pure prover serves as a key component for separation logic prover. It provides two features: Check implication between two given pure formulas and find implicit equalities of variables in them. The prover will give back both results and proof objects.

Check implication The input is in the form of $\Pi_1 \vdash \Pi_2$, where Π_i is the conjunction of some integral expressions. We prove it by contradiction, that is $\Pi_1 \wedge \neg\Pi_2 \vdash False$. Here the *Simplex*[17] algorithm is used to check the satisfiablity of $\Pi_1 \wedge \neg\Pi_2$. If $\Pi_1 \wedge \neg\Pi_2$ is not satisfiable, $\Pi_1 \wedge \neg\Pi_2 \vdash False$ is true.

The first step is to normalize all the expressions to the form *const* $<=$ *exp* according to the comparator. As we only concern integers in our memory model, we can avoid introducing δ in original Simplex and simply transform $x > n$ to $n + 1 \leq x$. And we can detect the failed case $n < x < n + 1$ in an early stage. The tricky part is how to deal with equality and in-equality.

$$Prem \wedge a = b \vdash Conc \Rightarrow Prem \wedge (a \leq b) \wedge (b \leq a) \vdash Conc$$

$$Prem \wedge a \neq b \vdash Conc \Rightarrow (Prem \wedge a < b \vdash Conc) \wedge (Prem \wedge b < a \vdash Conc)$$

After the normalize step, we can build initial context and use Simplex algorithm to check the satisfiablity of current context and adjust it. This step is recursively called until a satisfied model is found or no more adjustment can be done.

Find implicit equalities We collect all the expressions of comparator "=" and find implicit equalities:

$$\begin{cases} n \leq x \leq n & x = n \\ x = n + z_1 \wedge y = n + z_2 \wedge z_1 = z_2 & x = y \\ x = p_1 + q_1 \wedge y = p_2 + q_2 & x = y \\ \quad \wedge p_1 = q_1 \wedge p_2 = q_2 & \end{cases} \tag{1}$$

Currently, we do not allow predicates asserting on stack variables and their relations, thus we can bypass uninterpreted or interpreted functions here. So calculating congruence closure is quite straight forward.

Proof objects Proof objects generation in pure prover is not in the style of record-and-replay in separation prover. The reason is we find many redundant proof terms in the output object and make it significantly large. So we generate proof objects in pure prover as follow.

Within each step of our prover, we store proved sub-goals in a "proof library" and mark each sub-goal as a proof hole. When prover finishes, we get a bunch of proof objects and a proof tree of the original formula. The generation is to traverse this tree and fill all of these proof holes by finding proof objects in the "proof library".

3.4 Motivation of Combination

We have profiled with this automated prover, and noticed that a lot of proof of separation logic is charged in the form of :

$$\Sigma_1 * P * \Sigma_2 \ \vdash \ P * \Sigma_1 * \Sigma_2$$

This happens when we need to apply "unfold/fold" rules and do "clean up". For example, consider when we have P both in premise and conclusion.

$$\Sigma_1 * P * \Sigma_2 \ \vdash \ \Sigma_1' * P * \Sigma_2'$$

We can use "frame" rule:

$$\Sigma_1 \vdash \Sigma_2 \Rightarrow P * \Sigma_1 \vdash P * \Sigma_2$$

However, we need the following steps to apply this rule:

1. proof $\Sigma_1 * P * \Sigma_2 \ \vdash \ P * \Sigma_1 * \Sigma_2$
2. proof $P * \Sigma_1' * \Sigma_2' \ \vdash \ \Sigma_1' * P * \Sigma_2'$
3. apply "frame" rule, and leave proof obligation $\Sigma_1 * \Sigma_2 \ \vdash \ \Sigma_1' * \Sigma_2'$

The previous way to proof formulas as in step 1 and 2 here is use the following properties of separation logic:

$$\Sigma_1 * \Sigma_2 \ \vdash \ \Sigma_1 * \Sigma_2$$

$$(\Sigma_1 * \Sigma_2) * \Sigma_3 \ \vdash \ \Sigma_1 * (\Sigma_2 * \Sigma_3)$$

And this will lead to a $O(n^2)$ complexity according to the size of the specification. Think we need to apply rules many times, this is a great difficulty to scale our prover.

4 Reflexive Tactic for Separation Logic

4.1 Proof by reflection

Proof by reflection is a feature of proof assistant to embed a programming language inside the logical language[7]. Generally, this technology is to reduce time-consuming proof to computation. A visible example in Coq is that we often rely on term reductions: $\beta\delta\zeta$-reduction for simple functions and ι-reduction for recursive functions. However, reflection proofs become challenging when we are facing more complicated goals. There are mainly two schemes to set up reflection proofs[8].

Prover-based To obtain reflection proof based on a theorem prover requires the following steps:

1. encode the original logical proposition p into a symbolic expression F
2. write the evaluation function *Eval* of type $ENV \rightarrow F \rightarrow Prop$ which evaluate F under environment ENV back to the logical proposition. The environment contains information such as variable bindings.
3. write a prover $Prover$ of type $F \rightarrow bool$ and proof the correctness of the prover. With the correctness proof, we can conclude: $\forall\ p, Prover(p) = true \rightarrow p$.

With these steps we can build a reflexive proof in the following way. First, we construct the correspond environment ENV such as variable bindings. Then change the original proposition p by symbolic expressions F which satisfies $Eval\ F\ ENV = p$. We can conclude p holds if prover returns $true$.

Checker-based For NP-complete decision problems, prover-based reflection proof is impossible. However, checkers for these proofs often have polynomial complexity. We can build reflection proofs based on a certificate checker:

1. write a certificate generator $Cert$ for original proposition p
2. write a checker $Check$ of type $cert \rightarrow bool$ and prove the correctness of checker: $\exists cert, cert = Cert(p) \rightarrow Check(cert) = true \rightarrow p$

4.2 Certified prover for targeted separation logic

In this section, we will implement a reflexive prover for a fragment of separation logic with the prover-based scheme.

Targeted separation logic To reduce proof size and speed up the process of CComp, we have implemented a certified prover for a decidable fragment of separation logic[4] in Coq with which we can translate some part of the proofs to compact form in "proof by reflection" way. To support assertions from CComp, we extended it with linear arithmetic to pointers. As we handle predicates in the automated theorem prover, we use index to mark each predicate. The target fragment of separation logic is by the following grammar:

$$
\begin{aligned}
\text{index} &::= i \\
\text{binop} &::= + \;\; - \;\; * \;\; / \\
\text{variable} &::= \text{Var } i \\
\text{const} &::= \text{Z} \\
\text{expression(E)} &::= \text{variable} \mid \text{const} \mid \text{E } \mathit{binop} \text{ E} \\
\text{simple spatial formula(S)} &::= \text{E} \mapsto \text{E} \mid \text{E} \mapsto ? \mid \text{Pred } i \\
\text{spatial formula}(\Sigma) &::= \text{emp} \mid S * \Sigma
\end{aligned}
$$

index is used to distinguish different variables and predicates from the original proposition. E $\mapsto ?$ is equivalent to $\exists E', E \mapsto E'$.

Encoding of the targeted separation logic We encode the targeted separation logic in Coq as:

```
Inductive sigma:Set:=
  pto : expr → expr → sigma
| cell: expr → sigma
| semp: sigma
| star: sigma → sigma → sigma
| pred: index → sigma
.
```

pto $e1$ $e2$ and *cell* e present $E \mapsto E$ and $E \mapsto ?$. *emp* present the empty heap. *star* is the normal separating conjunction. *pred* i present the predicate which has key i in the predicate map. We will implement variable map and predicate map in **varmap** of "Quote" library in Coq, *index* corresponds to the type "index". For example, the following separation logic formula

$$list(x) * x \mapsto 1 * emp * y \mapsto ?$$

will be encoded as:

$$star\ (pred\ End_idx)\ (star\ (pto\ (var\ End_idx)\ (const\ 1))\ (star\ semp\ (cell\ (var\ (Left_idx\ End_idx)))))$$

Implementation of original separation logic We have implement separation logic much like [13]. Locations and values are of type Z while *heap* is of type $Z \to option\ Z$. Thus the separation logic proposition is:

```
Definition prop:= heap->Prop.
```

Basic separation logic propositions are *emp*, separating conjunction(use notation "$**$"), map to(use notation "$E_1|->E_2$" and "$E|->?$"). And we have already defined some predicates for linked-list, list segment, circularly linked list, double linked list, tree. For instance, the definition of list segment is:

```
Inductive lseg(head:ptr)(tail:ptr):heap->Prop:=
  | emp_lseg : head=tail → lseg head tail empty
  | head_lseg : ∀ h,head<>tail → (∃ v,exists next,
     ((head|->v)**((head+4)|->next)**(lseg next tail)) h) → lseg head tail h.
```

We think it extends to definitions of other predicates on heap. We use notation "$\sim\sim>$" for implication on separation logic where $P\ \sim\sim>\ Q$ means $\forall\ h : heap,\ P\ h \to\ Q\ h$.

Evaluation To evaluate the encoded separation logic proposition, we need environments which contain a map from index to variable($Vmap$) and index to predicate($Pmap$). As described in 4.1, the evaluate function is of type $sigma \to pmap \to vmap \to prop$.

```
Fixpoint eval sigma pmap vmap:=
  match sigma with
    | pto e1 e2 ⇒
       (eval_expr e1 vmap) |-> (eval_expr e2 vmap)
    | cell e ⇒
       fun h ⇒ ∃ v, (eval_expr e vmap |-> v) h
    | semp ⇒ emp
    | star s1 s2 ⇒
       (eval s1 pmap vmap) ** (eval s2 pmap vmap)
    | pred idx ⇒ get_pred vmap idx
  end.
```

Implement the decision procedure We implement the certified decision procedure in Coq is of type $sigma * sigma \to bool$, the pair of sigma corresponds to premise and conclusion.

First, we remove all the "semp"s from premise P and conclusion Q. Then, for each element in premise, remove equal element from conclusion. Only if we get "semp" after this step shall the decision procedure return *true*. As described in Sec 3, we use pure prover to find all the possible equalities and do substitution before we call this decision procedure, we don't rely on a certified decision procedure for linear arithmetic to check whether two pointers or values are equal.

Correctness proof Each step of the decision procedure is followed by a correctness proof to show that if we evaluate the targeted separation logic formula back to original formula, it holds before and after the according step in decision procedure. For example, we prove the correctness of remove all "semp"s.

```
Lemma remove_semp_correct: ∀ sigma pmap vmap, eval (remove_semp sigma)
pmap vmap ~~> eval sigma pmap vmap.
```

And finally, the correctness of the decision procedure:

```
Lemma sep_decision_correct: ∀ s1 s2 pmap vmap, sep_decision s1 s2 = true →
eval s1 pmap vmap~~> eval s2 pmap vmap.
```

4.3 Implement a reflexive tactic

The procedure we have done so far in Coq can be extracted[21] to an OCaml program, but we are interested in using this decision procedure directly as a tactic because we need the proof object.

We use Coq's metalanguage Ltac[15], which is possible to do pattern match on Coq terms, to build a representation in targeted separation logic of the given separation logic formula F along with variable(*vmap*) and predicate(*pmap*) bindings. We are able to apply the decision procedure after this process.

To be precise, we first use two functions: **get_vars** and **get_preds** which scan F and get a list of all the variables and predicates. Then we use function **list2map** which builds up *vmap* and *pmap*. After retrieve these environment, we can convert F to targeted separation logic.

```
Ltac denote_sep sep vmap pmap:=
  match sep with
```

```
  | ?P |->? ⇒ let e:=denote_aux P vmap in constr:(cell e)
  | ?P ** ?Q ⇒
    let e1:=denote_sep P vmap pmap in
      let e2:=denote_sep Q vmap pmap in
        constr:(star e1 e2)
  | emp ⇒ semp
  | ?P |-> ?Q ⇒
    let e1:=denote_aux P vmap in
      let e2:=denote_aux Q vmap in
        constr:(pto e1 e2)
  | ?P ⇒
    let idx:=getIndex P pmap in
      constr:(pred idx)
end.
```

At this point, we can use tactic **change**(P and Q are the representation in targeted separation logic):

```
change(eval P pmap vmap~~>eval Q pmap vmap).
```

This tactic asks Coq to compute the "eval" function and check that it is equal to the original formula. Now, we can wrap everything up with the correctness proof of the decision procedure in Sec4.2. Then the goal is changed to

$$\text{sep_decision P Q} = \text{true}$$

Coq starts to compute with the decision procedure, if it returns **true**, the goal will become "$true = true$" which is trivial, otherwise, Coq will complain.

5 Implementation Details

With the automated prover in Sec 3 and the reflexive tactic in Sec 4, we combine these tools to provide a tactic "**sep**" for separation logic in Coq's top level. Figure 1 shows process of the combination.

Starting from CComp, it generates VCs from annotated C program according to the specification. Then in step ①, CComp feeds a template proof script to Coq. At this point, Coq launches the automated prover with the **sep** tactic with steps ③,④,⑤ and ⑥. Before calling the automated prover, Coq has to translate goal to the input format of the prover. During step ②, we use some OCaml code to read current goal and produce a text file for automated prover as input. These code also calls automated prover in step ③. As describe in Sec 3 and Sec 4 , the automated prover will use the reflexive tactic to represent some part of the proofs. In steps ④ and ⑤, the automated prover will feeds Coq script using this tactic to get reflection proofs. After these steps, the automated prover will

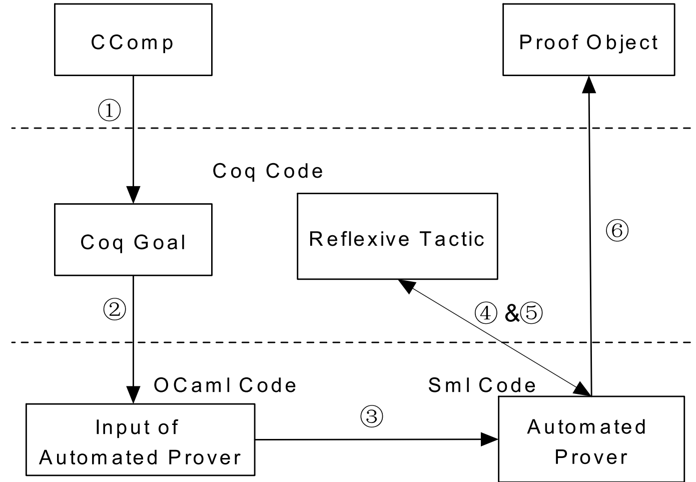

Fig. 1. Process of the Combination

produce proof objects if it is successfully called. In step ⑥, some OCaml code will load the output of automated prover and use it to finish the proof session.

The OCaml code also provide the “**sep**” tactic in Coq’s top level. The template proof script from CComp in step ① is in the form of:

Lemma CComp_VC_*num*: *VC_PRESENTATION_IN_COQ*.
Proof.
sep.
Qed.
Print CComp_VC_*num*.

The “**sep**” shall lead the remain steps. Steps ② and ③ are trivial, we just need to print the current goal as a suitable format and call automated prover in command line. Step ④ and ⑤ need to call Coq from the automated prover. However, we can not just feed back the proof script, because we are already in the proof session of a certain VC. So we need to start another Coq instance[9] and feed proof script using the reflexive tactic as in step ①. This will slow down the “**sep**” tactic. It can be fixed if we directly produce plain reflection proof term in automated prover.

6 Experiment

We

7 Related Work

It is common to describe memory model with separation logic, such as CompCert([19]), [10] and [16].

Along with these research, there are some tools aims at automated verification on separation logic. The typical work among them is SmallFoot([6]). The idea to use a SMT prover to automatically prove separation logic is not new. [10] used Z3[14] and a separation logic prover from jStar. [26] is an extension to author's previous work in [27], and it uses an SMT solver for checking validity of pure formula. The key difference of our work in CComp is that we produce a plain proof term which is crucial in PCC framework.

The technique of "proof by reflection" is not new [11]. In Coq, we already have built-in reflexive tactics such as **romega** which is based on proof tracks from Omega Test, **micromega**[8] which is based on certificate checker to prove linear arithmetic problems. Not only for arithmetic, Lescuyer has developed SAT solver[20] with this technique. The most related work is Affeldt's [23]. It also provides a certified decision procedure in Coq and use extraction[21] to get a certified verifier. We extend this work to abstract predicates in Coq which allow extensions of predicates in automated prover and wrap a reflexive tactic which is able to produce proof object.

Tactic for separation logic in Coq is also not new. Appel has a promotion work in [2]. Typical works includes McCreight's [24] and Chlipala's from Ynot[13]. [24]'s work provides many practical tactics but requires more tactics to finish proof. [13]'s work is mostly done with Coq and can prove separation logic formula with a single tactic which use tactics handling predicates as parameters. But user needs to write tactics for new data structure to be called by this tactic. And it need $O(n^2)$ complexity for the example we show in Sec 3.4. Our work moves this part of work to automated prover, and allows user to write extensions with more convenient language.

8 Conclusion and Future Work

In this paper, we present the combination of automated prover and Coq. This greatly eases the process of CComp. As shown with examples, the "**sep**" tactic reduces size complexity of proof and improves the scalability of the automated prover. We also show that the "proof by reflection" technique can be used for a complicated logic.

The performance of this tactic has a promising potential as we described in Sec 5 and we can build a reflexive tactic to reduce proof size of pure prover. Another drawback is the termination proof of our automated prover and we need to provide a convenient way to extend with new data structures.

References

1. Appel, A., Blazy, S.: Separation logic for small-step Cminor. Theorem Proving in Higher Order Logics pp. 5–21 (2007)

2. Appel, A.: Tactics for separation logic. Early draft on http://www.cs.princeton.edu/~appel/papers/septacs.pdf (2006)
3. Ayache, N., Filliâtre, J.: Combining the Coq proof assistant with first-order decision procedures. unpublished) (2006)
4. Berdine, J., Calcagno, C., O'Hearn, P.: A decidable fragment of separation logic. FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science pp. 110–117 (2005)
5. Berdine, J., Calcagno, C., O'Hearn, P.: Symbolic execution with separation logic. Programming Languages and Systems pp. 52–68 (2005)
6. Berdine, J., Calcagno, C., O'Hearn, P.: Smallfoot: Modular automatic assertion checking with separation logic. In: Formal Methods for Components and Objects. pp. 115–137. Springer (2006)
7. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer-Verlag New York Inc (2004)
8. Besson, F.: Fast reflexive arithmetic tactics the linear case and beyond. Types for Proofs and Programs pp. 48–62 (2007)
9. Boldo, S., Filliâtre, J., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs. Intelligent Computer Mathematics pp. 59–74 (2009)
10. Botincan, M., Parkinson, M., Schulte, W.: Separation logic verification of C programs with an SMT solver. Electronic Notes in Theoretical Computer Science 254, 5–23 (2009)
11. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Theoretical Aspects of Computer Software. pp. 515–529. Springer (1997)
12. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 289–300. ACM (2009)
13. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. ACM SIGPLAN Notices 44(9), 79–90 (2009)
14. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems pp. 337–340 (2008)
15. Delahaye, D.: A tactic language for the system Coq. In: Logic for Programming and Automated Reasoning. pp. 377–440. Springer (2000)
16. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. p. 414. ACM (2006)
17. Gomory, R.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society 64(5), 275–278 (1958)
18. Hoare, C.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
19. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
20. Lescuyer, S., Conchon, S.: A reflexive formalization of a SAT solver in Coq. In: Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (2008)
21. Letouzey, P.: Extraction in coq: An overview. Logic and Theory of Algorithms pp. 359–369 (2008)

22. Li, Z., Zhuang, Z., Chen, Y., Yang, S., Zhang, Z., Fan, D.: A Certifying Compiler for Clike Subset of C Language. In: accepted by 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (2010)
23. Marti, N., Affeldt, R.: A certified verifier for a fragment of separation logic. Information and Media Technologies 4(2), 304–316 (2009)
24. McCreight, A.: Practical tactics for separation logic. Theorem Proving in Higher Order Logics pp. 343–358 (2009)
25. Necula, G.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 106–119. ACM (1997)
26. Nguyen, H., Chin, W.: Enhancing program verification with lemmas. In: Computer Aided Verification. pp. 355–369. Springer (2008)
27. Nguyen, H., David, C., Qin, S., Chin, W.: Automated verification of shape and size properties via separation logic. In: Verification, Model Checking, and Abstract Interpretation. pp. 251–266. Springer (2007)
28. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science, 2002. Proceedings. pp. 55–74 (2002)
29. Reynolds, J.: An introduction to separation logic. In: Proceedings Marktoberdorf Summer School (2008)
30. Stump, A., Dill, D.: Generating proofs from a decision procedure. In: Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy. Citeseer (1999)