Proof automation for functional correctness in separation logic

Citation for published version:
Maclean, E, Ireland, A & Grov, G 2014, 'Proof automation for functional correctness in separation logic', *Journal of Logic and Computation*. https://doi.org/10.1093/logcom/exu032

Digital Object Identifier (DOI):
10.1093/logcom/exu032

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Journal of Logic and Computation

Proof Automation for Functional Correctness in Separation Logic

Ewen Maclean

University of Edinburgh, UK

Andrew Ireland

Heriot-Watt University, UK

Gudmund Grov

Heriot-Watt University, UK

Abstract

We describe an approach to automatically prove the functional correctness of pointer programs that involve iteration and recursion. Building upon separation logic, our approach has been implemented as a tightly integrated tool chain incorporating a novel combination of proof planning and invariant generation. Starting from shape analysis, performed by the Smallfoot static analyser, we have developed a proof strategy that combines shape and functional aspects of the verification task. By focusing on both iterative and recursive code, we have had to address two related invariant generation tasks, i.e. loop and frame invariants. We deal with both tasks uniformly using an automatic technique called *term synthesis*, in combination with the IsaPlanner/Isabelle theorem prover. In addition, where verification fails, we attempt to overcome failure by automatically generating missing preconditions. We present in detail our experimental results. Our approach has been evaluated on a range of examples, drawn in part from a functional extension to the Smallfoot corpus.

Keywords: Proof planning, functional correctness, invariant discovery, separation logic, substructural logic, automated theorem proving.

1. Introduction

Pointers are a powerful and widely used programming mechanism. Developing and maintaining correct pointer programs is notoriously hard. It is therefore highly desirable to be able to routinely reason about the correctness of pointer programs. The stumbling block to achieving such a goal has been the lack of scalable methods of reasoning. Separation logic [29, 31] holds the promise of providing the logical foundation for building scalable methods. Here we present automated reasoning techniques which support the verification of functional properties of pointer programs formalised in separation logic – both iterative and recursive. In order to highlight the verification challenges our work addresses, consider the two following versions of in-place reversal of an acyclic singly linked list:

Email addresses: `E.Maclean@ed.ac.uk` (Ewen Maclean), `A.Ireland@hw.ac.uk` (Andrew Ireland), `G.Grov@hw.ac.uk` (Gudmund Grov)

```
{$\exists a.\ data\_lseg(a,i,null) \wedge a_0 = a$}
o = null;
while not(i = null) do
   t = i->tl;
   i->tl = o;
   o = i;
   i = t
od
{$\exists b.\, data\_lseg(b,o,null) \wedge a_0 = rev(b)$}
```

```
{$\exists a.\ data\_lseg(a,i,null) * (o \mapsto \_) \wedge a_0 = a$}
list_reverse(s; i, o)  =
   if not(i = null) then
      k = i->tl;
      i->tl = o;
      list_reverse(s; k, i)
   else
      s = o
{$\exists b.\, data\_lseg(b,s,o) * (o \mapsto \_) \wedge a_0 = rev(b)$}
```

Note that the program variables `i` and `o`, within the iterative version on the left, point to the initial and reversed linked lists respectively. In the recursive version on the right, `i` points to the initial list, and `s` points to the final list. Note also that we use `i->tl` to de-reference the next pointer field of `i`, while `i->hd` de-references the data field.

At the specification level, we use the predicate *data_lseg(X, Y, Z)* to denote an acyclic singly linked list, where *X* is the content of a list, i.e. a sequence of data elements, while *Y* and *Z* delimit the actual linked list structure. Moreover, we use *rev* to denote sequence reversal – we delay the formal definition of such notations until later (see §2).

Automating the verification of such programs is challenging. As well as guidance for proof search, auxiliary assertions need to be automatically generated. In the case of the iterative code above, a *loop invariant* is required which needs to specify two disjoint lists, i.e. the list pointed to by *i*, representing the segment that remains to be reversed, and the list pointed to by *j*, representing the segment that has been reversed so far. In separation logic, such an invariant can be represented as follows:

$$(\exists a, b.\ data\_lseg(a,i,null) * data\_lseg(b,o,null) \wedge a_0 = app(rev(b),a)) \tag{1}$$

where *app* denotes sequence concatenation. The disjointness property mentioned above is specified using the $*$ operator, which will be described in more detail in §2.

An analogous assertion generation challenge arises when reasoning about recursion, i.e. a *frame invariant* is required which describes the parts of the heap that are not changed by a recursive call. In the case of `list_reverse` above, the frame invariant takes the form:

$$(o \mapsto \_) * \exists a_h, a_{tl}.\ a_0 = [a_h|a_{tl}] \ldots \tag{2}$$

Note that the $\mapsto$ operator denotes the points-to relation, i.e. o points to a part of heap, separated using the disjoint $*$ operator, from $data\_lseg(a,i,null)$. The simple extra annotation $(o \mapsto \_)$ ensures that the heap described by $data\_lseg(a,i,null)$ is not reachable by o, thus eliminating complicated reachability conditions as described in [31].

Our work directly addresses the challenges outlined above. Moreover we adopt a weakest precondition style analysis which renders our proof techniques applicable to substructural logics in general, but in particular linear logic [13] and the logic of bunched implications [30]. As described in §7, many existing systems use symbolic execution (strongest postcondition analysis) and have proven to be very effective at verifying light weight properties of pointer programs such as shape [2, 33, 11]. Our focus is on functional correctness where weakest precondition analysis gives rise to simpler verification conditions as highlighted in [14]. Other existing systems allow for interactive proof of functional goals, or automatic proofs of goals with restricted functional annotation [24, 34]. Our approach differs from existing work in that it endeavours to automatically verify the functional correctness of pointer programs. While existing systems can prove the correctness of programs presented in this paper, we argue that our methods provide greater automation for fully functional cases. By following the proof planning paradigm we invent compound methods which automate the proof of conjectures expressed in the logic of bunched implications, meaning that our techniques extend beyond software verification and are applicable to other domains. Equally our proof methods are extensible to other data structures; we show some evidence of this in §8.

The work reported here represents a detailed account of the proof strategy introduced in [19], together with our assertion generation technique. Moreover, we also provide a more detailed account of our implementation which was first reported in [20]. The contributions of this paper in relation to existing work are predominantly in the area of proof techniques for automation. Specifically:

- We present a proof technique by which it is possible automatically to prove some program properties that other systems need to do interactively.
- We present a technique for extracting and synthesising a functional specification of a missing invariant.
- The proof techniques we have developed extend to other data structures and logics and are not specific to a particular programming language or paradigm.

Other approaches, which use symbolic execution, are capable of proving correct similar programs as those presented here, but we believe our approach extends beyond Software Verification into Automatic Theorem Proving in General. We know of no existing automatic prover for Separation Logic or the logic of bunched implications for which it is possible to run the same examples in a comparable fashion.

The paper is structured as follows. Background on separation logic is provided in §2. In §3 we describe the proof strategy which underpins our approach, while our invariant and precondition generation mechanism is described in §4. The overall tool chain which implements our approach is described in §5, while our experimental results are presented in §6. Related and future work are discussed in §7 and §8 and we conclude in §9.

2. Separation Logic and Pointer Program Proof

Separation logic was developed as an extension to Hoare logic with the aim of simplifying pointer program proofs. A key feature of the logic is that it focuses the reasoning effort on only those parts of the heap that are relevant to a program – so called local reasoning. Here we give a brief introduction to separation logic, for a full account see [31].

Separation logic extends predicate calculus with new forms of assertions for describing the heap:

- Empty heap: the assertion *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.* X denotes the address of a cell with contents E.
- Separating conjunction: the assertion $P * Q$ holds for a heap if the heap can be divided into two disjoint heaps H_1 and H_2, where P holds for H_1 and Q holds for H_2 simultaneously.
- Separating implication: the assertion $P \mathrel{-\!\!*} Q$ holds for a heap H_1, if whenever H_1 is extended with a disjoint heap H_2, for which P holds, then Q holds for the extended heap.

Typically one wants to assert that a pointer variable, say X, points to E within a larger group of heap cells. This can be represented by $(X \hookrightarrow E)$, which is an abbreviation for $(\mathit{true} * (X \mapsto E))$ and asserting that the heap can be divided into two parts: a singleton heap, for which $(X \mapsto E)$ holds; and the rest of the current heap, for which *true* holds. Note that *true* holds for any heap. In what follows we will focus on pointers that reference a pair of adjacent heap cells. So we will use $(X \mapsto E_1, E_2)$ as an abbreviation for $(X \mapsto E_1) * (X + 1 \mapsto E_2)$. This pair notation can be generalised to any product type, and to records which can be seen as labelled product types. In particular, we will sometimes use $(X \overset{data}{\mapsto} E_1) * (X \overset{next}{\mapsto} E_2)$ for $(X \mapsto E_1) * (X + 1 \mapsto E_2)$ for linked lists. Central to the 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 – where R denotes the frame invariant mentioned earlier. 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.

We will also make significant use of the following rule which expresses the relationship between separating conjunction and separating implication:

$$\frac{P * Q \vdash R}{Q \vdash P \mathrel{-\!\!*} R} \qquad (3)$$

In the system we present in this paper, we employ a verified weakest precondition generator which introduces the $-\!\!*$ symbol for each statement in the program being analysed. It is possible to use the above rule to eliminate the use of the $-\!\!*$ symbol, but the functional verification challenges remain. In our case we need to prove goals of the form

$$Q \vdash P' * (P -\!\!* R) \tag{4}$$

where P' can be viewed as a heap update of P performed by some program operation. In this case we need to reason about the $-\!\!*$ operator, and make explicit use of the rule

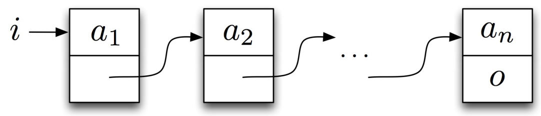
$$Y \vdash X -\!\!* (X * Y)$$

which is desribed in more detail in §3.

Returning to the examples introduced in §1, we now formally define the *data_lseg* predicate:

$$\begin{aligned} \mathit{data\_lseg}([], Y, Z) &\leftrightarrow \mathit{emp} \wedge Y = Z & (5)\\ \mathit{data\_lseg}([W|X], Y, Z) &\leftrightarrow (\exists p.\ (Y \mapsto W, p) * \mathit{data\_lseg}(X, p, Z)) & (6)\end{aligned}$$

Note that the first argument denotes a sequence, where sequences are represented using the Prolog list notation. The second and third arguments delimit the corresponding linked-list structure. This definition excludes cycles, and can been seen pictorially as follows for the predicate $\mathit{data\_lseg}(a, i, o)$:



where i and o delimit the singly linked-list representing a, *i.e.* the sequence $[a_1, a_2, \ldots, a_n]$.

To illustrate the nature of program proof in separation logic, consider the following simple specification:

$\{\mathit{data\_lseg}([1, 2], i, \mathit{null})\}$
```
   j = i->tl;
   i->tl = null;
```
$\{\mathit{data\_lseg}([1], i, \mathit{null}) * \mathit{data\_lseg}([2], j, \mathit{null})\}$

The precondition states that when applied to a singly linked-list of length two, with content $[1, 2]$, then the code fragment splits the list into a pair of disjoint singleton lists, with contents 1 and 2 respectively. The specification is also shown pictorially in Figure 1.

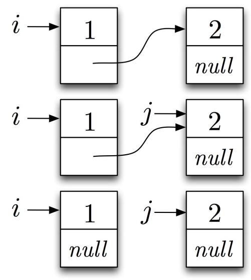
While the code fragment is trivial, it provides enough structure in order to demonstrate program verification within the context of separation logic. Using *weakest precondition* (WP) analysis, intermediate assertions can be calculated for the code given above as follows:

$\{\mathit{data\_lseg}([1, 2], i, \mathit{null})\}$
$\{(i \mapsto \mathcal{F}_1, \mathcal{F}_2) * ((i \mapsto \mathcal{F}_1, \mathit{null}) -\!\!* (\mathit{data\_lseg}([1], i, \mathit{null}) * \mathit{data\_lseg}([2], \mathcal{F}_4, \mathit{null}))) \wedge (i \hookrightarrow \mathcal{F}_3, \mathcal{F}_4)\}$
```
   j = i->tl;
```
$\{(i \mapsto \mathcal{F}_1, \mathcal{F}_2) * ((i \mapsto \mathcal{F}_2, \mathit{null}) -\!\!* (\mathit{data\_lseg}([1], i, \mathit{null}) * \mathit{data\_lseg}([2], j, \mathit{null})))\}$
```
   i->tl = null;
```
$\{\mathit{data\_lseg}([1], i, \mathit{null}) * \mathit{data\_lseg}([2], j, \mathit{null})\}$

Note that $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$ and $\mathcal{F}_4$ denote meta-variables. Verification corresponds to proving that the precondition implies the calculated WP, i.e.

$$\begin{aligned}&\mathit{data\_lseg}([1, 2], i, \mathit{null}) \vdash\\ &\quad (i \mapsto \mathcal{F}_1, \mathcal{F}_2) * ((i \mapsto \mathcal{F}_1, \mathit{null}) -\!\!* (\mathit{data\_lseg}([1], i, \mathit{null}) * \mathit{data\_lseg}([2], \mathcal{F}_4, \mathit{null}))) \wedge (i \hookrightarrow \mathcal{F}_3, \mathcal{F}_4)\end{aligned} \tag{7}$$

How we automatically prove such *verification conditions* (VCs) is described below in §3.



$\{data\_lseg([1,2], i, null)\}$

```
j = i->tl;
i->tl = null;
```

$\{data\_lseg([1], i, null) * data\_lseg([2], j, null)\}$

The code specified above (right) splits a singly linked list of length two into a pair of disjoint singleton lists. The specification is represented using pre- and postconditions. The pre-heap, intermediate heap (between the two commands), and post-heap are show pictorially above (left).

Figure 1: Specification and code for split-pair

3. Proof Strategy

Our overall strategy for reasoning about separation logic VCs involves two interacting parts, as shown in Figure 2. The *mutation* strategy is responsible for eliminating $-\!\!*$ operators from a VC, in preparation for the *heap equivalence* strategy, i.e. checking that the shape of the heaps specified on either side of the turnstile match. Note that the strategy is iterative, requiring the decomposition of heap structures followed by further applications of mutation. A key feature of the strategy is that functional constraints are generated as a side-effect of mutation and heap equivalence, giving rise to purely functional VCs. As hinted in §1, we use IsaPlanner/Isabelle to discharge the functional VCs. Below we describe in detail both parts and illustrate their application.

3.1. Mutation

Proving VCs such as (7), gives rise to the following patterns of reasoning:

$$\begin{array}{ccc}
\vdash (\ldots R -\!\!* Q \ldots) & \qquad & \ldots P \ldots \vdash S * Q \\
\vdots & & \vdots \\
\vdash (\ldots R -\!\!* (S * Q') \ldots) & & \ldots R * P' \ldots \vdash S * Q \\
\vdash \ldots Q' \ldots & & \ldots P' \ldots \vdash (R -\!\!* (S * Q)) \\
 & & \ldots P' \ldots \vdash Q
\end{array}$$

Note that both patterns contain a common core – the selection, attraction and cancellation of heaplets R and S. Crucially R and S are unifiable, and occur on either side of the $-\!\!*$ operator. The pivotal cancellation step corresponds to an application of the following rewrite rule[1]

$$(X -\!\!* (X * Y)) \Rightarrow Y \tag{8}$$

Note that this pattern arises when reasoning about programs that mutate the heap, i.e. programs that allocate or update heaplets. We have represented this pattern of reasoning as a proof strategy which we call *mutation*. As indicated above, the pattern may involve just the goal, but also a given hypothesis. For the purposes of explaining the mutation

[1]We use $\Rightarrow$ to denote the rewrite relation and $\rightarrow$ to denote logical implication.

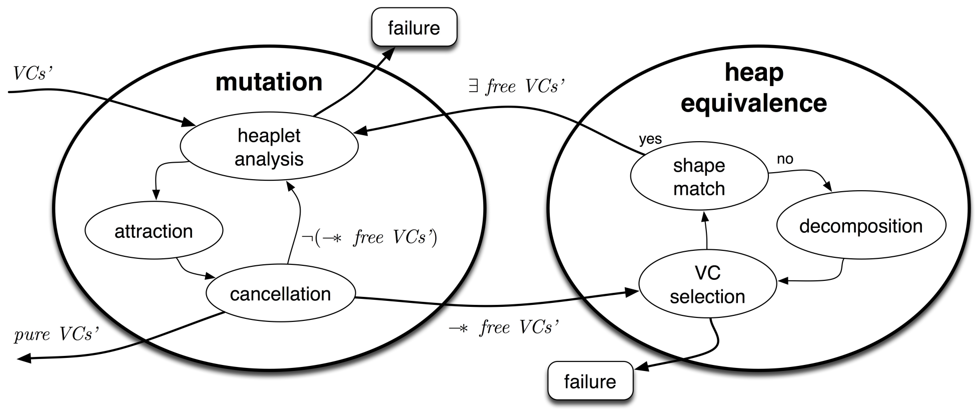


Figure 2: Overall proof strategy.

strategy, we will use meta-level annotations. Specifically, we will refer to a heaplet X that occurs on the left-hand side of a $\mathrel{-\!\!*}$ as an *anti-heaplet*, and annotate it as $\boxed{X}^-$, alternatively if X occurs on the right-hand side of a $\mathrel{-\!\!*}$ it will be annotated as $\boxed{X}^+$. Using these meta-level annotations, the general pattern of mutation is illustrated in Figure 3.

As will be illustrated later, for a given verification, multiple applications of the mutation strategy will be required in order to complete a proof. But first we will describe in more detail mutation in terms of its lower level parts, i.e. heaplet analysis, attraction and cancellation.

$$\begin{array}{c}
\vdash (\ldots \boxed{R}^- \mathrel{-\!\!*} Q \ldots) \\
\vdots \\
\vdash (\ldots \boxed{R}^- \mathrel{-\!\!*} (\boxed{S}^+ * Q') \ldots) \\
\vdash \ldots Q' \ldots
\end{array}
\qquad\qquad
\begin{array}{c}
\ldots P \ldots \vdash \boxed{S}^+ * Q \\
\vdots \\
\ldots \boxed{R}^- * P' \ldots \vdash \boxed{S}^+ * Q \\
\ldots P' \ldots \vdash (\boxed{R}^- \mathrel{-\!\!*} (\boxed{S}^+ * Q)) \\
\ldots P' \ldots \vdash Q
\end{array}$$

goal mutation **hypothesis mutation**

Figure 3: General pattern of mutation

3.1.1. Heaplet Analysis:

The process of identifying a heaplet R that cancels an anti-heaplet S may involve choice. In the presentation that follows we give example heaplets and anti-heaplets as pointer cells, but in general a heaplet and anti-heaplet can be any structure existing within the heap. Below we list the heuristics that we use in identifying such heaplets[2]:

[2]In order to simplify our description we exploit the fact that $P \vdash Q$ and $\vdash P \mathrel{-\!\!*} Q$ are equivalent (see rule (3)).

1. **Explicit case:** there exists a heaplet and anti-heaplet which unify, i.e.

$$(\ldots * \boxed{(x \mapsto y, z)}^{-} * \ldots) \mathrel{-\!\!*} (\ldots * \boxed{(x' \mapsto y', z')}^{+} * \ldots)$$

where x, y, z and x', y', z' unify respectively.

2. **Implicit case:** there exists a heaplet but no explicit anti-heaplet which unifies, or vice versus:
   (a) Where an explicit anti-heaplet $\boxed{(x \mapsto \_, y)}^{-}$ exists, a complementary heaplet introduced by either unfolding the "head" of a *data_lseg* predicate, i.e.

$$\begin{array}{l} \ldots \mathrel{-\!\!*} (\ldots * \mathit{data\_lseg}(\_, \boxed{x}^{+}, \_) * \ldots) \\ \ldots \mathrel{-\!\!*} (\ldots * (\boxed{(x \mapsto \_, T)}^{+} * \mathit{data\_lseg}(\_, T, \_)) * \ldots) \end{array}$$

   or the "tail" of a *data_lseg* predicate:

$$\begin{array}{l} \ldots \mathrel{-\!\!*} (\ldots * \mathit{data\_lseg}(\_, \_, \boxed{y}^{+}) * \ldots) \\ \ldots \mathrel{-\!\!*} (\ldots * (\mathit{data\_lseg}(\_, \_, T) * \boxed{(T \mapsto \_, y)}^{+}) * \ldots) \end{array}$$

   (b) Where an explicit heaplet $\boxed{(x \mapsto \_, y)}^{+}$ exists, a complementary anti-heaplet is introduced by either unfolding the "head" of a *data_lseg* predicate, i.e.

$$\begin{array}{r} (\ldots(\ldots * \mathit{data\_lseg}(\_, \boxed{x}^{-}, \_)\ldots) \mathrel{-\!\!*} \ldots) \\ (\ldots(\ldots * (\boxed{(x \mapsto \_, T)}^{-} * \mathit{data\_lseg}(\_, T, \_)) * \ldots) \mathrel{-\!\!*} \ldots) \end{array}$$

   or the "tail" of a *data_lseg* predicate, i.e.

$$\begin{array}{r} ((\ldots * \mathit{data\_lseg}(\_, \_, \boxed{y}^{-}) * \ldots) \mathrel{-\!\!*} \ldots) \\ ((\ldots * (\mathit{data\_lseg}(\_, \_, T) * \boxed{(T \mapsto \_, y)}^{-}) * \ldots) \mathrel{-\!\!*} \ldots) \end{array}$$

   (c) In general, the required (anti-)heaplet may be embedded within a *data_lseg*. In the case of a heaplet, this involves unfolding in the presence of a meta-variable ($\mathcal{F}_i$), while in the case of an anti-heaplet unfolding will be in the presence of a skolem constant, i.e. $\mathcal{X}_j$. At the level of annotations, we use a dotted box to indicate the potential for decomposing (anti-)heaplets, e.g. $\boxed{\mathcal{X}_i}^{-}$ and $\boxed{\mathcal{F}_j}^{+}$.

It should be noted that decomposition refers to a general decomposition of a linked list segment, which generalises unfolding via the rules of the data structure. By definition, separation logic ensures that if an exact head pointer match is found then no other potential match is possible, i.e. a pointer cannot reference two distinct parts of the heap at the same time. However, this is not the case for tail decomposition, or decomposition from within a list segment, since the tail pointer of more than one list segment can point to the same heap location.

3.1.2. Heaplet Attraction and Cancellation:

Once heaplet analysis is complete, rewriting is used to attract the selected heaplet and anti-heaplet so as to enable a cancellation step:

$$\begin{array}{rcll} ((\ldots) * \boxed{(x \mapsto y, z)}^{-} * (\ldots)) & \mathrel{-\!\!*} & ((\ldots) * \boxed{(x' \mapsto y', z')}^{+} * (\ldots)) & \\ \vdots & & \vdots & \\ ((\ldots) * (\ldots) * (\boxed{(x \mapsto y, z)}^{-} & \mathrel{-\!\!*} & (\boxed{(x' \mapsto y', z')}^{+} * ((\ldots) * (\ldots))) & \\ & & & \text{by (8)} \\ ((\ldots) * (\ldots)) & * & ((\ldots) * (\ldots)) & \end{array}$$

The rewriting that proceeds the cancellation step involves the application of rewrite rules which are derived from basic properties of separation logic, *e.g.*

$$X * Y \quad \Rightarrow \quad Y * X \tag{9}$$

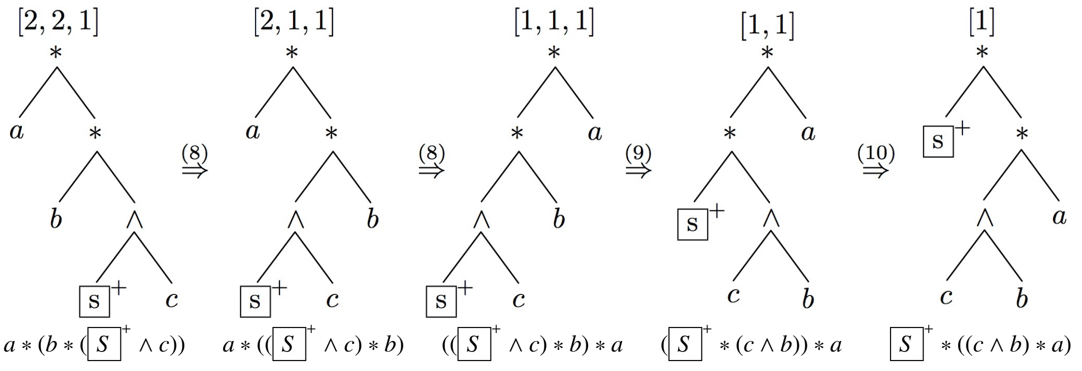
$$pure(Z) \rightarrow (X \wedge Z) * Y \quad \Rightarrow \quad X * (Z \wedge Y) \tag{10}$$

$$(X * Y) * Z \quad \Rightarrow \quad X * (Y * Z) \tag{11}$$

where *pure* is a predicate which denotes that its argument has no shape content. The full list of such rules is too large to include in this presentation.

Mutation selects rewrite rules which reduce the term tree distance between a heaplet and its anti-heaplet. There is no search in this process. To illustrate this consider the term tree manipulation shown in Figure 4. The term $\boxed{S}^{+}$ is moved within the term $a * (b * (\boxed{S}^{+} \wedge c))$ and transformed into the term $\boxed{S}^{+} * ((c \wedge b) * a)$, in order that rule (8) can apply.

To illustrate an informal argument for termination of mutation, assume the position of a term within a term tree is given by a list of branch indices, starting at 1. Application of commutativity rules such as (9) reduces the number of occurrences of the index 2 within the position tree. Application of associativity rules such as (10) reduces the length of the position tree. The process terminates when the position tree [1] is reached, indicating that the desired term is at the top of the tree.



$[2,2,1]$
$[2,1,1]$
$[1,1,1]$
$[1,1]$
$[1]$
$\overset{(8)}{\Rightarrow}$
$\overset{(8)}{\Rightarrow}$
$\overset{(9)}{\Rightarrow}$
$\overset{(10)}{\Rightarrow}$
$a * (b * (\boxed{S}^{+} \wedge c))$
$a * ((\boxed{S}^{+} \wedge c) * b)$
$((\boxed{S}^{+} \wedge c) * b) * a$
$(\boxed{S}^{+} * (c \wedge b)) * a$
$\boxed{S}^{+} * ((c \wedge b) * a)$

Figure 4: Moving the desired term to the top of the tree (position of term on top and formula at bottom of each tree)

Now we return to the split-pair example, and an annotated version of (7):

$$data\_lseg([1,2], \boxed{i}^{-}, null) \vdash$$

$$\boxed{(i \mapsto \mathcal{F}_1, \mathcal{F}_2)}^{+} * (\boxed{(i \mapsto \mathcal{F}_1, null)}^{-} \mathrel{-\!\!*} (data\_lseg([1], \boxed{i}^{+}, null) * data\_lseg([2], \boxed{\mathcal{F}_4}^{+}, null))) \wedge \boxed{(i \hookrightarrow \mathcal{F}_3, \mathcal{F}_4)}^{+} \tag{12}$$

The anti-heaplet on the left-hand side of the $-\!\!*$ triggers the selection of a complementary heaplet on the right-hand side of the $-\!\!*$. Selection requires unfolding, and as described above, the concrete case, i.e. the list pointed to by $\boxed{i}^{+}$, is preferred over the variables case, i.e. the list pointed to by $\boxed{\mathcal{F}_4}^{+}$. Unfolding using (6), (12) becomes:

$$\begin{aligned} & data\_lseg([1,2], \boxed{i}^{-}, null) \vdash \\ & \quad \boxed{(i \mapsto \mathcal{F}_1, \mathcal{F}_2)}^{+} * (\boxed{(i \mapsto \mathcal{F}_1, null)}^{-} \mathrel{-\!\!*} \\ & \quad\quad ((\boxed{(i \mapsto 1, \mathcal{F}_5)}^{+} * data\_lseg([], \boxed{\mathcal{F}_5}^{+}, null)) * data\_lseg([2], \boxed{\mathcal{F}_4}^{+}, null)) \ \wedge \boxed{(i \hookrightarrow \mathcal{F}_3, \mathcal{F}_4)}^{+} \end{aligned} \tag{13}$$

Using (11), the heaplet $\boxed{(i \mapsto 1, \mathcal{F}_5)}^{+}$ is attracted towards the anti-heaplet to give:

$$data\_lseg([1,2], \boxed{i}^{-}, null) \vdash$$

$$\begin{aligned}&\boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+} * (\boxed{(i \mapsto \mathcal{F}_1, \mathit{null})}^{-} \mathrel{-\!\!*}\\&\quad(\boxed{(i \mapsto 1,\mathcal{F}_5)}^{+} * (\mathit{data\_lseg}([],\dbox{\mathcal{F}_5}^{+},\mathit{null}) * \mathit{data\_lseg}([2],\dbox{\mathcal{F}_4}^{+},\mathit{null}))) \wedge \boxed{(i \hookrightarrow \mathcal{F}_3,\mathcal{F}_4)}^{+}\end{aligned}\tag{14}$$

Cancellation is now possible via an application of (8), reducing the goal to:

$$\begin{aligned}&\mathit{data\_lseg}([1,2],\boxed{i}^{-},\mathit{null}) \vdash\\&\quad\boxed{(i \mapsto 1,\mathcal{F}_2)}^{+} * (\mathit{data\_lseg}([],\mathit{null},\mathit{null}) * \mathit{data\_lseg}([2],\dbox{\mathcal{F}_4}^{+},\mathit{null})) \wedge \boxed{(i \hookrightarrow \mathcal{F}_3,\mathcal{F}_4)}^{+}\end{aligned}\tag{15}$$

Simplification using (5) finishes the first application of the mutation strategy, reducing (15) to:

$$\mathit{data\_lseg}([1,2],\boxed{i}^{-},\mathit{null}) \vdash \boxed{(i \mapsto 1,\mathcal{F}_2)}^{+} * \mathit{data\_lseg}([2],\dbox{\mathcal{F}_4}^{+},\mathit{null}) \wedge \boxed{(i \hookrightarrow \mathcal{F}_3,\mathcal{F}_4)}^{+}\tag{16}$$

Further progress requires a second application of the mutation strategy. This time selection of an anti-heaplet is required in order to cancel $\boxed{(i \mapsto 1,\mathcal{F}_2)}^{+}$ within (16). Using (6), the *data_lseg* predicate within (16) is unfolded to give:

$$\boxed{(i \mapsto 1,\mathcal{X}_1)}^{-} * \mathit{data\_lseg}([2],\dbox{\mathcal{X}_1}^{-},\mathit{null}) \vdash \boxed{(i \mapsto 1,\mathcal{F}_2)}^{+} * \mathit{data\_lseg}([2],\dbox{\mathcal{F}_4}^{+},\mathit{null}) \wedge \boxed{(i \hookrightarrow \mathcal{F}_3,\mathcal{F}_4)}^{+}$$

Note that $\mathcal{X}_1$ denotes a skolem constant. Focusing firstly on the left-hand conjunct in the goal, we use attraction to move the anti-heaplet from the hypothesis into the goal. This is achieved using (3), giving:

$$\mathit{data\_lseg}([2],\dbox{\mathcal{X}_1}^{-},\mathit{null}) \vdash \boxed{(i \mapsto 1,\mathcal{X}_1)}^{-} \mathrel{-\!\!*} (\boxed{(i \mapsto 1,\mathcal{F}_2)}^{+} * \mathit{data\_lseg}([2],\dbox{\mathcal{F}_4}^{+},\mathit{null}))\tag{17}$$

Again using (8), cancellation is possible and we are left with

$$\mathit{data\_lseg}([2],\dbox{\mathcal{X}_1}^{-},\mathit{null}) \vdash \mathit{data\_lseg}([2],\dbox{\mathcal{F}_4}^{+},\mathit{null})\tag{18}$$

which by instantiation of $\mathcal{F}_4$ to be $\mathcal{X}_1$ simplifies to true. Returning to the right-hand conjunct of (17), we are left to prove:

$$\boxed{(i \mapsto 1,\mathcal{X}_1)}^{-} * \mathit{data\_lseg}([2],\dbox{\mathcal{X}_1}^{-},\mathit{null}) \vdash \boxed{(i \hookrightarrow \mathcal{F}_3,\mathcal{X}_1)}^{+}\tag{19}$$

Recalling that $\hookrightarrow$ is an abbreviation, (19) simplifies to give:

$$\boxed{(i \mapsto 1,\mathcal{X}_1)}^{-} * \mathit{data\_lseg}([2],\dbox{\mathcal{X}_1}^{-},\mathit{null}) \vdash \boxed{(i \mapsto \mathcal{F}_3,\mathcal{X}_1)}^{+} * \mathit{true}$$

which by further attraction, i.e.

$$\mathit{data\_lseg}([2],\dbox{\mathcal{X}_1}^{-},\mathit{null}) \vdash \boxed{(i \mapsto 1,\mathcal{X}_1)}^{-} \mathrel{-\!\!*} (\boxed{(i \mapsto \mathcal{F}_3,\mathcal{X}_1)}^{+} * \mathit{true})$$

and cancellation instantiates $\mathcal{F}_3$ to be 1, leaving the residue:

$$\mathit{data\_lseg}([2],\boxed{\mathcal{X}_1}^{-},\mathit{null}) \vdash \mathit{true}$$

which is trivial, given that *true* holds for any heap. In the presentation of this simple example, we have unfolded functional structure within each step for ease of presentation, but in general a functional residue is generated which needs to be proved, as is discussed below.

3.1.3. Verification of Loop-Based Code

Now we return to the iterative version of list reversal given in §1, and specifically we show that the mutation strategy introduced above automates loop invariant proofs. Using invariant (1), the verification of the loop body becomes:

$$\{(\exists a,b.\ \ data\_lseg(a,i,null) * data\_lseg(b,o,nil)\ \ \wedge a_0 = app(rev(b),a)) \wedge \neg(i = null)\}$$

```
t = i->tl;
i->tl = o;
o = i;
i = t
```

$$\{(\exists a,b.\ \ data\_lseg(a,i,null) * data\_lseg(b,o,nil)\ \ \wedge a_0 = app(rev(b),a))\}$$

and gives rise to the following annotated VC:

$$\begin{array}{l} data\_lseg(\mathcal{X}_a,\boxed{i}^{-},null) * data\_lseg(\mathcal{X}_b,\boxed{o}^{-},null)\ \wedge a_0 = app(rev(\mathcal{X}_b),\mathcal{X}_a))\ \wedge \neg(i = null) \vdash \\ \quad \boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+} * (\boxed{(i \mapsto \mathcal{F}_1,o)}^{-} \mathrel{-\!\!*} \\ \qquad\qquad data\_lseg(\mathcal{F}_a,\boxed{\mathcal{F}_4}^{+},null) * data\_lseg(\mathcal{F}_b,\boxed{i}^{+},null)\ \wedge \boxed{(i \hookrightarrow \mathcal{F}_3,\mathcal{F}_4)}^{+}\ \wedge a_0 = app(rev(\mathcal{F}_b),\mathcal{F}_a) \end{array} \tag{20}$$

Note that the existential variable a within the hypotheses is replaced by the skolem constant $\mathcal{X}_a$ and in the goal it is replaced by the meta-variable $\mathcal{F}_a$. We now give formal definitions of *rev* and *app*:

$$\begin{array}{rcl@{\qquad\qquad}rcl} rev([]) &=& [] & app([],Z) &=& Z \\ rev([X|Y]) &=& app(rev(Y),[X]) & app([X|Y],Z) &=& [X \mid app(Y,Z)] \end{array}$$

Proving (20) requires three applications of the mutation strategy. Here we only sketch the high-level pattern. The first application focuses on the goal:

$$\begin{array}{ll} \ldots \vdash & \ldots * (\boxed{(i \mapsto \mathcal{F}_1,o)}^{-} \mathrel{-\!\!*} \ldots * data\_lseg(\mathcal{F}_b,\boxed{i}^{+},null)) \wedge \ldots \\ \ldots \vdash & \ldots * (\boxed{(i \mapsto \mathcal{F}_1,o)}^{-} \mathrel{-\!\!*} \ldots * (\boxed{(i \mapsto \mathcal{F}_{b_{hd}},\mathcal{F}_6)}^{+} * data\_lseg(\mathcal{F}_{b_{tl}},\mathcal{F}_6,null))) \wedge \ldots \\ \ldots \vdash & \ldots * (\boxed{(i \mapsto \mathcal{F}_1,o)}^{-} \mathrel{-\!\!*} (\boxed{(i \mapsto \mathcal{F}_{b_{hd}},\mathcal{F}_6)}^{+} * (\ldots * data\_lseg(\mathcal{F}_{b_{tl}},\mathcal{F}_6,null)))) \wedge \ldots \\ \ldots \vdash & \ldots * (\ldots * data\_lseg(\mathcal{F}_{b_{tl}},\mathcal{F}_6,null)) \wedge \ldots \end{array}$$

With all the $\mathrel{-\!\!*}$ operators eliminated, the heap equivalence part of the strategy becomes applicable. Typically heap equivalence requires a generalised notion of decomposition, but in the case of list reversal, heap equivalence results in a simple decomposition of list segment $data\_lseg(\mathcal{X}_a,\boxed{i}^{-},null)$ within a hypothesis, as shown below:

$$\begin{array}{rl} data\_lseg(\mathcal{X}_a,\boxed{i}^{-},null) * \ldots \vdash & (\boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+} * \ldots) \\ (\boxed{(i \mapsto \mathcal{X}_{a_{hd}},\mathcal{X}_1)}^{-} * data\_lseg(\mathcal{X}_{a_{tl}},\mathcal{X}_1,null)) * \ldots \vdash & (\boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+} * \ldots) \end{array}$$

Note that this decomposition step corresponds to unfolding the definition of *data_lseg* and is motivated by the occurrence of $\boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+}$ within the conclusion. The full generality of the decomposition step will be explained in §3.2.

A second application of mutation is now required. Focusing on the left conjunct within the goal, we obtain the following high-level pattern:

$$\begin{array}{rl} (\boxed{(i \mapsto \mathcal{X}_{a_{hd}},\mathcal{X}_1)}^{-} * data\_lseg(\mathcal{X}_{a_{tl}},\mathcal{X}_1,null)) * \ldots \vdash & (\boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+} * \ldots) \\ data\_lseg(\mathcal{X}_{a_{tl}},\mathcal{X}_1,null) * \ldots \vdash & (\boxed{(i \mapsto \mathcal{X}_{a_{hd}},\mathcal{X}_1)}^{-} \mathrel{-\!\!*} (\boxed{(i \mapsto \mathcal{F}_1,\mathcal{F}_2)}^{+} * \ldots)) \\ data\_lseg(\mathcal{X}_{a_{tl}},\mathcal{X}_1,null) * \ldots \vdash & (\ldots) \end{array}$$

where $\mathcal{F}_1$ and $\mathcal{F}_2$ are instantiated to be $\mathcal{X}_{a_{hd}}$ and $\mathcal{X}_1$ respectively. Turning to the right conjunct we require a third application of mutation:

$$\begin{array}{rl}
\boxed{(i \mapsto \mathcal{X}_{a_{hd}}, \mathcal{X}_1)}^{-} * \ldots \vdash & \boxed{(i \hookrightarrow \mathcal{F}_3, \mathcal{F}_4)}^{+} \wedge \ldots \\
\boxed{(i \mapsto \mathcal{X}_{a_{hd}}, \mathcal{X}_1)}^{-} * \ldots \vdash & (\boxed{(i \mapsto \mathcal{F}_3, \mathcal{F}_4)}^{+} * \mathit{true}) \wedge \ldots \\
\ldots \vdash & \boxed{(i \mapsto \mathcal{X}_{a_{hd}}, \mathcal{X}_1)}^{-} \mathrel{-\!\!*} ((\boxed{(i \mapsto \mathcal{F}_3, \mathcal{F}_4)}^{+} * \mathit{true}) \wedge \ldots) \\
\ldots \vdash & \boxed{(i \mapsto \mathcal{X}_{a_{hd}}, \mathcal{X}_1)}^{-} \mathrel{-\!\!*} (\boxed{(i \mapsto \mathcal{F}_3, \mathcal{F}_4)}^{+} * (\mathit{true} \wedge \ldots)) \\
\ldots \vdash & \mathit{true} \wedge \ldots
\end{array}$$

where $\mathcal{F}_3$ and $\mathcal{F}_4$ are instantiated to be $\mathcal{X}_{a_{hd}}$ and $\mathcal{X}_1$ respectively. We are left to verify the functional residue:

$$\ldots \wedge a_0 = \mathit{app}(\mathit{rev}(\mathcal{X}_b), [\mathcal{X}_{a_{hd}} | \mathcal{X}_{a_{tl}}]) \quad \vdash \ldots \wedge a_0 = \mathit{app}(\mathit{rev}([\mathcal{X}_{a_{hd}} | \mathcal{X}_b]), \mathcal{X}_{a_{tl}})$$

where $\mathcal{F}_a$, $\mathcal{F}_{b_{hd}}$ and $\mathcal{F}_{b_{tl}}$ are instantiated to be $\mathcal{X}_{a_{tl}}$, $\mathcal{X}_{a_{hd}}$ and $\mathcal{X}_b$ respectively. As will be explained later, we rely upon IsaPlanner to automate proof search for the functional residue.

3.1.4. Discussion of Design Choices for Mutation

As noted above, the focus of mutation is to eliminate all $\mathrel{-\!\!*}$ symbols introduced by the weakest precondition generator. The mutation strategy is implemented within a lightweight proof-planning system [5]. The system incorporates a planner which performs a search for a proof, by combing proof "methods" combined using an explicit plan. Search is incorporated whenever a choice point arises in the proof. The system itself is described in greater detail in §5.

Following the proof-planning paradigm, the components of the mutation strategy are applied using a depth-bounded heuristic depth-first search (see §5) which uses heaplet analysis to rank the candidates for attraction and cancellation as described in §3.1.1. The heuristics described above for heaplet analysis in combination with the search strategy, determine the heaplet chosen for attraction and cancellation. At each point where there is a choice of heaplet, a branch in the search space takes place. The strategy chosen is to follow the best option and if a point where no further progress can be made, then backtracking in the search takes place.

The preferences deciding the order in which to select heaplets is described in more detail in §3.1.1. The results of choosing such a heuristic order are described in the evaluation in §6. The motivation for choosing this heuristic order is to choose the heaplet which introduces the least number of existentially quantified variables, since each existential instantiation choice introduces a potentially infinite branch point.

3.2. Heap Equivalence

We now describe in detail the heap equivalence part of the proof strategy. As shown in Figure 2, heap equivalence involves three processes, i.e. *VC selection*, *shape match* and *decomposition*. VC selection selects one of the open VCs, and depends on the search strategy. The details of the *shape match* and *decomposition* algorithms are are summarised in Figure 5. In order to fully describe heap equivalence we now consider /tooliterative list append, an example which requires a decomposition other than a simple unfolding step:

```
list_append(x;y) {data_list(a, x) * data_list(b, y)}
  local n,t;
  if (x == NULL) then
    x = y
  else
    t = x;
    n = t->tl;
    while (n != NULL) do
```

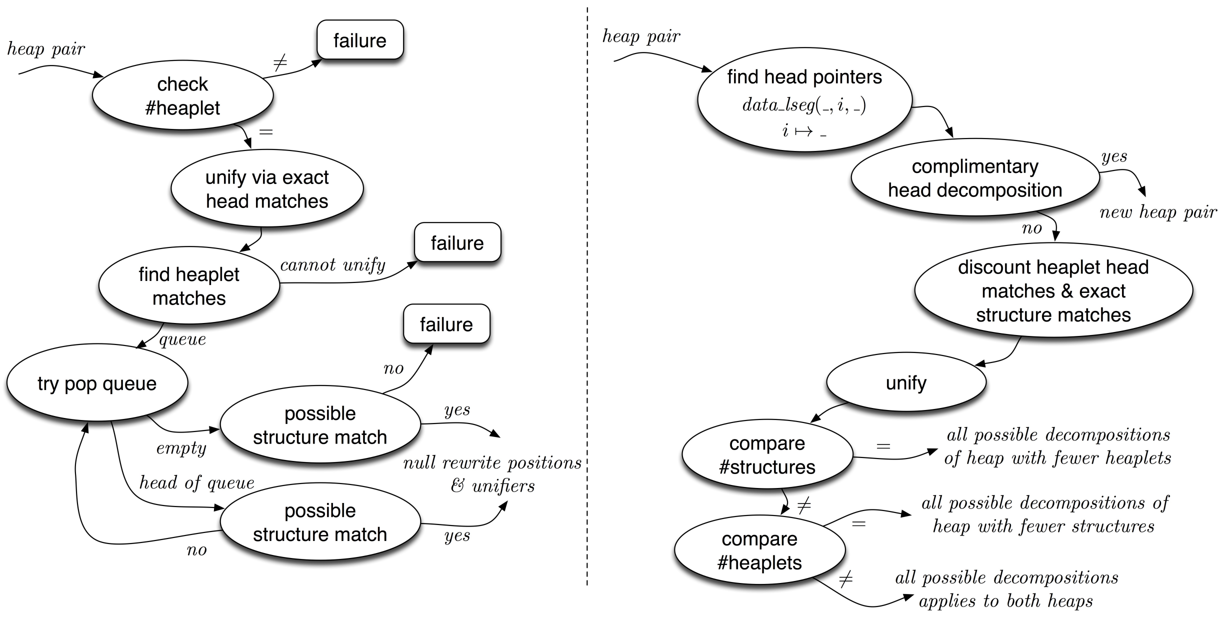

heap pair
check #heaplet
≠
failure
=
unify via exact head matches
find heaplet matches
cannot unify
failure
queue
try pop queue
no
failure
possible structure match
yes
null rewrite positions & unifiers
empty
head of queue
possible structure match
yes
no
heap pair
find head pointers
data_lseg(_, i, _)
i ↦ _
complimentary head decomposition
yes
new heap pair
no
discount heaplet head matches & exact structure matches
unify
compare #structures
=
all possible decompositions of heap with fewer heaplets
≠
compare #heaplets
=
all possible decompositions of heap with fewer structures
≠
all possible decompositions applies to both heaps

Figure 5: The shape match (left) and decomposition (right) strategies within heap equivalence.

```
    {$\exists \alpha,\beta,h.data\_lseg(\alpha,x,t) * (t \mapsto h,n) * data\_list(\beta,n) * data\_list(b,y) \wedge n \neq x \wedge app(\alpha,[h|\beta]) = a$}
      t = n;
      n = t->tl
    od
  t->tl = y
{$(\exists c.\, data\_list(c,x) \wedge c = app(a,b))$}
```

In order to illustrate heap equivalence as given in Figures 2 and 5, we will focus on the associated loop invariant VC, once mutation has eliminated all $-\!\!*$ operators. Moreover, we use annotations to focus the readers attention at each stage on the relevant parts of the VC:

$$\begin{array}{r}
data\_lseg(\mathcal{X}_1,\boxed{x}^-,\boxed{t}^-) * data\_lseg(\mathcal{X}_2,\boxed{n}^-,null) * \\
data\_lseg(\mathcal{X}_b,\boxed{y}^-,null) * \boxed{(t \mapsto \mathcal{X}_3,n)}^- \wedge (x \neq n \wedge n \neq null \wedge a_0 = app(\mathcal{X}_1,[\mathcal{X}_3|\mathcal{X}_2])) \vdash \\
data\_lseg(\mathcal{F}_3,\boxed{\mathcal{F}_1}^+,null) * data\_lseg(\mathcal{F}_2,\boxed{x}^+,\boxed{n}^+) * \\
data\_lseg(\mathcal{X}_b,\boxed{y}^+,null) * \boxed{(n \mapsto \mathcal{F}_4,\mathcal{F}_1)}^+ \wedge a_0 = app(\mathcal{F}_2,[\mathcal{F}_4|\mathcal{F}_3])
\end{array} \qquad (21)$$

Proving (21) involves 3 iterations of heap equivalence, which works as follows:

Iteration 1: To achieve a **shape match**, the VCs must contain (i) the same number of heaplets and anti-heaplets, and (ii) for each heaplet there must exist a matching anti-heaplet. In (21), we have the same number of heaplets and anti-heaplets, i.e. $\boxed{(t \mapsto \mathcal{X}_3,n)}^-$ and $\boxed{(n \mapsto \mathcal{F}_4,\mathcal{X}_1)}^+$, but they fail to match because they have different head pointers.

As shown in Figure 2, the failure of **shape match** gives rise to an application of **decomposition**. The selection of the decomposition step is determined by considering all *potential* head pointer matches. In the case of (21), both the hypotheses and conclusion contain 4 head pointers, i.e. for the hypotheses there is t (heaplet), x, y, n ($\mathit{data\_lseg}$), and for the conclusion there is n (heaplet), y, $\mathcal{F}_1$, x ($\mathit{data\_lseg}$). Here only n provides a complimentary head decomposition, i.e $\boxed{(n \mapsto \mathcal{F}_4, \mathcal{F}_1)}^+$ within the conclusion will match with $\mathit{data\_lseg}(\mathcal{X}_2, \boxed{n}^-, \mathit{null})$ in the hypothesis if a head decomposition step is performed:

$$\begin{aligned}
&\mathit{data\_lseg}(\mathcal{X}_1, \boxed{x}^-, \boxed{t}^-) * \boxed{(n \mapsto \mathcal{X}_4, \mathcal{X}_5)}^- * \mathit{data\_lseg}(\mathcal{X}_6, \mathcal{X}_5, \mathit{null}) * \\
&\quad \mathit{data\_lseg}(\mathcal{X}_b, \boxed{y}^-, \mathit{null}) * \boxed{(t \mapsto \mathcal{X}_3, n)}^- \wedge (x \neq n \wedge n \neq \mathit{null} \wedge a_0 = \mathit{app}(\mathcal{X}_1, [\mathcal{X}_3, \mathcal{X}_4|\mathcal{X}_6]) \vdash \\
&\quad\quad \mathit{data\_lseg}(\mathcal{F}_3, \boxed{\mathcal{X}_5}^+, \mathit{null}) * \mathit{data\_lseg}(\mathcal{F}_2, \boxed{x}^+, \boxed{n}^+) * \\
&\quad\quad\quad \mathit{data\_lseg}(\mathcal{X}_b, \boxed{y}^+, \mathit{null}) * \boxed{(n \mapsto \mathcal{X}_4, \mathcal{X}_5)}^+ \wedge a_0 = \mathit{app}(\mathcal{F}_2, [\mathcal{X}_4|\mathcal{F}_3])
\end{aligned} \tag{22}$$

Note that as a side-effect of the decomposition, $\mathcal{X}_2$ has been replaced by $[\mathcal{X}_4|\mathcal{X}_6]$, and substitutions $\{\mathcal{X}_5/\mathcal{F}_1, \mathcal{X}_4/\mathcal{F}_4\}$ have been applied.

Iteration 2: There is a mismatch between the number of heaplets and anti-heaplets within (22), so **shape match** fails again. Applying the potential match analysis identifies 5 head pointers in the hypotheses, i.e. t, n (heaplet), x, y, $\mathcal{X}_5$ ($\mathit{data\_lseg}$), and the same 4 head pointers in the goal, i.e. n (heaplet), y, $\mathcal{F}_1$, x ($\mathit{data\_lseg}$). Here none of potential head pointer matches suggests a decomposition. However, focusing on the unmatched terms, i.e.

$$\cdots \mathit{data\_lseg}(\mathcal{X}_1, \boxed{x}^-, \boxed{t}^-) * \cdots * \boxed{(t \mapsto \mathcal{X}_3, n)}^- \cdots \vdash \cdots \mathit{data\_lseg}(\mathcal{F}_2, \boxed{x}^+, \boxed{n}^+) \cdots$$

suggests a tail decomposition, i.e.

$$\cdots \mathit{data\_lseg}(\mathcal{X}_1, \boxed{x}^-, \boxed{t}^-) * \cdots * \boxed{(t \mapsto \mathcal{X}_3, n)}^- \cdots \vdash \cdots \mathit{data\_lseg}(\mathcal{F}_6, \boxed{x}^+, \boxed{t}^+) * \boxed{(t \mapsto \mathcal{F}_5, n)}^+ \cdots$$

where $\mathcal{F}_2 = \mathit{app}(\mathcal{F}_6, [\mathcal{F}_5])$. Applying this decomposition step to (22) yields:

$$\begin{aligned}
&\mathit{data\_lseg}(\mathcal{X}_1, x, t) * (n \mapsto \mathcal{X}_4, \mathcal{X}_5) * \mathit{data\_lseg}(\mathcal{X}_6, \mathcal{X}_5, \mathit{null}) * \\
&\quad \mathit{data\_lseg}(\mathcal{X}_b, y, \mathit{null}) * (t \mapsto \mathcal{X}_3, n) \wedge (x \neq n \wedge n \neq \mathit{null} \wedge a_0 = \mathit{app}(\mathcal{X}_1, [\mathcal{X}_3, \mathcal{X}_4|\mathcal{X}_6]) \vdash \\
&\quad\quad \mathit{data\_lseg}(\mathcal{X}_6, \mathcal{X}_5, \mathit{null}) * \mathit{data\_lseg}(\mathcal{F}_6, x, \mathcal{F}_7) * (\mathcal{F}_7 \mapsto \mathcal{F}_5, n) * \\
&\quad\quad\quad \mathit{data\_lseg}(\mathcal{X}_b, y, \mathit{null}) * (n \mapsto \mathcal{X}_4, \mathcal{X}_5) \wedge a_0 = \mathit{app}(\mathcal{F}_2, [\mathcal{X}_4|\mathcal{X}_6]) \wedge \mathcal{F}_2 = \mathit{app}(\mathcal{F}_6, [\mathcal{F}_5])
\end{aligned} \tag{23}$$

Iteration 3: In (23), we have i) the same number of heaplets and anti-heaplets, and ii) the structures match, i.e. $\{\mathcal{X}_3/\mathcal{F}_5, \mathcal{X}_1/\mathcal{F}_6\, t/\mathcal{F}_7\}$.

3.3. Functional Extraction via Mutation

In the above example of heap equivalence, we need to prove the VC:

$$\begin{aligned}
&\mathit{data\_lseg}(\mathcal{X}_1, x, t) * (n \mapsto \mathcal{X}_4, \mathcal{X}_5) * \mathit{data\_lseg}(\mathcal{X}_6, \mathcal{X}_5, \mathit{null}) * \\
&\quad \mathit{data\_lseg}(\mathcal{X}_b, y, \mathit{null}) * (t \mapsto \mathcal{X}_3, n) \wedge (x \neq n \wedge n \neq \mathit{null} \wedge a_0 = \mathit{app}(\mathcal{X}_1, [\mathcal{X}_3|\mathcal{X}_4|\mathcal{X}_6]) \vdash \\
&\quad\quad \mathit{data\_lseg}(\mathcal{X}_6, \mathcal{X}_5, \mathit{null}) * \mathit{data\_lseg}(\mathcal{X}_1, x, t) * (t \mapsto \mathcal{X}_3, n) * \\
&\quad\quad\quad \mathit{data\_lseg}(\mathcal{X}_b, y, \mathit{null}) * (n \mapsto \mathcal{X}_4, \mathcal{X}_5) \wedge a_0 = \mathit{app}(\mathcal{X}_1, [\mathcal{X}_3|[]]), [\mathcal{X}_4|\mathcal{X}_6])
\end{aligned}$$

In order to extract a pure functional residue, we re-employ the mutation strategy of attraction and cancellation described in section §3.1.2 repeatedly for each of the shape elements. Since the heap equivalence algorithm has succeeded and generated a fully instantiated goal, no search is required. The functional residue in this case takes the form:

$$a_0 = \mathit{app}(\mathcal{X}_1, [\mathcal{X}_3, \mathcal{X}_4|\mathcal{X}_6]) \vdash a_0 = \mathit{app}(\mathit{app}(\mathcal{X}_1, [\mathcal{X}_3]), [\mathcal{X}_4|\mathcal{X}_6])$$

This is simplified to give:

$$\mathit{app}(\mathcal{X}_1, [\mathcal{X}_3, \mathcal{X}_4|\mathcal{X}_6]) = \mathit{app}(\mathit{app}(\mathcal{X}_1, [\mathcal{X}_3]), [\mathcal{X}_4|\mathcal{X}_6])$$

To complete the proof requires mathematical induction, which is to be expected when reasoning about inductively defined data types. As mentioned earlier, we use IsaPlanner/Isabelle to automate the proof of such inductive lemmas.

3.4. Verification of Recursive Code

When verifying recursive code, we exploit the frame rule as described in §2. Specifically our recursive procedure gives rise to a VC of the form:

$$P \vdash P' * (Q' \mathrel{-\!\!*} Q'')$$

Note that P denotes the original precondition, while P' is calculated via WP analysis applied to the precondition at the recursive call. Q' is the post-condition of the recursive call, and Q'' is calculated via WP analysis applied to the original post-condition. As noted in §1, the verification of a recursive procedure requires the introduction of a *frame invariant R*:

$$P \vdash P' * (Q' \mathrel{-\!\!*} Q' * R) \tag{24}$$

Note that R describes the parts of the heap which are not modified by the recursive call. We apply rule (8) to eliminate the Q' and and leaving a proof residue of the form:

$$P \vdash P' * R$$

Within the context of functional correctness, discovering R is in general undecidable.

In our work we perform a *frame analysis* which utilises Smallfoot [2] to provide the shape part of the frame invariant for recursive functions. Frame analysis takes the information given by Smallfoot and uses the heaplet analysis as already outlined. We give an example of such an analysis in the next section, where we also address the challenges of invariant generation, both of frame and loop invariants.

4. Invariant Synthesis Strategy

In this section we describe a strategy for automatically synthesising functional assertions which has five steps, as outlined below:

Insertion of Schematic Invariants: We initially introduce higher-order variables into the specifications, which represents unknown functional assertions for loop invariants. This is done explicitly for loop invariants, and implicitly for frame invariants in recursive code, as will be described in §4.1.

Functional Constraints: The proof strategy described in §3 generates pure formulae, which we treat as constraints on the inserted higher-order variables.

Term Synthesis: We use this technique to generate terms of the right type, with which to instantiate higher-order variables in the generated functional constraints. This is a process which generates terms incrementally, bounded by term size. This means the process is guaranteed to terminate, although it is not complete.

Testing Validity: Each term synthesised by term synthesis, is then tested using a counter-example checker using the generated functional constraints. Any terms that produce counter-examples are filtered out.

Proof of Correctness: If no counter-example is found for a particular instantiation of the schematic invariant, the ground functional constraints are sent as proof obligations to a theorem prover to test their validity.

In the remaining part of this section we will illustrate this process with our running list reversal example, and describe another mechanism in the system which can synthesise missing assertions. Note that we utilise the Smallfoot family of tools as a basis for generating schematic frame invariants.

4.1. Insertion of Schematic Invariants

To illustrate, we provide a refined version of the list reversal specification given in §1:

```
{(∃a. data_lseg(a, i, null) ∧ a0 = a)}
o = null;
while not(i = null) do
{(∃a, b. data_lseg(a, i, null) ∗ data_lseg(b, o, null) ∧ G(a0, b, a) ∧ ¬(i = null)}
   t = i->tl; i->tl = o; o = i; i = t
od
{(∃b. data_lseg(b, o, null)  ∧ a0 = rev(b))}
```

Note that a schematic loop invariant has been introduced, where $\mathcal{G}$ denotes a meta-predicate, a place-holder for the functional part of the invariant. A similar schematic frame invariant is generated for the recursive version:

```
{(∃a, o_h, o_t. data_lseg(a, i, null) ∗ (o ↦ o_h, o_t) ∧ a0 = a)}
list_reverse(s; i, o)  =
   if not(i = null) then
      k = i->tl; i->tl = o;
      {(∃a'0, a_h, a_tl, i_t. ((o ↦ o_h, o_t) ∧ H(a0, a_h, a'0)) ∗ data_lseg(a_tl, k, null) ∗ (i ↦ a_h, i_t) ∧ a'0 = a_tl))}
      list_reverse(s; k, i)
      {(∃a'0, a'_h, b_tl, i'_t. ((o ↦ o_h, o_t) ∧ H(a0, a'_h, a'0)) ∗ data_lseg(b_tl, s, i) ∗ (i ↦ a'_h, i'_t) ∧ a'0 = rev(b_tl))}
   else
      s = o
{(∃b, o'_h, o'_t. data_lseg(b, s, o) ∗ (o ↦ o'_h, o'_t) ∧ a0 = rev(b))}
```

where $\mathcal{H}$ is used as the place-holder for the functional part of the invariant.

In the case of iterative code, the following VCs are generated:

$$(\exists a.\ data\_lseg(a,i,null) \wedge a_0 = a) \vdash (\exists a,b.\ data\_lseg(a,i,null) * data\_lseg(b,null,null) \wedge \mathcal{G}(a_0,b,a)) \tag{25}$$

$$\begin{aligned}&(\exists a,b.\ data\_lseg(a,i,null) * data\_lseg(b,o,null)\ \wedge \mathcal{G}(a_0,b,a))\ \wedge \neg(i = null) \vdash \\ &\qquad (\exists f_1,f_2,f_3,f_4,a,b.\ (i \mapsto f_1,f_2) * \\ &\qquad\qquad ((i \mapsto f_1,o) \mathrel{-\!\!*} data\_lseg(a,f_4,null) * data\_lseg(b,i,null)\ \wedge (i \hookrightarrow f_3,f_4)\ \wedge \mathcal{G}(a_0,b,a)))\end{aligned} \tag{26}$$

$$\begin{aligned}&(\exists a,b.\ data\_lseg(a,i,null) * data\_lseg(b,o,null) \wedge \mathcal{G}(a_0,b,a)) \wedge i = null \vdash \\ &\qquad (\exists b.\ data\_lseg(b,o,null) \wedge a_0 = rev(b))\end{aligned} \tag{27}$$

Note that (25) corresponds to the pre-loop VC, (26) corresponds to the loop VC and (27) corresponds to the post-loop VC.

In the case of the recursive code, Smallfoot derives the shape part of the frame invariant. Using this we first perform a pre-processing frame analysis step. For the recursive list reverse, Smallfoot returns $(o \mapsto \_, o_t)$. Using this together with the heap equivalence strategy described in §3.2, frame analysis suggests expanding the list segment in the post condition $data\_lseg(b,i,o)$ by isolating the tail element. In this recursive example the following VCs are generated:

$$a_0 = [a_h|a_{tl}] \quad \vdash \quad (\exists a_h, a_{tl}.\ \mathcal{H}(a_0,a_h,a_{tl})) \tag{28}$$

$$(\exists a_h, a_{tl}.\ \mathcal{H}(a_0,a_h,a_0') \wedge a_0' = a_{tl}) \quad \vdash \quad (\exists a_h', b_{tl}.\mathcal{H}(a_0,a_h',a_0') \wedge rev(b_{tl}) = a_0') \tag{29}$$

$$(\exists a_{tl}.\mathcal{H}(a_0,a_h,a_{tl})) \wedge (a_{tl} = rev(b_{tl})) \quad \vdash \quad (\exists b.\ a_0 = rev(b) \wedge b = app(b_{tl},[a_h])) \tag{30}$$

(28) corresponds to the pre-recursive call VC, (29) corresponds to the recursive call VC and (30) corresponds to the post-recursive call VC. We want to be able automatically discover an instantiation for $\mathcal{H}$ such that all three VCs are true. In this case, the correct instantiation for $\mathcal{H}$ is

$$\mathcal{H} \equiv \lambda a_o, a_h, a_t.\ (a_0 = [a_h|a_t])$$

which gives a final *frame invariant*:

$$R \equiv (\exists o_h, o_t, a_h, a_t.\ (o \mapsto o_h, o_t) \wedge a_0 = [a_h|a_t])$$

In our work we use Smallfoot to calculate frame invariants, which helps to direct our generation of the functional parts of frame invariants. We do not exploit Smallfoot to provide the shape parts of loop invariants, instead entering the shape parts by hand. There are systems which are capable of discovering the shape parts of loop invariants, as discussed in §7, but we have not yet integrated them into our tool chain.

4.2. Functional Constraints

The mutation and heap equivalence strategies described in §3 allow functional extraction to take place in the presence of schematic variables. In the case of list reversal, we calculate the following higher-order conjectures through functional extraction:

$$\begin{array}{rcl} & \vdash & \mathcal{G}(\mathcal{X}_a, \mathcal{X}_a, null) \quad (31)\\ \mathcal{G}(\mathcal{X}_a, [\mathcal{X}_3|\mathcal{X}_2], \mathcal{X}_1) & \vdash & \mathcal{G}(\mathcal{X}_a, \mathcal{X}_2, [\mathcal{X}_3|\mathcal{X}_1]) \quad (32)\\ \mathcal{G}(\mathcal{X}_a, [], \mathcal{X}_1) & \vdash & \mathcal{X}_1 = rev(\mathcal{X}_a) \quad (33) \end{array}$$

We treat these conjectures as constraints on the schematic variable $\mathcal{G}$. Ultimately, by the end of this process, we want to generate the following instantiation:

$$\mathcal{G} \equiv \lambda x, y, z.\ (x = app(rev(z), y))$$

4.3. Term Synthesis

In order to find a correct instantiation for an inserted schematic variable, we use a generate-and-test technique, where we generate candidate terms, filter out obvious false conjecture via a counter-example checker, and attempt to prove the remaining conjectures.

4.3.1. Partial Predicate Instantiation

We must first determine predicate symbols for $\mathcal{G}$ through which we can test the validity of generated terms. We do this by incrementally instantiating $\mathcal{G}$, first choosing where to insert equalities. The heuristic we use is to find locations where variables are shared. For example in the case of the constraint shown by (32), the variable $\mathcal{X}_a$ is shared in the first position meaning an equality could be inserted to simplify the goal. We therefore reduce the synthesis problem by means of the instantiation $\mathcal{G} = \lambda x, y, z.(x = \mathcal{F}(y, z))$. This now reduces the constraints to

$$\begin{array}{rcl} \mathcal{F}([\mathcal{X}_3|\mathcal{X}_2], \mathcal{X}_1) & = & \mathcal{F}(\mathcal{X}_2, [\mathcal{X}_3|\mathcal{X}_1]) \quad (34)\\ \mathcal{X}_1 & = & rev(\mathcal{F}([], \mathcal{X}_1)) \quad (35)\\ \mathcal{X}_a & = & \mathcal{F}(\mathcal{X}_a, []) \quad (36) \end{array}$$

4.3.2. Basic Function and Predicate Definitions

In order to generate terms, and evaluate expressions using them, we must specify a set of functions over which to search, and define their semantics through definitions. We do this by equating functions defined within the VCs to functions defined in ML. For example the function *rev* in the VCs above, corresponds to the ML list function `reverse`. For lists we use the following base set of functions in order to generate terms:

$$\lambda x, y.rev(x, y) : list(int) \to list(int) \qquad \lambda x, y.[x|y] : int \times list(int) \to list(int)$$
$$\lambda x, y.app(x, y) : list(int) \times list(int) \to list(int) \qquad [] : list(int) \qquad \lambda x, y.\ x : \sigma \times \tau \to \sigma \qquad \lambda x, y.\ y : \sigma \times \tau \to \tau$$

Here the types are used so that ground instances of terms can be created as described below in §4.3.4, and σ, τ are arbitrary types. We use integer lists for ease of computational representation.

4.3.3. Generation

Firstly we determine the type of $\mathcal{F}$ by analysing its position within other functions and how it is applied. In this case, we can see that both arguments are lists and the output type is list. Moreover the type of the elements of the list are the same, and since the arguments to $\mathcal{F}$ are lists we do not include $\lambda x, y.[x|y]$ in the search.

Terms are constructed incrementally by imposing a bound on the size of the term-tree, and constructing terms of the correct type. In order to illustrate this process let us consider the constraints given by (34), (35) and (36) and the possible possible instantiations for $\mathcal{F}$ for various node sizes:

Node size 0:	$\lambda x, y.\, []$	$\lambda x, y.\, x$	$\lambda x, y.\, y$	
Node size 1:	$\lambda x, y.\, rev(x)$	$\lambda x, y.\, rev(y)$	$\lambda x, y.\, rev([])$	
Node size 2:	$\lambda x, y.\, app(x, y)$	$\lambda x, y.\, app(x, [])$	$\lambda x, y.\, app([], y)$	$\lambda x, y.\, rev(rev(x))$ $\cdots$
Node size 3:	$\lambda x, y.\, app(rev(y), x)$	$\lambda x, y.\, app(x, rev(x))$	$\lambda x, y.\, rev(app(x, y))$	$\cdots$

Once all of the terms at a certain size have been generated, they are tested using a counter-example checker as described in the next section.

4.3.4. Testing Validity

To illustrate the counter-example checker, consider a candidate instantiation, e.g. $\mathcal{F} = \lambda x, y.app(rev(x), y)$. The constraints become:

$$\begin{aligned} app(rev([\mathcal{X}_3|\mathcal{X}_2]), \mathcal{X}_1) &= app(rev(\mathcal{X}_2), [\mathcal{X}_3|\mathcal{X}_1]) \\ \mathcal{X}_1 &= rev(app(rev([]), \mathcal{X}_1)) \\ \mathcal{X}_a &= app(rev(\mathcal{X}_a), []) \end{aligned} \tag{37}$$

For each expression, we determine the type of each variable, and create candidate values by which to test the equality. The type of a variable is given by the type of the function to which it is an argument. Internally non-predicate function symbols are non-polymorphic, and we assume the type of list always to be a list of natural numbers. Thus, for example the internal representation of the list reveral function referred to here is $rev : \mathit{list\ nat} \rightarrow \mathit{list\ nat}$[3].

In the case of (37), we first determine those variables which are of integer type, i.e. $\mathcal{X}_3$, and those of type list, i.e. $\mathcal{X}_2$ and $\mathcal{X}_1$. We try 0 for $\mathcal{X}_3$ and the list $[1, 2, 3]$ for $\mathcal{X}_2$ and $[4, 5, 6]$ for $\mathcal{X}_1$. Similarly we allocate ground lists in the other equalities. We evaluate the equalities using existing ML functions for lists. The constraints above become:

$$\begin{aligned} app(rev([0|[1, 2, 3]]), [4, 5, 6]) = app(rev([1, 2, 3]), [0|[4, 5, 6]]) &\longrightarrow [3, 2, 1, 0, 4, 5, 6] = [3, 2, 1, 0, 4, 5, 6]\ ✔ \\ [1, 2, 3] = rev(app(rev([]), [1, 2, 3])) &\longrightarrow [1, 2, 3] = [3, 2, 1]\ ✗ \\ [1, 2, 3] = app(rev([1, 2, 3]), []) &\longrightarrow [1, 2, 3] = [3, 2, 1]\ ✗ \end{aligned}$$

Thus we can rule out the candidate instantiation for $\mathcal{F}$ given above as it invalidates two of the constraints.

4.3.5. Proof of Correctness

Once an instantiation for $\mathcal{F}$ without a counter-example has been found, the constraints are sent to an inductive theorem prover to attempt to find a proof. An example of a correct instantiation here is $\mathcal{F} = \lambda x, y.app(rev(y), x)$. The resulting functional constraints are provable and gives the final instantiation for $\mathcal{G}$ in (31) to be:

$$\mathcal{G} \equiv \lambda x, y, z.\ (x = app(rev(z), y)).$$

This now describes the functional part of the loop invariant, free of schematic variables.

[3]See §8.6 for a discussion of a more general technique for generating terms and counterexamples

4.4. *Synthesis of Missing Preconditions*

In addition to the generation of invariants, our technique also provides automatic guidance when precondition strengthening is required in order to complete a verification. It should be noted of course, unlike invariant generation, the programmer must ultimately decide whether or not an automatically generated precondition is acceptable.

To illustrate how the technique works, consider the following refined version of the split-pair example:

$\{(\exists a.\ \mathit{data\_lseg}(a, i, \mathit{null}) \wedge a_0 = a)\}$
```
   j = i->tl; i->tl = null;
```
$\{(\exists b, c.\ \mathit{data\_lseg}(b, i, \mathit{null}) * \mathit{data\_lseg}(c, j, \mathit{null}) \wedge \alpha_0 = \mathit{app}(b, c)\}$

Note that we have generalised the specification, and introduced an explicit functional component. The mutation strategy applies as before – however, the proof attempt breaks down. That is, given the annotated VC:

$$\mathit{data\_lseg}(\mathcal{X}_a, \boxed{i}^{-}, \mathit{null}) \vdash$$
$$\boxed{(i \mapsto \mathcal{F}_1, \mathcal{F}_2)}^{+} * (\boxed{(i \mapsto \mathcal{F}_1, \mathit{null})}^{-} -\!\!* (\mathit{data\_lseg}(\mathcal{F}_b, \boxed{i}^{+}, \mathit{null}) * \mathit{data\_lseg}(\mathcal{F}_c, \boxed{\mathcal{X}_4}^{+}, \mathit{null})) \wedge (i \hookrightarrow \mathcal{F}_3, \mathcal{F}_4) \quad (38)$$

the mutation strategy selects the unfolding of $\mathit{data\_lseg}(\mathcal{F}_b, \boxed{i}^{+}, \mathit{null})$ using (6). However, this unfolding step is only valid if we know that i points to a non-null list, which is unprovable within the given context. This failure suggests $\neg(i = \mathit{null})$ as an additional precondition, i.e.

$\{(\exists a.\ \mathit{data\_lseg}(a, i, \mathit{null}) \wedge a_0 = a) \wedge \neg(i = \mathit{null}))\}$
```
   j = i->tl; i->tl = null;
```
$\{(\exists b, c.\ \mathit{data\_lseg}(b, i, \mathit{null}) * \mathit{data\_lseg}(c, j, \mathit{null}) \wedge \alpha_0 = \mathit{app}(b, c))\}$

With the precondition strengthened, the mutation guided proof succeeds.

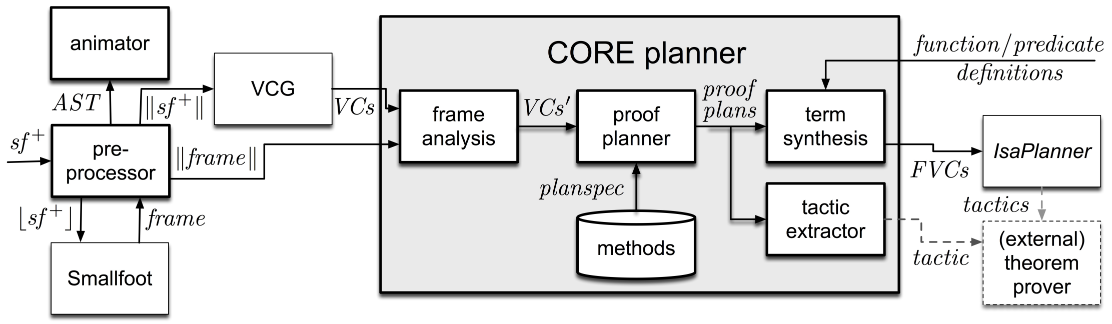
4.5. *Approaches to Term Synthesis*

A distinctive feature of proof planning is the flexibility it provides during proof search. For instance, *middle-out reasoning* [6], a technique where meta-variables are used to delay choice during the search for a proof. In middle-out reasoning proof search is used to directly guide the instantiation of meta-variables. An alternative approach is term synthesis which is described above, as implemented in the CORE Planner. Existing systems for term synthesis exist as extension to Isabelle – these are IsaCoSy [16] and IsaScheme [25].

The IsaCoSy system generates irreducible terms and is designed to generate appropriate background theories for conjectures, and has also been used to generate lemmas for certain problems. The IsaScheme system differs from IsaCoSy in that it uses the use of schemes [4] to specify the shape of a theorem. It then uses counter-example checking to eliminate false conjectures and tries to prove any remaining. We have implemented a link to the IsaScheme system, which is capable of synthesising correct invariants, but requires the shape of the expected invariant as a user-input. We cannot thus claim that this is automatic, and it is prohibitively slow on larger examples. However, the approach given in both IsaCoSy and IsaScheme is a more general notion of term synthesis and not tailored to the problem of finding loop invariants as out system is. We discuss this more in §8.6.

5. Implementation

A prototype system, called the CORE system, has been implemented which supports the reasoning technique described in the previous sections. Although the system is independent on the underlying programming language which the verification conditions are derived, we have provided support for a bytecode similar to Java bytecode — which enables the use a verification condition generator (VCG) which has been verified in the Coq theorem prover [1]. Moreover, we have implemented a translator from the syntax used by Smallfoot [2], which is an automatic tool for lightweight (separation logic) shape properties, extended with functional assertion properties. The complete tool-chain is shown in Figure 6. The rest of this section will focus on the planner, which implements the mutation and heap equivalence strategies, described in §3.1 and §3.2, and is at the heart of the system. For more details about the rest of the tool chain is found in [20].



Boxes with thin lines are external programs, and have not been developed as part of this paper. This tool accepts an extended Smallfoot program as input, which we call *sf*$^+$. $\| - \|$ is then used to translate *sf*$^+$ into bytecode, which the VCG can generate the VCs used by the CORE planner. $\lfloor - \rfloor$ is used to filter *sf*$^+$ into proper Smallfoot syntax to enable Smallfoot's frame invariant capability – while $\| - \|$ is used again to translate the resulting frame invariant into the bytecode notation used by the CORE planner. The internal abstract syntax tree (AST) is used to animate the behaviour of a given program for debugging purposes. The CORE planner produces functional VCs (*FVCs*), which are given to IsaPlanner [12] – which contains significant automation to prove these types of VCs. It also generates a tactic, which can be checked using a tactic based theorem prover which has an embedding of separation logic (although this is not yet implemented). Note that the stored methods are called plan specifications (*planspec*).

Figure 6: The tool chain of the CORE system

The mutation and heap equivalence strategies are encoded using a light weight proof planner [5]. This allows us to reason about the structure of proofs at an abstract level. The motivation for using a planner is that general patterns of reasoning, such as mutation and heap equivalences, can be encapsulated and combined to make a plan for the proof. A specification of a reasoning pattern such as mutation is called a *plan specification*, and the result of applying the planner to a plan specification with some given goals is a *proof plan*. This differs from using tactics in an automated theorem prover because we are able to separate the notions of search and soundness. A key advantage of working at such high-level is that small modification to a plan specification can be handled by a *critics* mechanism, which provides patches to the deviation by manipulating the proof plan.

The proof process will necessarily introduce search. Moreover, each (OR) branch may include many (AND) goals, thus we need to incorporate both AND and OR choices. This is encoded by a *GoalTree* in the planner, and the key components are:

$$\begin{aligned} \mathit{GoalTree} := \{&\mathit{score} :: \mathit{Int},\\ &\mathit{status} :: \{(\mathit{Success},\mathit{tactic}),(\mathit{Failure},\mathit{failure\_type}),\mathit{Empty}\},\\ &\mathit{vcs} :: [\mathit{Formula}],\\ &\mathit{children} :: [\mathit{GoalTree}], \cdots\} \end{aligned}$$

Here, *vcs* are the list of goals that has to be proven for this branch, thus describing the AND branching – while *children* contains the possible OR branches. Each child element is a different *GoalTree*, and provides a *continuation* for the branch. The *status* field indicates if the node has been processed and whether or not it succeeded, with the details described below. The *score* is used for heuristic search.

The planner receives a list of verification conditions (VCs), and from this, the following initial data-structure is created:

$$\{score = 0, status = Empty, vcs = VCs', children = [], \cdots\} \tag{39}$$

Planning is achieved by applying a *method* to a *GoalTree*, and the planner works on the full tree so the root node is always given. This is particularly important for the critic mechanism discussed below. In addition to the tree, a method takes a list of positions, which shows which nodes to work on and in which order. Moreover, the result of applying a method would be set of new children nodes to explore. The order of these depends on the search strategy used, thus the method also take a search strategy as input. The result is a new list of positions and a new goal tree. Written in a curried form, the type of a method becomes:

$$\mathit{method} :: \mathit{SearchStrategy} \rightarrow [\mathit{Pos}] \rightarrow \mathit{GoalTree} \rightarrow [\mathit{Pos}] \times \mathit{GoalTree}$$

The planner is thus abstracted over search strategies, and we have currently implemented: depth-first, heuristic depth-first, iterative deepening, breadth first and best first. The method will have a set of preconditions which has to be fulfilled before it is applied, and these are tried in the order specified by the list of positions in the tree. When they hold for a node, the first element of the *vcs* is typically selected, and from these a new set of (possible) *continuations* are computed, which becomes the *children* nodes of this node. If successful, a tactic *foo_tactic* for this step is generated, which can then be applied by a given (tactical) theorem prover. For example, this method applied to (39) will return the new goal tree:

$$\{\mathit{score} = 0, \mathit{status} = \underline{\mathit{(Success,foo\_tactic)}}, \mathit{vcs} = \mathit{VCs}', \mathit{children} = \underline{\mathit{continuations}}, \cdots\}$$

where the changes has been <u>underlined</u>. In addition, the position of (39) is removed and the positions of the *continuations* is added to the list, in an order depending on the search strategy. For example, if depth-first search was used, then they will be put in the front of this list in the same order as the children are.

The primary reason for choosing a planning approach, is that we can express the structure and reasoning patterns which are common to certain types of proof. Critics allow variations of an anticipated general structure when modifications must be made, by applying *patches* to the failures. In order to facilitate this sort of behaviour it is necessary to be able to manipulate the goal tree in such a way as to modify goals in previous nodes, including possibly the initial node. A typical example of a critic firing is when a missing precondition is detected. A goal is reached which cannot be discharged, for example a condition that a variable be non-null, as illustrated in §4.4. The planner then inserts this inequality into the original VCs as if it were written in the precondition of the code in question, and the proof is reattempted.

A critic has the following type

$$\mathit{critic} :: \mathit{failure\_type} \rightarrow \mathit{GoalTree} \rightarrow [\mathit{Pos}] \rightarrow \mathit{GoalTree} \times [\mathit{Pos}] \times \mathit{method}$$

It behaves similar to a method, however the additional *failure_type* input is used to guide the application, while it also return a method which should be applied next. However, the practical use is very different. As illustrated above a critic can change the complete goal tree. For example, it can change the initial goal of the root node as the illustration showed. A method on the other hand, typically explore only the nodes that have not been explored the before (the leaf nodes). In our proof planner, a critic is always used by the *patch_meth* methodical:

$$\mathit{patch\_meth} :: \mathit{critic} \rightarrow \mathit{method} \rightarrow \mathit{method}$$

Here, the application

$$\textbf{\textit{patch\_meth}}\ \mathit{critic\ method\ searchf\ pos\ goalt}$$

first applies the method:

$$(\mathit{pos}', \mathit{goalt}') \leftarrow \mathit{method\ searchf\ pos\ goalt}.$$

If the first position of *pos'* points to a failure node in *goalt'*, i.e. a tree node of the form $\{\cdots$, *status = (Failure, failure_type),* $\cdots\}$, then the critic is applied

$$(\mathit{pos}'', \mathit{goalt}'', \mathit{method}') \leftarrow \mathit{critic\ failure\_type\ \ pos'\ goalt'}.$$

And the returned method is applied to the resulting goal tree, meaning that the return value of ***path_meth*** becomes:

$$\mathit{method'\ searchf\ pos''\ goalt''}.$$

If the original application was not a failure node, then $(\mathit{pos}', \mathit{goalt}')$ is returned.

The main strategies in the CORE system is the *mutation* strategy followed by the *heap equivalence* strategy. As shown in Figure 2, these are applied repeatedly in a sequential order starting with *mutation*. This form a plan specification:

repeat_meth (*mutation_meth* ***then_meth*** *heap_equiv_meth*)

where **repeat_meth** and **then_meth**, are a so-called *methodicals* which are used to compose methods. Here, **repeat_meth** applies the given method until its preconditions fails, while **then_meth** sequentially composes two methods. *mutation_meth* and *heap_equiv_meth* is defined as follows:

mutation_meth is another compound method which repeatedly applies its subcomponents in sequence:

repeat_meth(*heaplet_analysis_meth* ***then_meth*** *attraction_meth* ***then_meth*** *cancellation_meth*)

- ***heaplet_analysis_meth*** is a method for analysing a heap, and choosing matching heaplets from before and after a $-\!\!*$ symbol, decomposing structure if necessary. As described for pointer heaplets, heaplet analysis is implemented via a scoring mechanism, and selects complementary heaplets which are more concretely similar – that is, require less instantiation. If the frame inferred is a heaplet $i \mapsto \_, j$ for example, and no heaplet exists with i as the pointer, then this motivates a decomposition to create a heaplet pointed to by i.

  Prior to the elimination of all $-\!\!*$ symbols, heaplet analysis considers heaplets, but in general it can also consider structures such as linked lists. Heap equivalence, as described in §3.2 ensures that these structures are existential free, and so no search is required.
- ***attraction_meth*** is a method which calculates the rewrites necessary to allow cancellation to take place.
- ***cancellation_meth*** is a method which applies rule (8), thus cancelling one instance of a $-\!\!*$ symbol.

heap_equiv_meth is also a compound method which repeatedly applies its subcomponents in sequence:

repeat_meth (*vc_selection_meth* ***then_meth*** *shape_match_meth* ***then_meth*** *decomposition_meth*)

- ***vc_selection_meth*** is a method which picks the next VC for analysis, according to the chosen search strategy.
- ***shape_match_meth*** is a method which determines if two heaps are equivalent, and returns positions where structures can be rewritten to *emp* if possible.
- ***decomposition_meth*** is a method which applies the most general decomposition possible to the heap with fewer heaplets, or in the worst case, it returns all possible decompositions of both heaps.

Heaplet analysis is a very hard problem, with a large search space. The techniques we have developed reduce the search space and are heuristically guided to search up to a given depth as described in §3. In certain circumtances, we can exploit existing technologies to reduce this search space. For example, integration with program analysis tools such as Smallfoot provides a *frame analysis*, which provides the shape of the frame invariant required, as discussed in §3.4. Although, we have argued for the generality working backwards provides, this extension also illustrates advantages by combining forwards and backwards reasoning.

6. Experimental Results

In this section we describe the experiments we have undertaken with the CORE system. Our evaluation methodology is to create a small development set of examples on which to build the system and create a test set on which to test the generality of our methods and plans. All the programs used in our experiments (see tables below) can be accessed via:

`http://www.macs.hw.ac.uk/~eahm2/core/Examples.html`

Invariant generation is applied with the functional parts of the loop invariants represented by a meta-variable as dicussed in §4. The search strategy chosen is heuristic depth first. We choose this since we believe the mutation strategy constrains the search space enough to justify not using breadth first, where proof search would be significantly slower and proofs considerably larger.

6.1. Measurements

The purpose of the experiments which are presented here is to evaluate the efficacy of the system introduced in this paper on a series of examples. We follow the evaluation methodology of introducing a development set and a test set. The programs in the developments set were used to develop the system and to refine the process by which heaplet analysis took place, in particular determining heuristics which gave a complete proof. In the test set the system was evaluated on examples which were not considered during the development of the system. The purpose of this evaluation is to determine the generality of the approach to previously unseen examples.

The system was run as an automatic "oracle" on all the examples, and where successful we record relevant information about the proof:

Program Type This indicates whether the program is straight line code, with an iterative loop, with a recursive call and whether it contains a function call

Mutation Application As mentioned in §3.1, the decomposition of a linked list structure can be via the head, tail or the middle.

Number of Branch Points The heaplet analysis heuristics as described in §3.1.1 rank choices of heaplet for which to apply attraction and cancellation. If a choice leads to a situation where a $-\!\!*$ symbol cannot be removed, backtracking takes place in the CORE planner. This leads to a branch point. The number of branch points is an indication of how successful the heuristics are at choosing the correct heaplet.

Time Taken This is a measure of how long it took to prove the specification of the given program. All tests were performed on a Toshiba TECRA A9 with 800MHz and 1GB RAM running Ubuntu 10.04.

Suggest Invariant In the case where an invariant is incompletely specified using higher-order meta variables, we record here the instantiation discovered by the CORE planner.

Proof found We record here if a proof was found of the resulting functional constraints by IsaPlanner.

6.2. Automated Verification

In the case of verification, all programs are fully annotated – the loop invariants are given – and a proof of correctness of the code with respect to the annotations is sought.

6.2.1. Development Set

We constructed a set of programs, all correct and annotated correctly for the development set. These were taken from the Smallfoot corpus, and some slightly modified versions of text book programs. All of the proofs succeeded and were completed automatically. Table 1 shows the size of each proof, and the number of branch points explored during proof search.

6.2.2. Test Set

The test set we created is composed of more examples from Smallfoot and some other created by hand. All specifications and code were complete and correct, including invariants. Results are shown in Table 2. Here, timings marked with a * indicate that interaction was required in order to complete the proof – this is explained further in §6.5. The timing in this case only refers to the automatic parts of the proof.

6.3. Automated Synthesis and Verification

In this section we follow the same methodology as used for verification, i.e. create distinct development and test sets. For synthesis the functional parts of the loop invariants and frame invariants were left as unknown higher-order meta-variables and instantiated via the term-synthesis machinery described in section §4.3. The shape parts of the invariants are obtained automatically via Smallfoot.

It is important to note that for programs which are entirely straight-line code, there is no invariant to be generated so these programs do not feature in the table of results in this section.

Program Name	Program Type	Mutation Application			Branch Points	Time taken/s
		Head	Middle	Tail		
split_list	Straight Line	✓			0	1
copylist	Recursive	✓		✓	0	5
list_reverse	Iterative	✓			0	6
list_traverse	Iterative	✓		✓	0	3
list_insert_rec	Recursive	✓	✓	✓	1	18
list_length	Iterative	✓		✓	0	4
list_append	Iterative	✓	✓	✓	0	8
list_remove	Recursive	✓	✓	✓	0	6
push	Straight Line	✓			0	3
enqueue	Function Call	✓			0	3
pop_dequeue	Straight Line	✓			0	2

Table 1: Development set verification results using the CORE system.

Program Name	Program Type	Mutation Application			Branch Points	Time taken/s
		Head	Middle	Tail		
double_list	Recursive	✓		✓	0	5
list_copy	Iterative	✓		✓	1	18
list_traverse	Recursive	✓		✓	0	4
list_deallocate_rec	Recursive	✓		✓	0	4
list_deallocate	Iterative	✓		✓	0	5
list_min	Recursive	✓		✓	0	22^*
list_append_rec	Recursive	✓	✓	✓	0	12
list_reverse_rec	Recursive	✓		✓	0	8
list_replace_last	Recursive	✓		✓	0	7
list_rotate	Iterative, Function Call	✓		✓	0	8
sortlist	Recursive, Function Call	✓	✓	✓	1	38^*
circular	Function Call	✓		✓	0	8

Table 2: Test set verification results using the CORE system.

Program Name	Program Type	Generated Invariant
copylist	Recursive	$\lambda a, a_h, a_{tl}.a = [a_h \vert a_{tl}]$
list_reverse	Iterative	$\lambda a, \alpha, \beta.a = app(rev(\beta), \alpha)$
list_traverse	Iterative	$\lambda a, \alpha, \beta.a = app(\alpha, \beta)$
list_insert_rec	Recursive	$\lambda a, \alpha, \beta.a = [a_h \vert a_{tl}]$
list_length	Iterative	$\lambda a, x, \alpha, \beta.a = app(\alpha, beta) \wedge x = length(alpha)$
list_append	Iterative	$\lambda h, a, \alpha, \beta.a = app(\alpha, [h \vert \beta])$
list_remove	Recursive	$\lambda a, \alpha, \beta.a = app(\alpha, \beta)$

Table 3: Synthesis results for the development set.

6.3.1. Development Set

Table 3 records the invariants found by the CORE system. In the case of iterative code this was the functional part of a loop invariant, and in the case of recursive code it was the functional part of the frame invariant.

6.3.2. Test Set

Table 4 records the suggested invariant, and whether this was provable in IsaPlanner.

Program Name	Program Type	Proof Found	Generated Invariant
double_list	Recursive	✔	$\lambda a, a_h, a_{tl}.\ a = [a_h \| a_{tl}]$
list_copy	Iterative	✔	$\lambda a, d1, cd, d2.\ a = app(d1, cons(cd, d2))$
list_traverse	Recursive	✔	$\lambda a, a_h, a_{tl}.\ a = [a_h \| a_{tl}]$
list_deallocate_rec	Recursive	✔	$\lambda a, a_h, a_{tl}.\ a = [a_h \| a_{tl}]$
list_deallocate	Iterative	✔	true
list_min	Recursive	✔	$(\lambda a, m, a_{tl}.\ a = [m \| a_{tl}] \wedge m < min)^*$
list_append_rec	Recursive	✔	$\lambda a, a_h, a_{tl}.\ a = [a_h \| a_{tl}]$
list_reverse_rec	Recursive	✔	$\lambda a, a_h, a_{tl}.\ a = [a_h \| a_{tl}]$
list_replace_last	Recursive	✔	$\lambda a, a_h, a_{tl}.\ a = [a_h \| a_{tl}]$
list_rotate	Iterative, Function Call	✔	$\lambda \alpha, a, c.\ \alpha = rotate(a, c)$
sortlist	Recursive, Function Call	✔	$\lambda a, a_{tl}, j_h.\ j_h = min(a) \wedge a_{tl} = delete(min(a), a)$

Table 4: Synthesis results for the test set.

Program Name	Program Type	Missing Invariant Location	Annotation Addition
split_list	Straight Line	Precondition	`i != NULL`
list_append	Iterative	Loop Invariant	`m != x`
list_rotate	Iterative, Function Call	Loop Invariant	`i != j`
pop_dequeue	Straight Line	Precondition	`r != tf`

Table 5: Programs where missing annotation was discovered.

6.4. Corrected Specifications

When attempting to discharge trivial goals, a critic is attached which suggests missing preconditions, or missing parts of an invariant. Table 5 shows the programs where this critic fired, suggesting a missing part of an annotation. Since the precondition and postcondition of a program form a contract which is a design decision, the task of verification is to show that the code fulfils this contract. In doing this constructs such as loop invariants must be discovered, as shown above, but in some circumstances missing preconditions can be "abducted". Table 5 shows the cases where a missing condition was found, and added to the precondition leading to a proof of the correctness of the program with respect to the updated specification. The updated precondition may not describe the intended specification of the program - it is up to the user to determine if this is the case. In the case where the new precondition is too strong, the code can be modified in order to attempt a new proof with a weakened precondition.

6.5. Evaluation

We have noted which types of mutation apply and how many branch points appeared in the proof. A branch point indicates that the heuristics employed by heaplet analysis indicated an incorrect decomposition, and backtracking took place to find an alternative. Branching points are typically zero, indicating that the mutation strategy performed very well. In the case where a wrong path was taken, the mutation strategy terminated due to the discovery of an ill-formed term, such as $x \overset{next}{\mapsto} x$.

All of the examples above are fully automatic except in the cases where the time taken is marked with a *. Where human interaction is required, this means that an explicit application of a decomposition is required, and an explicit instantion of a resulting existential variable is performed. The CORE planner allows reasoning via compound methods or via atomic methods, such as object-level rewriting, as described in §5. Once such low-level interaction has taken place, the planner can be reused in an automatic context by invoking the outermost plan-specification.

We see that mutation chose the wrong expansion in three proofs, and that in all proofs expansion of a linked list segment from the the head was required. In many cases expansion via the tail or from the middle was required. Mutation as a general technique worked well, but the scoring could be refined so that no branch points appear.

7. Related Work

The work builds upon previous work as described in [21, 22, 23], and the motivation for developing mutation analysis for separation logic came from ideas set out in [7]. Here we discuss the relevance to significant other work, some of which has influenced our own approach as well as our future plans.

7.1. The Smallfoot Family

Smallfoot [2] uses symbolic analysis to prove shape properties about pointer programs. The Smallfoot results provided us with a set of benchmark programs, and established a standard for other programs to follow in exploiting separation logic.

SmallfootRG [35] extends the original Smallfoot work, by inferring some loop invariants and introducing rules for dealing with the $-\!\!*$ symbol. Space Invader [10] extends the ideas of Smallfoot yet further by determining annotations for unannotated programs.

7.2. Separation Logic in Coq

[27] extends the treatment of $-\!\!*$ and classical implication by using the native types of the Coq theorem prover. It verifies the fast congruence closure algorithm of Nieuwenhuis and Oliveras, which requires a sophisticated treatment of function within separation logic. This work uses the principle of relection to allow equational reasoning about heaps. It is not clear from this work how much automation is achieved, but lemmas are included within the proofs. Unlike the work presented here, this research does not attempt to achieve automatic annotation of unknown specifications such as loop-invariants. Other work by the same author [34] extends the treatment of separation logic to concurrent algorithms, and is capable of proving thread safety properties.

[24] presents several effective proof techniques for reasoning about the heap, given the verification conditions generated in Coq of the Cminor programming language. The proof techniques presented here are exploited in an interactive proof of a garbage collector written in Cminor.

7.3. jStar

jStar [11] targets the verification of Java programs, and in particular the use of object orientation. It is designed to be highly customisable, and is implemented within eclipse, making it easy to use and a powerful verification development tool. It makes use of abstract interpretation for the generation of loop invariants. To our knowledge it does not compute fully functional invariants, and requires lemmas in order to perform the decomposition steps which we perform automatically.

7.4. The HIP System

Related to Smallfoot is the HIP system [28], which is also based upon separation logic. In addition to shape, HIP verifies properties such as the length and height of data structures. More recently, the developers of HIP have worked on verifying safety properties for recursive pointer programs that have unknown calls [18]. Techniques from abduction are employed to generate auxiliary specifications. This work was partly influenced by ideas on bi-abduction developed by the Smallfoot team [8].

7.5. HOLfoot

HOLfoot [33] formalises Smallfoot in HOL [15], and extends it with data assertions. HOLfoot is able to automatically verify the majority of the original Smallfoot corpus, along with many others. While no invariant generation is supported by HOLfoot, it does include a modified Hoare rule for loops which simplifies certain kinds of loop invariant proofs. We envisage using HOLfoot in the future as an object level proof checker for the CORE system.

7.6. C Memory Model

In [3] automatic proofs of C programs are performed by including an SMT solver and a separation logic prover. This work faithfully represents the C memory model and aims to be fully automatic. Our `circular` example is similar to queue examples they have automated. However, while we have focused on functional correctness, they have been more concerned with memory safety.

7.7. *Characteristic Formulae for Verification*

[9] introduces a calculus for computing characteristic formulae via Hoare-style triples for imperative programs. It incorporates the frame rule from Separation Logic for local reasoning by incorporating a predicate *local* in the language of the characteristic formulae. This technique has been employed to verify a number of imperative programs written in caml, and extends to verifying higher-order properties such as higher-order iterators.

7.8. *Quantifier Free Separation Logic*

In [26], a formalisation of Separation Logic is presented which removes the use of explicit existential quantification and thus is applicable to many theorem provers which use rewriting engines. This is achieved by combing function and shape into a list formalisation, and introducing predicates to assert the distinctness of the elements. This is ordinarily achieved via the semantics of the separating conjunction. The technique is successfully used to prove in-place list reversal, but some intermediate lemmas need to be introduced about the loop verification. The intermediate lemma which allows the proof to be completed using induction is equivalent to the invention of a loop-invariant in our setting, but in this work the intermediate lemma is not discovered automatically.

8. Discussion and Future Work

Below we discuss some of the issues that have arisen during the development of the CORE system. Moreover, we outline how these issues have helped to shape our plans for future work.

8.1. *Successes of Mutation*

We have observed that the mutation strategy works well for programs involving linked lists, but also we utilise it to mimic passing variables by reference in the byte code. In some Smallfoot$^+$programs, such as `list_remove`, two variables are passed in by reference. The way we solve this is to create a record with the returned values, as described in §5 where [`s0` $\overset{x}{\mapsto}$ $x1$] means that the projection of element `x` of record `s0` is x_1. Since the labels to the pointers are variables, and not labels specific to linked list manipulation, mutation automatically cancels relevant pointer cells from this procedure, without introducing any extra search into the rest of the proof.

8.2. *Interdependence of Functional and Shape*

In Smallfoot no reference is made in the specifications to the functional aspects of the program. In fact it is possible to use information from the functional parts of the specification to prove properties about shape, and vice-versa. We give three examples where this interdependence exhibits itself.

8.2.1. *Inference of Inequalities*

As an example, the program `double_list`, produces one verification condition which has goal

$$\frac{i \neq null \land data\_lseg(a, i, null)}{\cdots -\!\!* (\forall x1.\ \cdots -\!\!* (data\_lseg(a, i, null) * data\_lseg(map(\lambda x.\ double(x), a), x1, null))))}$$

here we can deduce immediately that $a \neq []$. If we can prove that

$$a \neq [] \rightarrow map(\lambda x.\ (double(x), a) \neq [])$$

then we can also assert that $x1 \neq null$ which allows us to expand the conclusion.

8.2.2. Use of Frame Inference

As mentioned above, we rely on Smallfoot to provide frame invariants. In the case of `double_list`, this is $(i \overset{next}{\mapsto} i_t)$ which directs us to expand the head of the term $data\_lseg(a, i, null)$. This in turn creates existential quantifiers which contributes to finding the functional part of a frame invariant. Recall from §3 that the shape of VCs in the context of recursion take the form:

$$P \vdash P' * (Q' \mathrel{-\!\!*} Q'')$$

where R is the frame invariant. Since we are given the shape part of R by Smallfoot, we can use mutation analysis to analyse the VC

$$R * Q' \vdash Q''$$

in order to calculate how Q'' should be expanded. This shows how information from Smallfoot and our mutation analysis achieve proofs within a recursive setting.

8.2.3. Necessity of Function

In some examples the functional part of the specification is necessary to prove a VC. For example, the program `list_remove` has precondition:

$$data\_lseg(a; l; null) \wedge mem(x, a) \wedge l \neq null$$

In this case we add the condition $l \neq null$ so that Smallfoot can prove the VC. In fact the functional part, i.e. $mem(x, a)$, ensures that a is not null, and that the variable x will be found. This implies that some of the code is redundant, i.e. it is included so as to support Smallfoot's analysis.

8.3. Precondition Generation

By exploiting our work on inserting meta-predicates, it is possible to generate valid preconditions and postconditions. This is challenging future work which would potentially reduce the burden of hand-crafting assertions.

8.4. Inductive Data Structure Decomposition

In this paper we have shown how our proof strategy can be applied to linked lists in order to extract functional content. We believe that the strategy can be generalised to other inductively defined data structures. We have already investigated some examples of binary trees, using the following definition:

$$\begin{aligned} data\_tree(t_0, X) &\leftrightarrow emp \wedge X = null \\ data\_tree(t(H, D1, D2), X) &\leftrightarrow (\exists l, r.\ (X \mapsto H, l, r) * data\_tree(D1, l) * data\_tree(D2, r) \end{aligned}$$

where the tree representation for the functional content is defined as

$$\begin{aligned} tree ::= &\mid t_0 \\ &\mid t(int, tree, tree) \end{aligned}$$

To illustrate the application of $data\_tree$, consider the following program fragment which isolates the left branch of a binary tree:

$\{data\_tree(t(1, t(2, t_0, t_0), t(3, t_0, t_0)), i)\}$

```
j = i->l; i->l = null;
```

$\{data\_tree(t(2, t_0, t_0), j) * data\_tree(t(1, t_0, t(3, t_0, t_0)), i)\}$

The above specification gives rise to the following VC:

$$\begin{aligned} &data\_tree(t(1, t(2, t_0, t_0), t(3, t_0, t_0)), \boxed{i}^{-}) \vdash \\ &\quad \boxed{(i \mapsto \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3)}^{+} * (\boxed{(i \mapsto \mathcal{F}_1, null, \mathcal{F}_3)}^{-} \mathrel{-\!\!*} \\ &\qquad data\_tree(t(2, t_0, t_0), \boxed{\mathcal{F}_5}^{+}) * data\_tree(t(1, t(3, t_0, t_0), t_0), \boxed{i}^{+}) \wedge \boxed{(i \hookrightarrow \mathcal{F}_4, \mathcal{F}_5, \mathcal{F}_6)}^{+} \end{aligned}$$

Note that mutation is applicable, and the same potential and ground matches for heaplets and associated anti-heaplets. At present the mutation strategy is defined for linked lists and binary trees, but a challenging extension would be to extend it for arbitrary inductively defined data types.

8.5. Soundness

The overall CORE system integrates many tools so soundness of the overall approach is therefore a key issue. Due to recent improvement of automatic reasoners (like SMT solvers), the issue of soundness when integrating reasoning engines has attracted much interest recently[4].

Our system uses a VCG which has been verified in Coq. Hence, we can trust that it is correct. The bytecode translation $\| - \|$ follows standard byte-code compilation, however this could be strengthen by at least a semi-formal justification of correctness. In particular, this is the case for the translation of assertions.

Smallfoot is incorporated in order to help discover frame invariants. However, no trust is given to the result, thus we do not rely on its correctness. This does not mean that we do not believe that Smallfoot is correct – however, it does mean that we do not need any formal guarantees of the shape filter $\lfloor - \rfloor$.

As we have previously discussed, the proof plans can be verified by running the generated tactic through an external theorem prover – and this is planned future work. Once completed, our trusted kernel is then reduced to the kernel of this theorem prover, the Coq system and the translator. For these reasons we argue that the abstract proof planning level provide a good abstraction for such integration, since it does reduce the trustfulness of the system.

8.6. Experiments on Term Synthesis

The performance of our approach is determined by the effectiveness of our term synthesis algorithm. Given the nature of the application domain, analysis will always be empirical. An extension to the implementation so far on term synthesis is to evaluate the different methodologies for generating lemmas. So far we have experimented with our own system and with IsaScheme. In both systems a certain tailoring of the synthesis machinery is required to achieve a result within a reasonable amount of time. In our case we supply a small finite list of function symbols and a depth-bound. Moreover the counter-example checker is a generate and test system explicitly written for the datatypes about which we expect to reason – in this case lists.

In the case of IsaScheme, a more general notion of datatype is available and the system can reason using arbitrary functions defined in Isabelle. However, the scheme based approach means that an expected shape must be given of the scheme. For example in our running example reversing a list, we must state we expect the form of the invariant to be:

$$\lambda x, y.\mathcal{F}(y, \mathcal{G}(y))$$

where $\mathcal{F}$ corresponds to that of §4.3.5. This needs to be quite prescriptive in order to yield a result in a reasonable time.

We have not experimented with integrated IsaCoSy into the system, but feel that a fruitful future avenue of research would be to perform a comparison of the efficacy of these systems for problems such as invariant generation.

8.7. Applicability to Linear Logic

Although our focus has been on separation logic, the techniques introduced have potential to be applied to other substructional logics. We will now illustrate this on small example from Linear Logic [13], which introduces multiplicative and additive forms of disjunction and conjunction in order to allow reasoning in both a classical and intuitionistic way. Typically one attributes a resource semantics to the proof rules. There is existing work on proof search in Linear Logic in [17, 32] for example.

In order to illustrate how the mutation technique applies, we introduce the following proof rules for Intuitionistic Linear Logic:

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \vdash_{\otimes} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \vdash_{\multimap} \qquad \frac{\Gamma \vdash A[\tau/\xi]}{\Gamma \vdash \exists \xi.\, A} \vdash_{\exists}$$

From these we can construct a proof of a mutation inference rule

$$\frac{\Gamma \vdash A[\tau/\xi] \quad \Delta, D \vdash B[\tau/\xi]}{\Gamma, \Delta \vdash D \multimap \exists \xi. A \otimes B} \vdash_{mut}$$

[4]E.g. see *Workshop on Trusted Extensions of Interactive Theorem Provers*, Cambridge, UK, August, 2010. `http://www.cs.utexas.edu/~kaufmann/itp-trusted-extensions-aug-2010/`

Now consider we have a goal

$$\vdash L(\beta,\gamma) \multimap P(\alpha,\beta) \multimap S(\alpha,\gamma)$$

where S, P and L are some logical structures, which are related by a set of rules including

$$\begin{aligned}
S(A,C) &\equiv \exists B,D.P(B,A)\otimes P(A,D)\otimes L(B,C) && (40)\\
S(A,C) &\equiv \exists B.L(B,A)\otimes L(B,C) && (41)\\
S(A,C) &\equiv \exists B.P(A,B)\otimes L(B,C) && (42)\\
S(A,C) &\equiv L(A,C) && (43)
\end{aligned}$$

The proof rule $\vdash_{mut}$ combines reasoning for existential reasoning with the use of the $\multimap$ symbol and multiplicative conjunction $\otimes$. In order to choose how best to manipulate the term S using a defined equivalence, some proof search is necessary. The mutation technique described in §3 provides a set of heuristics for choosing which rule to select. Following the presentation of negative and positive heaplets from §3 we annotate our goal

$$\vdash \quad L(\beta,\gamma)\multimap \boxed{P(\alpha,\beta)}^{-} \multimap S(\boxed{\alpha}^{+},\gamma) \qquad (44)$$

which decomposes according the rules above to

$$\begin{array}{ll}
(40) & \vdash \quad L(\beta,\gamma)\multimap \boxed{P(\alpha,\beta)}^{-} \multimap \exists B,D.P(B,\alpha)\otimes \boxed{P(\alpha,D)}^{+}\otimes L(B,\gamma)\\
(41) & \vdash \quad L(\beta,\gamma)\multimap \boxed{P(\alpha,\beta)}^{-} \multimap \exists B.L(B,A)\otimes L(B,C)\\
(42) & \vdash \quad L(\beta,\gamma)\multimap \boxed{P(\alpha,\beta)}^{-} \multimap \exists B.\boxed{P(\alpha,B)}^{+}\otimes L(B,\gamma)\\
(43) & \vdash \quad L(\beta,\gamma)\multimap \boxed{P(\alpha,\beta)}^{-} \multimap L(\alpha,\gamma)
\end{array}$$

In this simple case we choose the rule which allows a match between the antecedent and postcedent of $\multimap$ to exist, while introducing the fewest number of existential variables. This is rule (42). The proof is then:

$$\cfrac{\cfrac{\cfrac{L(\beta,\gamma)\vdash L([\beta/B],\gamma) \qquad P(\alpha,\beta)\vdash P(\alpha,[\beta/B])}{L(\beta,\gamma)\vdash P(\alpha,\beta)\multimap \exists B.P(\alpha,B)\otimes L(B,\gamma)}\ \vdash mut}{\vdash L(\beta,\gamma)\multimap P(\alpha,\beta)\multimap \exists B.P(\alpha,B)\otimes L(B,\gamma)}\ \vdash\!\multimap}{\vdash L(\beta,\gamma)\multimap P(\alpha,\beta)\multimap S(\alpha,\gamma)}$$

In this simple example, it can be seen that the necessity to choose which hypotheses must be used to discharge multiplicative conjuncts from the goal is key to proof in substructural logic. In order to determine which conjuncts relate to which hypotheses, one must in general decompose the conlusion or hypotheses, as shown above. The presence of existential variables makes proof search difficult, and techniques such as mutation which help to identify positive and negative terms, as described in §3 is key to automation.

9. Conclusion

Building upon separation logic, we have developed a tightly integrated tool chain for automating the functional verification of pointer programs. We have specifically targeted iterative and recursive code, providing automatic synthesis and verification of loop and frame invariants respectively. Central to our contribution are (i) the mutation and heap equivalence plan specifications which guides proof search within separation logic; (ii) a term synthesis technique which provides a uniform basis for generating loop and frame invariants and (iii) an approach which has the potential to generalise to other substructural logics thanks to the use of a weakest precondition analysis, allowing the reasoning to take place indepently of a programming language. In addition, we have shown how proof-failure analysis can be used to generate missing preconditions for a program specification. The tool chain has been tested on a range of examples which manipulate linked lists and has delivered encouraging results.

Acknowledgements

The research reported in this paper is supported by EPSRC grants EP/F037597, EP/E005713, EP/J001058 and EP/H024204. Thanks go to Bob Atkey for his help and support with the Verification Condition Generation, Lucas Dixon for help with IsaPlanner, and Alan Bundy, Andrew Cook and Maria Teresa Llano for their feedback.

References

[1] Robert Atkey. Amortised resource analysis with separation logic. In *19th European Symposium on Programming*, pages 85–103, 2010.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] Matko Botinčan, Matthew Parkinson, and Wolfram Schulte. Separation logic verification of c programs with an smt solver. *Electron. Notes Theor. Comput. Sci.*, 254:5–23, October 2009.

[4] Bruno Buchberger, Talk B. B, Mathematical Theory, and Exploration Theorem Proving. Algorithm-supported mathematical theory exploration: A personal view and strategy. In *Johannes Kepler University*, pages 236–250. Springer, 2004.

[5] 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. Longer version available from Edinburgh as DAI Research Paper No. 349.

[6] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[7] R. M. Burstall. Some techniques for proving correctness of programs. In *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland., 1972.

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009.

[9] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. (Journal version of ICFP'11) Submitted, October 2012.

[10] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[11] Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM.

[12] 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.

[13] J.Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[14] M. Gordon and H. Collavizza. Forward with hoare. *Reflections on the Work of CAR Hoare*, pages 101–121, 2010.

[15] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[16] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*. To appear.

[17] P. D. Lincoln and N. Shankar. Proof search in first-order linear logic and other cut-free sequent calculi. In *In LICS*, pages 282–291. IEEE Computer Society Press, 1994.

[18] Chenguang Luo, Florin Craciun, Shengchao Qin, Guanhua He, and Wei-Ngan Chin. Verifying pointer safety for programs with unknown calls. *Journal of Symbolic Computation: Special Issue on Invariant Generation and Advanced techniques for Reasoning about Loops*, 45(11):1163–1183, November 2010.

[19] E. Maclean and A. Ireland. Mutation in linked data structures. In *Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM-11)*, LNCS. Springer, 2011. To Appear.

[20] E. Maclean, A. Ireland, and G. Grov. The core system: Animation and functional correctness of pointer programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2011): Tool Demonstration Paper*, Lawrence, Kansas., 2011. IEEE. To Appear.

[21] Ewen Maclean, Andrew Ireland, Lucas Dixon, and Robert Atkey. Refinement and Term Synthesis in Loop Invariant Generation. In Andrew Ireland and Laura Kovacs, editors, *2nd International Workshop on Invariant Generation (WING'09)*, pages 72–86, 2009.

[22] Ewen Maclean, Andrew Ireland, and Gudmund Grov. Synthesising Functional Invariants in Separation Logic. In Nikolaj Bjorner and Laura Kovacs, editors, *3rd International Workshop on Invariant Generation (WING'09)*, pages 72–86, 2009.

[23] Ewen Maclean, Andrew Ireland, and Gudmund Grov. Functional correctness for pointer programs. In *3rd International Conference on Verified Software, Theories, Tools and Experiment: Tools and Experiments Workshop*, 2010.

[24] Andrew Mccreight. Practical tactics for separation logic. In *In TPHOLs, volume 5674 of LNCS*, pages 343–358. Springer, 2009.

[25] Omar Montano-Rivas, Roy L. McCasland, Lucas Dixon, and Alan Bundy. Scheme-based theorem discovery and concept invention. *Expert Syst. Appl.*, 39(2):1637–1646, 2012.

[26] Magnus O. Myreen. Separation logic adapted for proofs by rewriting. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 485–489. Springer, 2010.

[27] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain*, pages 261–274, 2010.

[28] H.H. Nguyen, C. David, S. Qin, and W.N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI'07*, volume 4349 of *Lecture Notes in Computer Science*. Springer, 2007.

[29] 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.

[30] D.J. Pym. Logic programming with bunched implications. *Electronic Notes in Theoretical Computer Science*, 17:1–24, 1998.

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

[32] Tanel Tammet. Proof strategies in linear logic. *Journal of Automated Reasoning*, 12(3):273–304, 1994.

[33] Thomas Tuerk. A Formalisation of Smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer, 2009.

[34] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI 2009*, pages 335–348. Springer Berlin/Heidelberg, 2009.

[35] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.