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 [18, 19] was designed to support reasoning about pointer programs 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 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 the burden of intermediate assertions.

The contributions of this paper are two fold. Firstly we provide a detailed 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 our automated reasoning techniques is provided in §2 and §3 respectively. In §4 we present our general approach, while its application is illustrated in detail in §5. Future and related work is outlined in §6, while conclusions are presented in §7.

2 Separation Logic

Here we provide a very brief overview of separation logic, for a detailed presentation see [18, 19]. Within separation logic, program specifications are presented as Hoare triples, i.e. \( \{ P \} C \{ Q \} \) which means: If program \( C \) is executed within a state that satisfies \( P \), and if \( C \) terminates, then the final state will satisfy \( Q \). Here a state denotes the program’s store and heap. Assertions such as \( P \) and \( Q \) are expressed in separation logic. Separation logic extends predicate calculus with the following forms of assertion:

- empty heap: the assertion \( \text{emp} \) holds for a heap that contains no cells.
- singleton heap: the assertion \( X \mapsto E \) holds for a heap that contains a single cell, i.e. one cell at address \( X \) with contents \( E \).
- separating conjunction: the assertion \( P \ast Q \) 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.
- separating implication: the assertion \( P \rightarrow Q \) 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.
Note that while an assertion of the form \( X \mapsto E \) specifies a property of a singleton heap, separating conjunction allows us to extend the assertion to an arbitrary sized heap, i.e. true \(*\ (X \mapsto E)\). This gives rise to a useful definition of the form:

\[
(X \mapsto E) \iff (\text{true} \,*\ (X \mapsto E))
\]

Another useful definition which we will exploit is:

\[
(X \mapsto E_1, E_2) \iff (X \mapsto E_1) \,*\ (X + 1 \mapsto E_2)
\]

This gives a convenient way of expressing properties about an adjacent pair of heap cells. To illustrate, \([E] \) contains the values 4 and 7 respectively. In terms of dereferencing, we use \([E] \) to denote the contents addressed by \(E\). Given \((x \mapsto 4, 7)\), then \(x\) evaluates to 4 while \([x+1]\) evaluates to 7.

We now consider the axioms and proof rules which are directly relevant to this paper. Firstly, heap allocation, and in particular the allocation of two adjacent heap cells is defined by the following axiom:

\[
\{(\forall x'. (x' \mapsto E_1, E_2) \Rightarrow P[x'/X])\} \; X := \text{cons}(E_1, E_2) \; \{P\}
\]

A corresponding lookup axiom is defined as follows:

\[
\{(\exists x'. (E \mapsto x') \& P[x'/X])\} \; X := [E] \; \{P\}
\]

Note that both axioms are presented in a backwards style. This reflects the style of proof construction we use later.

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

\[
\{P\} \; C \; \{Q\} \quad \{R \, P\} \; C \; \{R \, Q\}
\]

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. Within the context of goal directed proof, it allows us to focus on the correctness of \(C\) within a tight specification, expressed by assertions \(P\) and \(Q\).

An important role for the frame rule is in reasoning about recursively defined procedures. Consider a procedure definition of the form

\[
\text{procedure} \; h(x_1, \ldots, x_m; y_1, \ldots, y_n) \; \text{is} \; C
\]

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:

\[
\{P\} \; h(x_1, \ldots, x_m; y_1, \ldots, y_n) \; \{Q\}
\]

We therefore require the application of a standard proof rule for reasoning about recursively defined procedures [9]:

\[
\{P\} \; h(x_1, \ldots, x_m; y_1, \ldots, y_n) \; \{Q\}
\]

where \(h(x_1, \ldots, x_m; y_1, \ldots, y_n) = C\). This rule reduces the task of proving (3), 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:

\[
\{P\} \; C \; \{Q\}[e_1/v_1, \ldots, e_n/v_n]
\]

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 \(e_j\).

In terms of theorem proving, the recursive hypothesis typically needs to be strengthened in order for a proof attempt to succeed. That is, the footprint of the heap accessed within the procedure body is larger than the footprint expressed by the recursive hypothesis. The frame rule underpins the strengthening step, i.e. given a recursive hypothesis of the form \(\{P\} \; C \; \{Q\}\), then the frame rule allows us to derive \(\{R \, P\} \; C \; \{R \, Q\}\), assuming that no free variable in \(R\) is modified by \(C\). The crucial part of this step lies in finding the appropriate instance of the frame rule. In particular, the invariant \(R\) must be selected so as to support the proof of the procedure body. The identity of \(R\) is essentially a creative step, analogous to strengthening an induction hypothesis within the an inductive proof. Automating the discovery of \(R\) is the central problem that is being addressed by our proposal.

The final proof rule which we will make use of explicitly, is the rule for auxiliary variable elimination:

\[
\{P\} \; C \; \{Q\}
\]

where \(x\) is not free in \(C\).

3 Proof Planning

Central to our proposal is an automated reasoning paradigm called proof planning [5]. Proof planning automates the search for proofs through the use of high-level proof outlines, known as proof plans. A proof plan is represented by a generic proof tactic, and is associated with declarative preconditions. For a given conjecture, these preconditions are used to control the selection and instantiation of generic tactics during proof planning. Once generated, a tactic can then be checked using an appropriate proof checker.
By decoupling issues of search and soundness, proof planning supports the reuse of high-level proof strategies [15]. In addition, the explicit representation of search control information, i.e. generic tactic preconditions, means that partial success can be exploited during the search for a proof. Through a mechanism known as proof critics [13], proof-failure analysis has been used in patching proof attempts. Applications of proof patching include conjecture generalization and lemma discovery [13, 14], loop invariant discovery [11, 12], refining faulty conjectures [16].

3.1 Rippling

The focus of this paper is on reasoning about recursively defined procedures. A proof plan designed for reasoning about recursively defined structures is rippling. 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. The constraints of rippling are imposed through meta-level annotations. To be more precise, given a hypothesis of the form

\[ b \ast c \]  

and a goal of the form \( (a \ast b) \ast c \), then an annotated version of the goal takes the form:

\[ (a \ast b) \uparrow \ast c \]  

Note that the annotated part of the formula, represented by the shading, denotes a mismatch (wave-front) between the goal and the given. The uparrow records the direction in which the wave-front is being moved with respect to the unannotated term structure. A down arrow can also be associated with a wave-front. Directed wave-fronts enable the termination of rippling to be guaranteed, thus eliminating the problem of orienting rewrite rules that occurs within other theorem provers. An identical difference identification process is used to compute the applicable rewrite rules at each step within the planning of a proof.

To illustrate, a 1-step ripple proof of the conjecture shown above can be achieved using an annotated rewrite rule of the form:\footnote{We use \( \Rightarrow \) to denote rewrite rules and \( \rightarrow \) to denote logical implication.}

\[ (X \ast Y) \uparrow \ast Z \Rightarrow X \ast (Y \ast Z) \uparrow \]  

Note that such annotated rewrite rules are known as wave-rules. The application of (6) to (5) gives

\[ a \ast (b \ast c) \uparrow \]  

Note that the unannotated inner subterm, \( b \ast c \), matches (4), leaving a simplified residue of the form \( a \ast \text{true} \). In Boyer and Moore terminology [4], this process of matching parts of a goal with a given hypothesis is called fertilization. In general, as will be illustrated later, a ripple proof may require a few non-rippling steps. These unblocking steps involve the manipulation of wave-fronts so as to enable further rippling. The class of unblocking steps must obviously be defined so as to ensure that termination of rippling is not lost. A completely formal account of the rippling can be found in [2, 6, 8].

3.2 Extending Rippling Annotations

While rippling was designed for reasoning about recursively objects, the notion needs to be extended to deal with pointer programs. That is, an new annotation is required for highlighting differences at the level of pointer references.

We use \( X \uparrow \downarrow \) 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. Conventional rippling deals with multiple given hypotheses, as is the case when reasoning about recursively defined trees for example. In terms of our pointer wave-front proposal, multiple given hypotheses give rise to multiple pointer references associated with a wave-front, e.g. \( \uparrow X \).

Given the nature of separation logic, proofs typically involve existential quantification, where existential variables denote pointer references internal to the heap. Such pointer references represent “potential” differences between goal and given in that the difference can be eliminated by selecting the appropriate existential witness. Following conventional rippling, we use the dotted box annotation to represent potential wave-fronts. Extending this notation for pointer references is straightforward. To illustrate, \( \exists X. P(\uparrow X) \) indicates that the difference between \( X \) and \( Y \), if it exists, is still to be determined.

4 Our General Approach

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

\[ \{P\} \}

\[ \vdots \]

\[ \{R_1 \ast P_1\} \]

\[ h(x_1^1, \ldots, x_1^p; y_1^1, \ldots, y_1^q)\}; \]

\[ \{R_1 \ast Q_1\} \]

\[ \vdots \]

\[ \{R_n \ast P_n\} \]

\[ h(x_n^1, \ldots, x_n^p; y_n^1, \ldots, y_n^q)\}; \]

\[ \{R_n \ast Q_n\} \]

\[ \vdots \]

\[ \{Q\} \]
Note that each recursive call is associated with assertions, i.e. pre- and postconditions. Discovering appropriate intermediate assertions is a major burden, and in practice, a burden that a programmer cannot be expected to address. As noted in §2, this discovery task corresponds to finding instances of the frame rule that will enable a proof to succeed. We aim is to provide automation, and our approach combines failure driven refinement with schematic proof. That is, starting with $R_i$ ($1 \leq i \leq n$) set to true, we incrementally strengthen $R_i$ via a process of proof-failure analysis. We also use meta-variables to delay choice during the search for a proof, in the spirit of middle-out reasoning [7]. This means using meta-variables to represent unknowns with respect to $P_i$ and $Q_i$ ($1 \leq i \leq n$). The basic idea is that by focusing initially on the concrete parts of the verification task, constraints will be generated that guide the refinement and instantiation of the schematic parts. Our notion of failure driven refinement will be illustrated in §5.2.

A strength of assertion based program reasoning is that it partitions the overall verification task into into a number of smaller and independent verification tasks. Here there are 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:

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

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

Verifying intermediate assertions: to prove the $i^{th}$ assertion we are given a hypothesis of the form:

$$R_{i-1} \ast P_{i-1}$$

and a goal of the form:

$$R'_i \ast Q'_i$$

where $R'_i$ and $Q'_i$ are calculated from $R_i$ and $Q_i$ using weakest precondition semantics. Where strong syntactic similarities exist between intermediate assertions, rippling will provide guidance in terms of proof planning.

The observant reader will have noticed that our failure driven refinement approach reduces the independence between verification conditions highlighted above. That is, as assertions are refined via proof-failure analysis, the refinements will need to be propagated across all the sub-verification tasks. With regards to the overall verification task, this interdependence requires a more global perspective. We delay further discussion of this issue until §6. Below in §5, we illustrate our approach using a concrete example.

5 An Example

We now consider the verification of `copylist`, a procedure which takes 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. Our `copylist` procedure is defined as follows:

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

In order 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:

$$\text{list([], Z) } \iff \text{emp } \land \text{Z = nil}$$

$$\text{list([X|Y], Z) } \iff (\exists p. (Z \rightarrow X, p) \ast \text{list(Y, p)})$$

(7)

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:

$$\{\text{list(a, i)}\} \text{copylist(i; j)} \{\text{list(a, i) } \ast \text{list(a, j)}\}$$

(8)

In words, this states: Assuming $i$ points to an acyclic list with contents $a_i$, then if the execution of `copylist(i; j)` terminates then $i$ and $j$ will point to distinct acyclic lists, both containing $a$. Using the procedure proof rule (see §2), the verification of (8) reduces to verifying the procedure body given in Figure 1, under the assumption that the procedure specification holds for the recursive call. Note that intermediate assertions have been added to the code given in Figure 1. These assertions have been added in order to show the extra burden they place on the programmer. With these assertions, we present the verification of `copylist` below in §5.1. In particular, we show the role that rippling plays in guiding such verification proofs. This sets the scene for §5.2, where we show how proof-failure analysis can be used to guide the discovery of the intermediate assertions.

5.1 Proving Partial Correctness

In the remainder of this section we consider the verification of the else-branch associated with `copylist`. With
recursive call, we get to assume the assertions given in Figure 1, this gives rise to 3 verifi-
cation tasks, i.e. pre-recursive call, recursive call and post-recursive call.

5.1.1 Pre-recursive call:

Focusing on the verification of the else-branch, prior to the recursive call, we get to assume

\[ \text{list}(\alpha, i) \wedge \neg(i = \text{nil}) \]  

and have to prove:

\[ (\exists \iota_0'. (i \mapsto \iota_0') \wedge (\exists \iota_1'. (i + 1 \mapsto \iota_1') \wedge (\exists \alpha. ([\alpha]_{\alpha_1} = \alpha) \wedge (i \mapsto \iota_0', \iota_1') \ast \text{list}(\alpha, \iota_1')))) \]  

and by (2) we can derive:

\[ ([\alpha]_{\alpha_1} = \alpha) \wedge (i \mapsto \alpha_h) \ast (i + 1 \mapsto p) \ast \text{list}(\alpha_t, p) \wedge \neg(i = \text{nil}) \]  

Now turning to the goal, reasoning backward from (10) using existential introduction we get

\[ (i \mapsto \alpha_h) \wedge (i + 1 \mapsto p) \wedge ([\alpha]_{\alpha_1} = \alpha) \wedge (\alpha_h = \alpha_h) \wedge (i \mapsto \alpha_h, p) \ast \text{list}(\alpha_t, p) \]

where \( \iota_0', \iota_1', \alpha_h, \alpha_t \) are instantiated to be \( \alpha_h, p, \alpha_h, \alpha_t \) respectively. Unfolding using (2) and (1), followed by simplification, we get:

\[ (i \mapsto \alpha_h \ast \text{true}) \wedge (i + 1 \mapsto p \ast \text{true}) \wedge ([\alpha]_{\alpha_1} = \alpha) \wedge (\alpha_h = \alpha_h) \wedge (i \mapsto \alpha_h) \ast (i + 1 \mapsto p) \ast \text{list}(\alpha_t, p) \]

Fertilization with (13) completes the proof.

5.1.2 Recursive call:

When it comes to proving the recursive call, we have access to a recursive hypothesis of the form:

\[ \{ \text{list}(\alpha, i) \} \ast \text{copylist}(i, j) \{ \text{list}(\alpha, i) \ast \text{list}(\alpha, j) \} \]

and by (14) we must prove:

\[ (i \mapsto \alpha_h) \wedge (i \mapsto \alpha_h) \ast (i \mapsto \iota_0, \iota_1) \ast \text{list}(\alpha_t, i) \ast \text{copylist}(i; j_1) \]

Given (14), we must prove:

\[ (\exists \alpha_h, \alpha_t. ([\alpha]_{\alpha_1} = \alpha) \wedge (i \mapsto \alpha_h) \wedge (i \mapsto \iota_0, \iota_1) \ast \text{list}(\alpha_t, i) \ast \text{copylist}(i; j_1) \]

Note that the [] case gives rise to a trivial contradiction.
By auxiliary variable elimination, (15) reduces to:
\[\{([\alpha_h | \alpha] = \alpha) \land (i_h = \alpha_h) \land (i \mapsto i_h, \iota) * \]
\[copylist(\alpha_i, \iota) \}
\[\{([\alpha_h | \alpha] = \alpha) \land (i_h = \alpha_h) \land (i \mapsto i_h, \iota) * \]
\[list(\alpha_i, \iota) \} \]

Applying the frame rule to (16) gives:
\[\{list(\alpha_i, \iota) \} \land \{copylist(\alpha_i, \iota) \} \{list(\alpha_i, \iota) \} \}

Note that the frame rule allows us to focus on only the parts of the heap that are accessed via the recursive call. Using the substitution rule, we can now instantiate (14), and complete this branch of the proof.

5.1.3 Post-recursive call:

Finally we consider the post-recursive call verification. The rewrite rules required for the verification proof are given in Figure 2. Given the hypothesis
\[(\exists \alpha_h, \alpha_i. ([\alpha_h | \alpha] = \alpha) \land (i_h = \alpha_h) \land (i \mapsto i_h, \iota) * \]
\[\land \{list(\alpha_i, \iota) \} \}
\[(24)\]
the goal is to prove
\[\forall \alpha'. \forall i_h, j_h \forall j' \forall i, j \forall \alpha'' \exists \alpha''' \exists \alpha'''. \left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \rightarrow \left( \land \{\alpha''' \} \land \{list(\alpha_i, \iota) \} \right) \]

Note that this verification task fits with the rippling proof pattern, i.e. there exists a strong syntactic similarity between goal and given. Focusing on the sub-formula \[list(\alpha_i, \iota) \land \{list(\alpha_i, \iota) \} \] within (24), we can get an annotated goal of the form:
\[\forall \alpha'. \forall i_h, j_h \forall j' \forall i, j \forall \alpha'' \exists \alpha''' \exists \alpha'''. \left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \rightarrow \left( \land \{\alpha''' \} \land \{list(\alpha_i, \iota) \} \right) \]

Note that rippling is not directly applicable to (25), i.e. only a partial match between goal the left-hand side of wave-rule (17) exists, as shown below:
\[\left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \rightarrow \left( \land \{\alpha''' \} \land \{list(\alpha_i, \iota) \} \right) \]

Note that the same partial match with (17) exists for sub-term \[\left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \]. Motivated by this partial match, proof-failure analysis suggests replacing the two occurrences of \[\alpha'' \] with wave-fronts of the form \[\left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \]. From (24), we can generate the required wave-fronts via existential instantiation and substitution, i.e. the replacement of \[\alpha \] by \[\alpha_h | \alpha_i \] within the goal to give:
\[\forall \alpha'. \forall i_h, j_h \forall j' \forall i, j \forall \alpha'' \exists \alpha''' \exists \alpha'''. \left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \rightarrow \left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \]

\[\alpha \]

A ripple proof can now be generated as shown in Figure 3. Note that all the unblocking steps involve manipulating wave-fronts so as to progress the rippling.

5.2 Failure Driven Assertion Refinement

We now return to the problem of generating intermediate assertions. Following our general approach outlined in §4, we start with the weakest of possible instantiations for invariant \( R \), which gives:
\[\{true \land list(S, i) \} \land \{copylist(\alpha_i, j) \} \land \{true \land list(S, i) \} \land \{list(S, j) \} \]

Note that \( S \) is a meta-variable, a place holder for missing term structure. The expectation is that proof planning, and in particular the constraints imposed by rippling, will guide the strengthening of \( R \) and the instantiation of \( S \).

We focus here on the post-recursive call verification, as it is the most constrained of the verification tasks. We have a given hypothesis of the form:
\[true \land list(S, i) \land list(S, j) \]

while the goal is to prove
\[\forall \alpha', \forall i_h, j_h \forall j' \forall i, j \forall \alpha'' \exists \alpha''' \exists \alpha'''. \left( \land \{\alpha'' \} \land \{list(\alpha_i, \iota) \} \right) \rightarrow \left( \land \{\alpha''' \} \land \{list(\alpha_i, \iota) \} \right) \]

Note that in proving this goal, the aim is to refine (26), i.e. strengthen (26) in order to obtain (24). The details of the proof-failure analysis and assertion refinement are presented in Figure 4. In summary, the iterative refinement process provides the following instantiation for the invariant \( R \):
\[([\alpha_h | \alpha_i] = \alpha) \land (i_h = \alpha_h) \land (i \mapsto i_h, \iota) \]

Note also that \( S \) is instantiated to be \( \alpha_i \).

Finally, as pointed out in §4, assertion refinements need to be propagated between sub-verification tasks. In terms of the copylist example, this involves communicating the assertion refinement to the proof planning of the recursive and pre-recursive calls. In the case of the recursive call, the frame rule will ensure that the separation constraints are not violated, i.e. \( i, i_h \) and \( i \iota \) are not modified by \[copylist(\alpha_i, j) \]. The recursive call. In terms of the assumption that \( \alpha \) is non-empty (first assertion refinement), note that the justification for the refinement arises from the verification of the pre-recursive call. This illustrates the interdependence that exists between verification tasks, and suggests that a relatively fine grained interleaving between proof planning attempts may prove beneficial.

6 Future and Related Work

Our proposed approach to automating assertion refinement has been tested by-hand on copylist, and the
\[
\text{list}([X \ Y \ Z], \ W) \Rightarrow (\exists p. (Z \leftarrow X, p) \ast \text{list}(Y, \ W))
\]  
\[
(\exists P, R) \ast Q \Rightarrow (\exists P, R \ast Q)
\]  
\[
Q \ast (\exists P, R) \Rightarrow (\exists P, Q \ast R)
\]  
\[
X \ast (Y \ast Z) \Rightarrow Y \ast (X \ast Z)
\]  
\[
X \rightarrow (Y \ast Z) \Rightarrow Z
\]  
\[
Q \rightarrow (\exists P, R) \Rightarrow (\exists P, Q \rightarrow R)
\]

Note that in the case of (18), (19), and (23), \( P \) can not occur free in \( Q \).

Figure 2. Wave-rules and unblocking rules

copytree example given in [18, 19]. 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 §3.2, has been tested, again by-hand, on the verification of an in-place list reversal program given in [19]. 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 [3].

In terms of implementing the proposal, a number of the aspects have been investigated previously. In particular, the ideas on assertion refinement presented above have strong similarities to earlier work on proof patching via conjecture refinement [16]. In [16] non-theorems are refined into theorems by a process of weakening. For instance, a contradictory goal arising from a failed proof attempt is used to suggest a weaker conjecture, i.e. an exception which suggests a conditional conjecture that is provable. The major difference with what is proposed here is the need for global analysis in communicating assertion refinements between separate verification tasks. Proof critics allow for such a global perspective. However, one could also imagine an agent based reasoning architecture providing such a capability.

Finally, in order to advance our work a mechanization of separation logic within a tactic based proof development environment is required. The only such mechanization known to the author has been undertaken by Tjark Weber [20]. Weber’s mechanization was developed within Isabelle/HOL [17]. Building upon Isabelle/HOL would have significant advantages since we could also exploit IsaPlanner [10], an Isabelle based proof planning system.

7 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 the notation of 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, Paul Jackson, Alberto Momigliano, Alan Smaill, and members of the Dependable Systems and Mathematical Reasoning Groups for their feedback on this work.
Figure 3. Verifying post-recursive call of Copylist: ripple guided proof
**First proof attempt:** Following the general approach presented in §4, our initial schematic hypothesis takes the form

\[ \text{true} \ast \text{list}(S, i) \ast \text{list}(S, j) \]

while the annotated goal takes the form:

\[
(\forall j'. (j' \mapsto i_h, j_t) \rightarrow (\text{list}(\alpha, i) \ast \text{list}(\alpha, j_t)))
\]

As was the case in §5.1.3, the ripple proof is initially blocked, i.e. only a partial wave-rule match is possible. However, unlike the proof patch presented in §5.1.3, there is no hypothesis which enables the missing wave-fronts to be introduced. Overcoming the failure requires a casesplit on \( i_h \).

**Second proof attempt:** Assuming \((\exists \alpha_h, \alpha_t, ([\alpha_h | \alpha_t] = \alpha))\), then the given hypothesis becomes

\[
([\alpha_h | \alpha_t] = \alpha) \land \text{true} \ast \text{list}(S, i) \ast \text{list}(S, j)
\]

allowing a ripple proof to develop as before:

\[
(\forall j'. (j' \mapsto i_h, j_t) \rightarrow (\text{list}(\alpha_h, i) \ast \text{list}(\alpha_t, j_t)))
\]

The proof again is blocked. Motivated by wave-rule (21), proof-failure analysis suggests weakening the given hypothesis further by assuming \( i_h = \alpha_h \).

**Third proof attempt:** Assuming \( i_h = \alpha_h \), then the given hypothesis becomes:

\[
([\alpha_h | \alpha_t] = \alpha) \land (i_h = \alpha_h) \land \text{true} \ast \text{list}(S, i) \ast \text{list}(S, j)
\]

Further rippling is now possible:

\[
(\forall j'. (j' \mapsto i_h, j_t) \rightarrow (\text{list}(\alpha_h, i) \ast \text{list}(\alpha_t, j_t)))
\]

Note that the fertilization step instantiates \( S \) to be \( \alpha_t \). Note also that the residue of the ripple proof is not provable in the given context. This suggests refining the hypothesis further by assuming the residue.

**Fourth proof attempt:** Assuming \((i \mapsto i_h, i_t)\), then the given hypothesis becomes:

\[
([\alpha_h | \alpha_t] = \alpha) \land (i_h = \alpha_h) \land (i \mapsto i_h, i_t) \ast \text{list}(S, i) \ast \text{list}(S, j)
\]

Which corresponds to the successful proof presented in Figure 3, with \( S \) being instantiated to be \( \alpha_t \) as a side-effect.

---

**Figure 4. Failure Driven Assertion Refinement**

9
References


