Separation logic adapted for proofs by rewriting

Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. We present a formalisation of separation logic which, by avoiding the use of existential quantifiers, allows proofs that only use standard equational rewriting methods as found in off-the-shelf theorem provers. This proof automation is sufficiently strong to free the user from dealing with low-level details in proofs of functional correctness. The work presented here has been implemented in HOL4 and ACL2. It is illustrated on a standard example (reversal of a linked-list).

1 Introduction

Separation logic [7] has emerged as an effective technique for proving the correctness of pointer manipulating programs. As a result, there have been a number of theorem prover formalisations of separation logic [1, 4, 9, 11] and tactics for dealing with separation logic-style reasoning in theorem provers [2, 5, 10].

In this paper we present a novel formalisation which, by avoiding the use of existential quantifiers, allows basic rewriting to suffice as a useful proof tactic. We believe that the simplicity of our setup makes it independent of any particular theorem prover: the work presented here has been implemented in HOL4 [8] (by the current author) and subsequently in ACL2 [3] (by Matt Kaufmann).

Gast [2] also modifies separation logic to fit better with current theorem provers. Compared with Gast [2], our approach follows the idea of separation logic more closely (we state memory content and memory layout together) and achieves a higher degree of automation by simple rewriting (Sections 3 and 4).

2 Separation logic without quantifiers

At the heart of separation logic lies the separating conjunction $*$. The separating conjunction is defined such that $p * q$ holds for state s whenever state s can be split into two disjoint parts (according to some definition of disjoint union $\uplus$) such that p holds for one part and q for the other part.

$$(p * q)\ s \quad = \quad \exists s_1\ s_2.\ (s = s_1 \uplus s_2) \wedge p\ s_1 \wedge q\ s_2$$

The value of the separating conjunction becomes apparent in the context of (avoiding unwanted) pointer aliasing: if $\mathsf{list}\ a\ xs$ asserts that a linked list is in memory then $\mathsf{list}\ a\ xs * \mathsf{list}\ b\ ys$ states that there are two list in memory and these occupy disjoint addresses in memory. Thus asserting that there is no

pointer aliasing between the lists. This list assertion is conventionally defined as follows. Here the notation $a \mapsto x$ means that memory location a holds value x.

$$\begin{array}{lcl} \mathsf{list}\ a\ [] & = & (a = 0) \\ \mathsf{list}\ a\ (x{::}xs) & = & \exists a'.\ (a \mapsto a') * (a{+}1 \mapsto x) * \mathsf{list}\ a'\ xs * (a \neq 0) \end{array}$$

In this paper, we will show that the common case of linked-lists and other assertions of the form $(a \mapsto x) * (b \mapsto y) * \ldots * (c \mapsto z)$ can be formalised without quantifiers in such a way that simple rewriting is a sufficiently powerful proof tool for proofs of functional correctness.

Instead of defining a separating conjunction directly, we define a function separate which mimics $(a \mapsto x) * (b \mapsto y) * \ldots * (c \mapsto z)$ when supplied with a list of the form $[(a, x), (b, y), \ldots, (c, z)]$ and here the separation is due to all_distinct xs, which states that there are no duplicate elements in its argument list xs. Below t is a list of addresses that must be distinct from those mentioned in l.

$$\begin{array}{lcl} \mathsf{separate}\ []\ t\ state & = & \mathsf{all\_distinct}\ t \\ \mathsf{separate}\ ((a, x){::}l)\ t\ state & = & (state(a) = x) \wedge \mathsf{separate}\ l\ (a{::}t)\ state \end{array}$$

A suitable adaption of list is then defined as a function listx which produces a list of (address,value) pairs which can be substituted for the variable l above. To avoid the existential quantifier used in the definition of list, we make the internal addresses external and explicit. Here addr xs returns a pointer to the head of the list xs.

$$\begin{array}{lcl} \mathsf{addr}\ [] & = & 0 \\ \mathsf{addr}\ ((a, x){::}xs) & = & a \\ \mathsf{listx}\ [] & = & [] \\ \mathsf{listx}\ ((a, x){::}xs) & = & (a, \mathsf{addr}\ xs){::}(a{+}1, x){::}\mathsf{listx}\ xs \end{array}$$

Separation logic states the correctness of programs as Hoare triple judgments $\{pre\}\, code\, \{post\}$. We define a similar judgement as follows based on an operational semantics exec (defined in [6]) for a toy machine language where the code resides in memory. This judgement spec is defined to assert that, if the initial state s satisfies pre_1 and contains the code $code$ separately from addresses pre_2, then n steps of execution will reach a state which satisfies $post_1$ and contains $code$ separately from addresses $post_2$. We write list append as $+\!+$.

$$\begin{array}{l} \mathsf{spec}\ s\ n\ (pre_1, pre_2)\ code\ (post_1, post_2)\ = \\ \quad \mathsf{separate}\ (code +\!+ pre_1)\ pre_2\ s \implies \\ \quad \mathsf{separate}\ (code +\!+ post_1)\ post_2\ (\mathsf{exec}\ n\ s) \end{array}$$

3 Automatic rewriting tactic

The above definitions allow rewriting alone to suffice for proving specifications. Our rewriting tactic essentially just expands all the definitions and rewrites where applicable with lemmas that describe append ($+\!+$) i.e. the expansion of separate $(xs +\!+ ys)\ t\ s$ and all_distinct $(xs +\!+ ys)$.

This simple rewriting tactic is capable of proving specifications for single passes though code. For example, it can automatically prove the following specification for a sequence of four instructions which swap the next-pointers in two linked lists, thus swapping the tails of the lists (xs is swapped for ys). The program counter which is incremented by 12 is stored at address 0. This specification states that addresses 3 and 4 are used as temporaries during execution. Their initial and final values are not recorded. Let $\mathsf{llist}\ p\ xs = (p, \mathsf{addr}\ xs)::\mathsf{listx}\ xs$.

$\mathsf{spec}\ s\ 4$
$([(0,p)] +\!\!+ \mathsf{llist}\ 1\ (x::xs) +\!\!+ \mathsf{llist}\ 2\ (y::ys) +\!\!+ frame, [3,4] +\!\!+ rest)$
$(\mathsf{pointer\_swap\_code}\ p)$
$([(0,p{+}12)] +\!\!+ \mathsf{llist}\ 1\ (x::ys) +\!\!+ \mathsf{llist}\ 2\ (y::xs) +\!\!+ frame, [3,4] +\!\!+ rest)$

The reason for why rewriting alone can prove this is very simple: the expansion of preconditions produces a number of inequalities, e.g. $p \neq q$, on the left-hand-side of the implication in spec. These inequalities resolve if-statements,

$$\mathsf{if}\ p = q\ \mathsf{then}\ x\ \mathsf{else}\ state(p)$$

that arise in the expansion of postconditions, i.e. the right hand-side of the implication in the definition of spec.

4 Verification example

Finally, we will demonstrate the use of our rewriting tactic as part of a standard example: verification of in-place list reversal.

The code we will verify expects on entry, that location 1 holds a pointer to the linked-list which is to be reversed. On each iteration of the loop (around location `18`), one element of the list is popped from the list pointed to from location 1 and prepended to a list whose pointer is kept in location 2. On exit, location 2 holds a pointer to the reversed list. Our toy machine language has no registers, thus locations 1, 2 and 3 are used here as if they were registers.

```
 0 :  mem[2] := 0
 3 :  jump to 18
 6 :  mem[3] := mem[mem[1]]
 9 :  mem[mem[1]] := mem[2]
12 :  mem[2] := mem[1]
15 :  mem[1] := mem[3]
18 :  jump to 6, if mem[1] ≠ 0
```

We first prove a lemma about the behaviour of the body of the loop. The loop body transfers an element from one of the linked lists to the other, looping around program location 18 in the code rev_code, which is positioned relative to address p. This goal is automatically discharged by our rewriting tactic.

$\mathsf{spec}\ s\ 5$
$([(0,p{+}18)] +\!\!+ \mathsf{llist}\ 1\ (x::xs) +\!\!+ \mathsf{llist}\ 2\ ys +\!\!+ frame, [3] +\!\!+ rest)$
$(\mathsf{rev\_code}\ p)$
$([(0,p{+}18)] +\!\!+ \mathsf{llist}\ 1\ xs +\!\!+ \mathsf{llist}\ 2\ (x::ys) +\!\!+ frame, [3] +\!\!+ rest)$

The proof of the main loop is a simple induction on the length of the list pointed to from location 1, i.e. xs. The base case is solved by our rewrite tactic; the step case is a simple 4-line proof which composes the above lemma for the body of the loop with the inductive hypothesis.

$$\begin{array}{l}\mathsf{spec}\ s\ (1 + \mathsf{length}\ xs \times 5)\\ \quad([(0, p{+}18)] +\!\!+ \mathsf{llist}\ 1\ xs +\!\!+ \mathsf{llist}\ 2\ ys +\!\!+ frame, [3] +\!\!+ rest)\\ \quad(\mathsf{rev\_code}\ p)\\ \quad([(0, p{+}21)] +\!\!+ \mathsf{llist}\ 2\ (\mathsf{reverse}\ xs +\!\!+ ys) +\!\!+ frame, [1, 3] +\!\!+ rest)\end{array}$$

By joining the above specification with a similar lemma for the initialisation code, we arrive at the final specification which states that list xs is reversed:

$$\begin{array}{l}\mathsf{spec}\ s\ (3 + \mathsf{length}\ xs \times 5)\\ \quad([(0, p)] +\!\!+ \mathsf{llist}\ 1\ xs +\!\!+ frame, [2, 3] +\!\!+ rest)\\ \quad(\mathsf{rev\_code}\ p)\\ \quad([(0, p{+}21)] +\!\!+ \mathsf{llist}\ 2\ (\mathsf{reverse}\ xs) +\!\!+ frame, [1, 3] +\!\!+ rest)\end{array}$$

Acknowledgements. I would like to thank Matt Kaufmann for writing and explaining how this work can be reproduced in the ACL2 theorem prover. His ACL2 script, which includes informative comments, is available on-line [6]. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

1. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2007.
2. Holger Gast. Lightweight separation. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 199–214. Springer, 2008.
3. Matt Kaufmann and J. Strother Moore. An ACL2 tutorial. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 17–21. Springer, 2008.
4. N. Marti, R. Aeldt, and A. Yonezawa. Towards formal verication of memory properties using separation logic. In *Workshop of the Japan Society for Software Science and Technology*. Japan Society for Software Science and Technology, Japan, 2005.
5. Andrew McCreight. Practical tactics for separation logic. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 343–358. Springer, 2009.
6. Magnus O. Myreen and Matt Kaufmann. HOL4 and ACL2 implementations, HOL4 (Myreen): `http://www.cl.cam.ac.uk/~mom22/sep-rewrite/` ACL2 (Kaufmann): `http://acl2-books.googlecode.com/`.
7. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.
8. Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 28–32. Springer, 2008.
9. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Principles of Programming Languages (POPL)*, pages 97–108. ACM, 2007.
10. Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 469–484. Springer, 2009.
11. Tjark Weber. Towards mechanized program verification with separation logic. In *Computer Science Logic (CSL)*, LNCS, pages 250–264. Springer, 2004.