Towards Automatic Assertion Refinement for Separation Logic

Andrew Ireland
School of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.ireland@hw.ac.uk

Abstract

Separation logic holds the promise of supporting scalable formal reasoning for pointer programs. Here we consider proof automation for separation logic. In particular we propose an approach to automating partial correctness proofs for recursive procedures. Our proposal is based upon proof planning and proof patching via assertion refinement.

1. Introduction

While pointers are a powerful and widely used programming mechanism, they are the source of many subtle program defects. As a consequence, developing and maintaining correct pointer programs is notoriously hard. This is reflected in some approaches to software development where the use of pointers is prohibited [1].

It would therefore be highly desirable to be able to automatically reason about the correctness of pointer programs. The stumbling block to achieving such a goal has been the lack of scalable reasoning techniques. Separation logic [21, 24] was designed to support pointer program proof by allowing specifications and reasoning to focus on only those parts of the heap that can be manipulated by a program. Through such *local reasoning*, separation logic holds the promise of scalable formal reasoning for pointer programs.

In this paper we exploit the local reasoning provided by separation logic. We propose an approach to automating the search for proofs within the context of separation logic. In particular, we focus upon partial correctness proofs for recursively defined procedures. While separation logic supports local reasoning, it does not remove the burden of the programmer having to supply intermediate assertions. Building upon existing automated reasoning techniques, our proposal aims to directly address this burden.

The contributions of this paper are twofold. Firstly we outline a proposal as to how proof automation could be achieved for a significant class of programs. Our proposal exploits in particular the automated reasoning technique known as rippling. Our second contribution is a proposal to extend rippling to deal with pointer references.

The paper is structured as follows. Background material on separation logic and the problem we are addressing are summarized in §2. In §3 our general approach is outlined, while future and related work is highlighted in §4. Our conclusions are presented in §5. For a more detailed presentation of our proposal see [14].

2. Separation Logic

Here we provide a brief overview of separation logic, for a detailed presentation see [21, 24]. In separation logic, program specifications are presented as Hoare triples, *i.e.* $\{P\}C\{Q\}$, where P and Q denote propositions, and C denotes program code. As well as predicate calculus, separation logic includes two new logical connectives. Firstly, *separating conjunction*, *i.e.* $P * Q$ holds for a heap if the heap can be divided into two disjoint heaps H_1 and H_2, where P holds in H_1 and Q holds in H_2 simultaneously. Secondly, *separating implication*, *i.e.* $P \mathrel{-\!\!*} Q$ holds if the current heap H_1 is extended with a disjoint heap H_2 in which P holds, then Q will hold in the extended heap.

At the level of heap cells, a singleton heap is denoted by the predicate $X \mapsto E$, *i.e.* one cell at address X with contents E. The empty heap is denoted by *emp*. Note that using separating conjunction, an assertion about a singleton heap can be extended to an arbitrary sized heap, *i.e.* $true*(X \mapsto E)$. A useful definition which we will exploit is:

$$(X \mapsto E_1, E_2) \quad \leftrightarrow \quad (X \mapsto E_1) * (X + 1 \mapsto E_2)$$

This gives a convenient way of expressing properties about an adjacent pair of heap cells.

The central proof rule of separation logic is the *frame rule*:

$$\frac{\{P\}\ C\ \{Q\}}{\{R * P\}\ C\ \{R * Q\}}$$

Note that the frame rule imposes a side condition, *i.e.* no variable occurring free in R is modified by C. It is the frame rule that supports local reasoning. That is, in proving the correctness of C the frame rule allows us to only focus on the variables and parts of the heap that are relevant to C, *i.e.* the "footprint" of C. The frame rule plays a pivotal role in reasoning about recursively defined procedures. Consider a procedure definition of the form:

$$\texttt{procedure}\ \ h(x_1, \ldots, x_m; y_1, \ldots, y_n)\ \texttt{is}\ C$$

where C denotes the procedure body. Note that $x_1, \ldots, x_m$ denote variables that are not modified by C while $y_1, \ldots, y_n$ denote variables that are modified by C. In terms of verification, we are interested in proving Hoare triples of the form:

$$\{P\}\ h(x_1, \ldots, x_m; y_1, \ldots, y_n)\ \{Q\} \tag{1}$$

We therefore require the application of a standard proof rule for reasoning about recursively defined procedures [12]:

$$\frac{\begin{array}{c}\{P\}\ h(x_1, \ldots, x_m; y_1, \ldots, y_n)\ \{Q\} \\ \vdots \\ \{P\}\ C\ \{Q\}\end{array}}{\{P\}\ h(x_1, \ldots, x_m; y_1, \ldots, y_n)\ \{Q\}}$$

This rule reduces the task of proving (1), to proving that the procedure body C satisfies the specification, under the assumption that any recursive calls also satisfy the specification. In order to apply this assumption, known as the *recursive hypothesis*, we require the following *substitution rule*:

$$\frac{\{P\}\ C\ \{Q\}}{(\{P\}\ C\ \{Q\})[e_1/v_1, \ldots, e_n/v_n]}$$

Note that $v_1, \ldots, v_n$ denote variables occurring free in P, C or Q, and if v_i is modified by C then e_i is a variable that does not occur free in any other e_j.

We now sketch the verification of a recursive procedure using separation logic. Our aim is to illustrate the role of the frame rule and highlight the search control problem that we are addressing. Consider the following `copylist` procedure[1]:

```
procedure copylist(i; j) is
   if i = nil then
      j := i
   else
      newvar i_h, i_t, j_t in
         i_h := [i];
         i_t := [i + 1];
         copylist(i_t; j_t);
         j := cons(i_h, j_t)
   end if
end copylist
```

Note that `copylist` has two pointer arguments, where the first argument points to an acyclic list. The effect of the procedure is to assign to the second argument a copy of the list referenced by first argument. To specify partial correctness of `copylist`, we introduce *list*, an inductively defined predicate which relates the notion of an acyclic singly-linked list to the abstract notion of sequences:

$$\begin{array}{rcl} list([], Z) & \leftrightarrow & emp \wedge (Z = nil) \\ list([X|Y], Z) & \leftrightarrow & (\exists p.\ (Z \mapsto X, p) * list(Y, p)) \end{array}$$

Note that the first argument of *list* denotes a sequence, where sequences are represented using the Prolog list notation. The second argument references the head of the corresponding linked-list structure. Armed with this definition, partial correctness of `copylist` can be specified as follows:

$$\{list(\alpha, i)\}\ \texttt{copylist}(i; j)\ \{list(\alpha, i) * list(\alpha, j)\} \tag{2}$$

[1] $[E]$ denotes the contents addressed by E.

In words, this states: Assuming i points to an acyclic list with contents α, then if the execution of `copylist`$(i; j)$ terminates then i and j will point to distinct acyclic lists, both containing α. Using the procedure proof rule (see above), the verification of (2) reduces to verifying the procedure body given the recursive hypothesis, *i.e.*

$$\{list(\alpha, i)\}\ \texttt{copylist}(i; j)\ \{list(\alpha, i) * list(\alpha, j)\} \tag{3}$$

In terms of verifying the recursive call, we have to prove:

$$\begin{array}{r} \{(\exists \alpha_h, \alpha_t.\ ([\alpha_h|\alpha_t] = \alpha) \wedge (i_h = \alpha_h) \wedge (i \mapsto i_h, i_t) * \\ list(\alpha_t, i_t))\} \\ \texttt{copylist}(i_t; j_t); \\ \{(\exists \alpha_h, \alpha_t.\ ([\alpha_h|\alpha_t] = \alpha) \wedge (i_h = \alpha_h) \wedge (i \mapsto i_h, i_t) * \\ list(\alpha_t, i_t) * list(\alpha_t, j_t))\} \end{array} \tag{4}$$

From (3), using the substitution rule, a precise specification of the recursive call can be inferred. However, such a specification is not strong enough to prove (4), as it does not specify the larger context of the procedure body. This is where the frame rule is required. In addition to the substitution rule, the frame rule, and the auxiliary variable elimination rule[2] enables us to bridge the gap between (3) and (4). Note that the application of the frame rule involves instantiating R to be:

$$([\alpha_h|\alpha_t] = \alpha) \wedge (i_h = \alpha_h) \wedge (i \mapsto i_h, i_t) \tag{5}$$

This process of instantiation corresponds to a programmer supplying an intermediate assertion. Below we provide an outline of our approach to automating the discovery of such intermediate assertions.

3. Overview of our General Approach

The task of verifying a procedure body can be represented schematically as follows:

$$\begin{array}{l} \{P\} \\ \quad \vdots \\ \{R_1 * P_1\} \\ \quad h(x_1^1, \ldots, x_p^1; y_1^1, \ldots, y_q^1); \\ \{R_1 * Q_1\} \\ \quad \vdots \\ \{R_n * P_n\} \\ \quad h(x_1^n, \ldots, x_p^n; y_1^n, \ldots, y_q^n); \\ \{R_n * Q_n\} \\ \quad \vdots \\ \{Q\} \end{array}$$

Achieving verification automation requires proof automation techniques as well as techniques for discovering appropriate frame rule instantiations, as highlighted above. We propose an iterative approach which integrates proof search with the process of instantiation. Central to our proposal is an automated reasoning paradigm called *proof planning* [6]. Proof planning automates the search for proofs through the use of explicit high-level proof outlines, known as *proof plans*. By making proof search knowledge explicit, proof planning supports proof patching via proof-failure analysis [13].

[2] The *auxiliary variable elimination rule* is used to introduce the existential variables α_h and α_t.

Applications of proof patching include conjecture generalization and lemma discovery [13, 15], loop invariant discovery [10, 11] and refining faulty conjectures [19]. In particular, the work on refining faulty conjectures is closely related to the refinement task proposed here. Here we see proof-failure analysis guiding the refinement of the frame rule instantiation. Below we first outline the proof automation aspect of the approach and then the proof-failure analysis.

3.1 Proof Automation

The schema presented above gives rise to two basic patterns of verification tasks:

Verifying recursive calls: to prove the i^{th} recursive call we are given a recursive hypothesis of the form:

$$\{P\}\ h(x_1, \ldots, x_p; y_1, \ldots, y_q)\ \{Q\}$$

and a goal of the form:

$$\{R_i * P_i\}\ h(x_1^i, \ldots, x_p^i; y_1^i, \ldots, y_q^i)\ \{R_i * Q_i\}$$

Working backwards from the goal, the frame rule and the substitution rule provide the basis for a proof plan.

Verifying intermediate assertions: to prove the i^{th} assertion we have to discharge a conjecture of the form:

$$(R_{i-1} * P_{i-1}) \rightarrow (R_i' * Q_i')$$

where R_i' and Q_i' are calculated from R_i and Q_i using weakest precondition semantics.

In terms of proof search, it is the verification of the intermediate assertions that requires guidance. We propose to use the *rippling* proof plan [2, 7, 8]. Rippling is a rewrite strategy that uses a difference reduction criterion to select applicable rewrite rules. Typically rippling is used to selectively rewrite a goal formula so that parts of the goal match with given hypotheses. Rippling was designed for reasoning about recursive structures, so it needs to be extended to deal with pointer programs. That is, a new annotation is required for highlighting differences at the level of pointer references. We use $\overset{Y\mapsto}{\boxed{\mathrm{X}}}$ to indicate that an occurrence of pointer reference X in a goal formula corresponds to an occurrence of pointer reference Y within the given hypothesis. Because of the nature of separation logic, proofs typically involve existential quantification. Our new pointer reference annotation can be extended to cover existential variables, more details can be found in [14].

3.2 Failure Driven Assertion Refinement

Now consider the problem of instantiating the frame rule. We start by approximating the frame rule instantiation. In terms of the schema given above, this involves instantiating R_i $(1 \leq i \leq n)$ to be $true$. In the case of the `copylist` example, this gives rise to the following approximation to (4), the specification of the recursive call:

$$\{true * list(S, i_t)\}$$
$$\texttt{copylist}(i_t; j_t);$$
$$\{true * list(S, i_t) * list(S, j_t)\}$$

Note that S denotes a meta-variable, a place holder for missing term structure. The initial approximation is then incrementally refined through an iterative process of proof planning and proof-failure analysis. This iterative process focuses on the verification of the intermediate assertions. In particular, we focus on post-recursive calls, as these verification tasks are most suitable to our proof-failure analysis techniques. In the case of `copylist`, the verification of the post-recursive call gives rise to a hypothesis of the form

$$true * list(S, i_t) * list(S, j_t) \tag{6}$$

while the goal is to prove

$$(\forall j'.\ (j' \mapsto i_h, j_t) \mathrel{-\!\!*} (list(\alpha, i) * list(\alpha, j'))) \tag{7}$$

In principle, rippling is applicable to this kind of verification task, modulo the extension proposed in §3.1. The expectation is that proof planning and proof-failure analysis within the context of rippling will provide sufficient constraints in order to discover an appropriate instantiation of the frame rule. In the specific case of (7), four iterations of proof planning are required in order to complete the proof. Note that (5), the required instantiation for R, is generated as a side-effect of the proof-planning and the associated proof-failure analysis. In addition, S is instantiated to be α_t within (6). Details of the actual proof planning and proof-failure analysis can be found in [14].

4. Future and Related Work

Our proposed approach to automating assertion refinement has been tested by-hand on `copylist`, and the `copytree` example given in [21, 24]. The `copytree` example deals with binary trees, giving rise to a procedure body with two recursive calls. In addition, the extension to rippling proposed in §3.1, has been tested, again by-hand, on the verification of an in-place list reversal program given in [24]. More challenging examples need to be explored, such as programs that manipulate graphs and directed acyclic graphs, which have been investigated on paper within the context of separation logic [5].

We are interested in partial correctness, where specifications express both the shape and contents of data structures. Our goal is to increase the level of proof automation that is possible when reasoning about such specifications, and as a consequence our approach is heuristic based. By restricting the kinds of assertions that one can verify, algorithmic approaches can be developed. For instance, SMALLFOOT [3] is an experimental tool that supports the automatic verification of shape properties specified in separation logic. SMALLFOOT uses a form of symbolic evaluation [4], where loop invariants are required, but instances of the frame rule are generated automatically. Interestingly, the method for generating instances of the frame rule is based upon a form of proof-failure analysis. Another interesting connection is that SMALLFOOT sometimes requires inductive lemmas. The proof plan for mathematical induction [7, 8] could potentially satisfy this requirement dynamically. More broadly, proof planning might provide a natural way of extending the power of SMALLFOOT to deal with partial correctness.

In [17] an algorithm for inferring loop invariants within the context of separation logic is described. Invariant properties related to the shape of the heap, as well as data, can be generated automatically. Like SMALLFOOT, the algorithm is based upon symbolic evaluation, and involves the repeated evaluation of the loop body in order to identify a fixed point. The search for a fixed point may not converge, as a consequence the algorithm is not complete.

Another related tool is the Pointer Assertion Logic Engine (PALE) [18] in which pointer programs and specifications are encoded in monadic second-order logic. Verification is achieved via the MONA tool [16]. While PALE can express relatively complex specifications, the user is required to provide loop invariants. Closely related to the goals of PALE is the work on parametric shape analysis as implemented in the TVLA tool [25]. TVLA provides significant verification automation. While not requiring loop invariants, TVLA may require a user to supply *instrumentation predicates*, *i.e.* predicates that encode *local* properties of datatypes. Instrumentation predicates, however, may be reused between applications.

Finally, in order to advance our work a mechanization of separation logic is required, preferably within a tactic based proof development environment. In [23], Preoteasa provides a formalization within PVS [22] which supports the verification of recursive procedures. A formalization which includes simple while loops, but not recursive procedures, has been undertaken by Weber [26]. Weber's mechanization was developed within Isabelle/HOL [20]. Building upon Isabelle/HOL would have significant advantages since we could also exploit IsaPlanner [9], an Isabelle based proof planning system.

5. Conclusion

Separation logic promotes scalable reasoning for pointer programs. Here we have focused upon the use of separation logic in specifying and reasoning about the partial correctness of recursive procedures. We propose an approach to automating such reasoning, based upon the iterative refinement of intermediate program assertions. The proposal builds upon existing automated reasoning techniques, *i.e.* proof planning and proof-failure analysis. We exploit in particular, the rippling search control technique. Our work has suggested a principled extension to rippling that is necessary if it is to be applied to reasoning about pointer programs. Initial investigations of the proposal have been positive, the time is ripe to prototype and further test the approach.

Acknowledgements: The research reported in this paper is supported by EPSRC grant GR/S01771. Thanks go to Alan Bundy, Bill Ellis, Jamie Gabbay, Paul Jackson, Alberto Momigliano, Alan Smaill, and members of the Dependable Systems and Mathematical Reasoning Groups for their feedback on this work.

6. References

[1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[2] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. Modular automatic assertion checking with separation logic. Draft, Department of Computer Science, Queen Mary, University of London, 2005.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[5] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation, and aliasing. In *Proceedings of the Second workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*, Venice, Italy, 2004.

[6] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.

[7] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[8] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[9] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[10] B.J. Ellis and A. Ireland. Automation for exception freedom proofs. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering*, pages 343–346. IEEE Computer Society, 2003.

[11] B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of 4th International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 67–86. Springer Verlag, 2004.

[12] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, 1971.

[13] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992.

[14] A. Ireland. Towards automatic assertion refinement for separation logic. Technical Report HW-MACS-TR-0039, School of Mathematical and Computer Sciences, Heriot-Watt University, 2006.

[15] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[16] N. Klarlund and A. Moller. MONA version 1.4 user manual. BRICS Notes Series NS-01-1, Dept. of Computer Science, University of Aarhus, 2001.

[17] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Proceedings of the Third Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'06)*, pages 47–60, Charleston, SC, 2006.

[18] A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of PLDI'01*, pages 221–231, 2001.

[19] R. Monroy, A. Bundy, and A. Ireland. Proof Plans for the Correction of False Conjectures. In F. Pfenning, editor, *5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, Lecture Notes in Artificial Intelligence, v. 822, pages 54–68, Kiev, Ukraine, 1994. Springer-Verlag.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[21] P. O'Hearn, J. Reynolds, and Y. Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001.

[22] S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In D. Kapur, editor, *Proceedings of CADE-11*. Springer Verlag, 1992. LNAI vol. 607.

[23] V. Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. TUCS Technical Report 753, Turku Centre for Computer Science, 2006.

[24] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[26] T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.