Tackling Separation via Abstraction (with proofs)

Cliff B. Jones and Nisansala Yatapanage

TECHNICAL REPORT SERIES

No. CS-TR-1447 January 2015

TECHNICAL REPORT SERIES

No. CS-TR-1447 January, 2015

Tackling Separation via Abstraction (with proofs)

C.B. Jones, N. Yatapanage

Abstract

This paper investigates the use of abstraction to specify and reason about separation in program design. Two case studies are presented: one concerns a sequential program and the other a concurrent application. The examples demonstrate that using separation as an abstraction is a potentially useful approach.

Bibliographical details

JONES, C.B., YATAPANAGE, N.

Tackling Separation via Abstraction (with proofs)
[By] C.B. Jones and N.Yatapanage

Newcastle upon Tyne: Newcastle University: Computing Science, 2015.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1447)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1447

Abstract

This paper investigates the use of abstraction to specify and reason about separation in program design. Two case studies are presented: one concerns a sequential program and the other a concurrent application. The examples demonstrate that using separation as an abstraction is a potentially useful approach.

About the authors

Nisansala Yatapanage completed her PhD in Griffith University, Australia, in 2012, on the topic of slicing of Behavior Tree specifications for model checking. This included the development of a novel form of branching bisimulation known as Next-preserving Branching Bisimulation, which has the unique property of preserving the Next temporal logic operator while still allowing stuttering steps to be removed. Nisansala has worked on research projects in software specification and verification since 2004 in both Griffith University and The University of Queensland (UQ), centering on the Behavior Tree specification language and model checking. From 2004 to 2007 she worked on the Dependability in Complex Computer-based Systems project, as part of the ARC Centre for Complex Systems, where she developed a translator from the Behavior Tree language to the input languages of model checkers, in order to automate Failure Modes and Effects Analysis. After completion of her PhD, she applied this technique to actual case studies as part of a project at UQ. Nisansala is now a Research Associate on the Taming Concurrency project at Newcastle University, UK.

Suggested keywords

CONCURRENCY
SEPARATION
SEPARATION LOGIC
RELY-GUARANTEE

Tackling Separation via Abstraction (with proofs)

Cliff B. Jones and Nisansala Yatapanage

School of Computing Science, Newcastle University, United Kingdom

Abstract. This paper investigates the use of abstraction to specify and reason about *separation* in program design. Two case studies are presented: one concerns a sequential program and the other a concurrent application. The examples demonstrate that using *separation as an abstraction* is a potentially useful approach.

Keywords: concurrency, separation, separation logic, rely-guarantee

1 Introduction

It is useful to distinguish the *issues* arising in the design of concurrent programs — two such issues are *separation* and *interference*. An obvious approach is to employ Separation Logic to tackle the first set of issues and something like Rely/Guarantee reasoning for the second. In [JHC14], the benefits of studying the issues prior to choosing an approach are discussed. In particular, that paper –and more fully [HJC14]– take a new look at specifying and reasoning about interference. The current paper attempts to offer a fresh approach to the issue of separation.

The separation of storage into disjoint portions is clearly an issue for concurrent program design — when it can be established, it is possible to reason separately about threads or processes that operate on the disjoint sections. Tony Hoare's early attempt to extend his "axiomatic basis" [Hoa69] to parallel programs provided this insight in [Hoa72]. Hoare showed that pre/post conditions of the code for separate threads could be conjoined providing the variables used by the threads are disjoint. He tackled normal (or "scoped") variables; it is more delicate to reason about "heap" variables whose addresses are computed by the programs in which they occur. Furthermore, the dynamic nature of such addresses leads naturally to the further issue of *ownership* because it is possible to write programs that effectively exchange the ownership of portions of store between threads.

The issues of separation and ownership are certainly handled well by Concurrent Separation Logic [O'H07].

The current paper suggests that some forms of separation can be specified by using data abstraction; The novelty is that the attendant obligation is to demonstrate that the separation property is preserved by the reification to a representation. Two examples are presented here: a simple list reversal algorithm that is sequential and comes from one of Reynolds' early papers [Rey02] on Separation Logic and a concurrent sorting algorithm. In both cases the representation uses heap storage.

The observation that it is possible to tackle some cases of reasoning about separation by using layers of abstraction is in no way intended to challenge research on separation logics. It might, however, give a new angle on notations for separation and/or reduce the need to develop new logics (Matt Parkinson appears to raise concerns about the proliferation of separation logics in [Par10]). Neither separation logic itself nor rely/guarantee reasoning [Jon81,Jon83a,Jon83b] are directly used although their relevance is discussed in the concluding section. Hints for a top-down development of the list reversal algorithm are sketched in [JHC14]. The current paper completes the development and fills in details omitted there — more importantly, it draws out the consequences (cf. Section 4) and adds the more substantial example of concurrent merge sorting in Section 3.

2 In-place List Reversal

As observed in [JHC14], as well as *separation* being crucial for concurrent programs, it also has a role in sequential programs. In fact, Separation Logic [Rey02] was conceived for sequential programs; Concurrent Separation Logic [O'H07] appeared later. While Section 3 applies the idea of *separation as an abstraction* to a concurrent sorting algorithm, this section shows the application of the same idea to the development of a sequential program whose final implementation performs in-place reversal of a sequence.

2.1 Original presentation

In [Rey02], John Reynolds presented an efficient sequential list reversal algorithm; the fact that the code operates in-place makes it an ideal vehicle for introducing the idea of using abstraction to handle separation. Interestingly, Reynolds introduced the problem by starting with the algorithm, shown in Figure 1. The list is represented by a value for each item, with the subsequent address containing a pointer to the next item. The algorithm utilises three pointers (`i, j, k`) where initially `i` points to the start of the list and `j` is set to null. At each step, the next pointer of item `i`, given by `*(i+1)`, is redirected to point to the location of the previous item, held by `j`. Next, the `j` pointer is updated to `i`. The `i` pointer is then moved to point to the original next item, now held by the temporary place-holder `k`. The process continues in this manner until the full list is reversed. Reynolds uses the separating conjunction of Separation Logic to develop a useful specification of the algorithm from the code.

```
j = null;
while (i != null) {
  k = *(i+1);
  *(i+1) = j;
  j = i;
  i = k;
}
```

Fig. 1. Reynolds' in-place list reversal program in C notation: `*k` is C-style pointer dereference of pointer `k`.

2.2 Abstract specification

The notion of reversing a sequence is easily expressed as a recursive function:[1]

$$\mathit{rev} : \mathit{Val}^* \to \mathit{Val}^*$$
$$\mathit{rev}(\mathit{list}) \quad \triangleq \quad \textbf{if } \mathit{list} = [\,] \textbf{ then } \mathit{list} \textbf{ else } \mathit{rev}(\textbf{tl}\, \mathit{list}) \curvearrowright [\textbf{hd}\, \mathit{list}]$$

The intention is to develop a program (using *rev* in the specification); the first thing is to note that the state of the program is a pair of lists:

$$\Sigma_a = (\mathit{Val}^* \times \mathit{Val}^*)$$

where the first, referred to as s, is the original list and the second, referred to as r, should finally contain the reversed list. It is worth observing that the two fields of Σ_a are implicitly separate.

An "operation" to compute the reverse of a list can be specified as follows:

$$\mathit{post\text{-}REVERSE}_a((s,r),(s',r')) \triangleq r' = \mathit{rev}(s)$$

It is easy to develop the abstract program in Figure 2. The body of the while loop is again given as a specified operation because its isolation makes the reification below clearer. The loop preserves the value of $\mathit{rev}(s) \curvearrowright r$; the standard VDM proof rule for loops handles termination by requiring that the relation be well-founded — thus $\mathit{rev}(s') \curvearrowright r' = \mathit{rev}(s) \curvearrowright r \land \textbf{len}\, s' < \textbf{len}\, s$.

2.3 Representing sequences

The program in Figure 2 is based on abstract states which cannot require the sorting to be performed in place (recursive implementations utilising temporary storage space would also satisfy the specification). To show how the list reversal can occur without moving the data, the abstract state is reified to a heap representation:

$$\mathit{Heap} = \mathit{Ptr} \xrightarrow{m} (\mathit{Val} \times [\mathit{Ptr}])$$

[1] Well-established VDM notation is used throughout the current paper; see [Jon90] for details.

```
$r \leftarrow [\,]$;
**while** $s \neq [\,]$ **do**
  $STEP_a$
**end while**
```

$$pre\text{-}STEP_a((r,s)) \triangleq s \neq [\,]$$
$$post\text{-}STEP_a((r,s),(r',s')) \triangleq r' = [\mathbf{hd}\, s] \frown r \land s' = \mathbf{tl}\, s$$

Fig. 2. Abstract list reversal program.

Maps in VDM ($D \xrightarrow{m} R$) are finite, constructed functions; the fields of a pair $pr \in (Val \times [Ptr])$ are accessed by index e.g. pr_1; the square brackets around $[Ptr]$ indicate that **nil** is a possible value.

In general, such a heap might contain information for other threads and/or garbage discarded by processes. The specification is most concisely expressed on a portion of the heap currently being used by *REVERSE*. The notion of a sequence representation (*Srep*) is a sub-heap containing a representation of one sequence.[2] Restricting types by predicates is useful and the invariant of *Srep* ensures that the sub-heap contains a pointer to the start of a list representation and that there are no loops present.

$$Srep = Heap$$

where

$$inv\text{-}Srep : Heap \to \mathbb{B}$$

$$\begin{aligned} &inv\text{-}Srep(hp) \quad \triangleq \\ &\quad hp = \{\,\} \lor \\ &\quad \exists b \in \mathbf{dom}\, hp \cdot is\text{-}start(hp, b) \land \\ &\quad \forall p, q \in \mathbf{dom}\, hp \cdot p \neq q \;\Rightarrow\; hp(p)_2 \neq hp(q)_2 \end{aligned}$$

The invariant for *Srep* uses a function that checks whether the given pointer is pointing to the start of the sequence representation by checking that b is the only pointer not contained in the second elements of any pair in the *Srep*:

$$is\text{-}start : Heap \times Ptr \to \mathbb{B}$$

$$is\text{-}start(hp, b) \quad \triangleq \quad \{hp(p)_2 \mid p \in \mathbf{dom}\, hp\} = (\mathbf{dom}\, hp - \{b\}) \cup \{\mathbf{nil}\}$$

Lemma 1. *The initial element in an Srep object is always unique:*

$$\forall sr \in Srep \cdot \forall c, d \in \mathbf{dom}\, sr \cdot is\text{-}start(sr, c) \land is\text{-}start(sr, d) \;\Rightarrow\; c = d$$

[2] The *Srep* concept is worth separating because it is useful both for this simple example and for the development of the concurrent program in Section 3.

The proof follows trivially from the definitions.[3]

This justifies the use of the *iota* operator, requiring the unique existence of a value, in the following definition of the function *start* that returns the initial element of an *Srep*:

$$start : Srep \to [Ptr]$$

$$start(sr) \quad \triangleq \quad \textbf{if } sr = \{\,\} \textbf{ then nil else } \iota\, b \in \textbf{dom}\, sr \cdot is\text{-}start(sr, b)$$

The state for this development step contains two separate *Srep* objects:

$$\Sigma_b = (Srep \times Srep)$$

where

$$inv\text{-}\Sigma_b((s, r)) \quad \triangleq \quad sep(s, r)$$

$$sep : Srep \times Srep \to \mathbb{B}$$

$$sep(s, r) \quad \triangleq \quad \textbf{dom}\, s \cap \textbf{dom}\, r = \{\,\}$$

On this representation, the specification of the operation corresponding to the body of the while loop in Figure 2 is:

$$pre\text{-}STEP_b(s, r) \triangleq s \neq \{\,\}$$
$$post\text{-}STEP_b((s, r), (s', r')) \triangleq$$
$$s' = \{start(s)\} \mathbin{\lhd\!\!\!-} s \wedge r' = r \cup \{start(s) \mapsto (s(start(s))_1, start(r))\}$$

It is necessary to show that $STEP_b$ preserves the invariant of Σ_b.

Lemma 2. $(s, r) \in \Sigma_b \wedge post\text{-}STEP_b((s, r), (s', r')) \;\Rightarrow\; (s', r') \in \Sigma_b$

The proof follows easily from the definitions.

Developing programs by data reification is standard in VDM (cf. [Jon90, Chap. 8]); here, the novel aspect is the need to show that the chosen representation preserves the separation of the representations of the abstract variables. As can be seen from the following retrieve function, each of the *Srep* components in Σ_b represents a sequence in Σ_a.

$$retr\text{-}a : \Sigma_b \to \Sigma_a$$

$$retr\text{-}a((s, r)) \quad \triangleq \quad (gather(s), gather(r))$$

[3] The conference version of this paper omits all detailed proofs which are mostly routine — they can be found in the Technical Report [**?**, App.].

which uses a function that collects the values in a given *Srep* into a list:

$$gather : Srep \to Val^*$$

```
$gather(sr) \;\triangleq\;$ if $sr = \{\,\}$
                 then $[\,]$
                 else let $b = start(sr)$ in
                        $[sr(b)_1] \curvearrowright gather(\{b\} \lhd sr)$
```

Retrieve functions are homomorphisms from the representation back to the abstraction. VDM defines an "adequacy" proof obligation which requires that, for each abstract state, there exists at least one representation state.

Lemma 3. *There is at least one representation for each abstract state:*

$$\forall s \in Val^* \cdot \exists sr \in Srep \cdot gather(sr) = s$$

The proof of this lemma is by induction on s.

The key commutativity proof for reification shows that the design step models the abstract specification:

Lemma 4. *$STEP_b$ models (under retr-a) the abstract $STEP_a$*

$$inv\text{-}\Sigma_b(\sigma_b) \wedge post\text{-}STEP_b(\sigma_b, \sigma'_b) \;\Rightarrow\; post\text{-}STEP_a(retr\text{-}a(\sigma_b), retr\text{-}a(\sigma'_b))$$

The proof follows from unfolding the defined functions/predicates.

2.4 Using Pointers

The final representation uses a heap (*hp*) and two pointers (i, j). The *hp* field of Σ_c is essentially the heap envisaged in Figure 1.[4] In this case, rather than a retrieve function, it is easier to define a relation between Σ_b and Σ_c

$$rel\text{-}b\text{-}c : \Sigma_b \times \Sigma_c \to \mathbb{B}$$

$$rel\text{-}b\text{-}c((s, r), (hp, i, j))) \;\triangleq\; hp = r \cup s \wedge i = start(s) \wedge j = start(r)$$

On Σ_c, the specification of the operation corresponding to $STEP_b$ above is:

$$post\text{-}STEP_c((hp, i, j), (hp', i', j')) \;\triangleq\; i' = hp(i)_2 \wedge j' = i \wedge hp' = hp \dagger \{i \mapsto (hp(i)_1, j)\}$$

The reification proof obligation in the case of a relation between the abstraction and representation is given in [Jon90, Sect. 9.3]

Lemma 5. *$STEP_c$ models $STEP_b$*

$$rel\text{-}b\text{-}c(\sigma_b, \sigma_c) \wedge post\text{-}STEP_c(\sigma_c, \sigma'_c) \;\Rightarrow\; \exists \sigma'_b \in \Sigma_b \cdot post\text{-}STEP_b(\sigma_b, \sigma'_b) \wedge rel\text{-}b\text{-}c(\sigma'_b, \sigma'_c)$$

[4] The fact that "cells" contain both data and pointer (rather than them being in locations n and $n + 1$ as in Figure 1) is incidental — think of *car*/*cdr* in Lisp. Furthermore, the decision to use *Ptr* rather than $\mathbb{N}$ is deliberate.

The proof follows from the definitions with the existential introduction using the one point rule.

The attentive reader might have noted that no invariant was given for Σ_c; this is most easily expressed as:

$$\mathit{inv}\text{-}\Sigma_c(\sigma_c) \quad \triangleq \quad \exists s, r \in \mathit{Srep} \cdot \mathit{sep}(s, r) \wedge \mathit{rel}\text{-}b\text{-}c((s, r), \sigma_c)$$

Code (in C++) that satisfies $\mathit{post}\text{-}\mathit{STEP}_c$ is given in Figure 3. The final step in the correctness argument is to note that Figure 2 terminates when $s = [\,]$, modelled on the representation by terminating when $i = \mathbf{nil}$, which, under *rel-b-c* and *retr-a*, are equivalent.

```
Class Pair{
    Val v;
    Pair* p;
}
Pair* reverse(Pair* i){
  Pair* k;
  Pair* j = NULL;
  while (i != NULL) {
    k = i->p;
    i->p = j;
    j = i;
    i = k;
  }
  return j;
}
```

Fig. 3. C++ implementation of the list reversal algorithm.

2.5 Observations

This simple sequential example illustrates how the motto *separation is an abstraction* can work in practice. In the abstraction (Σ_a) of Section 2.2, the two variables are assumed to be distinct; standard data reification rules apply where that distinction is obvious; here, it must be established that the abstraction of separation holds in the representation as (changing) portions of a shared heap. A valuable byproduct of the layered design is that the algorithm is discussed on the abstraction and neither the reification step nor its justification are concerned with list reversal as such. This is, of course, in line with the message of [Wir76].

There are some incidental bonuses from the use of VDM: the invariant (and the use of predicate restricted types) effectively provides pre conditions on the functions; use of relational post conditions avoids the need for what are essentially auxiliary variables to refer to the initial state; and the use of "LPF" [BCJ84] simplifies the construction of logical expressions where terms and/or propositions can fail to denote.

There is no need to use rely/guarantee conditions here because there is no concurrency but they could be used to demarcate the portion of the heap that is of concern. Notice that $hp = r \cup s$ could be relaxed to $r \cup s \subseteq hp$. This topic is reviewed in the next section where parallelism is an issue.

3 Mergesort

The list reversal example demonstrates the idea of handling *separation via abstraction* in a sequential development. This section applies the same idea to a concurrent design: the well-known *mergesort* algorithm which sorts by recursively splitting lists. At each stage, the argument list is divided into two parts (preferably, but not necessarily, of roughly equal sizes) which are then recursively submitted to merge sort. As the recursion unwinds, the two sorted lists are merged into a single sorted list.

3.1 Specification

The notion of sorting is easy to specify as a relation:

$$is\text{-}sort : Val^* \times Val^* \to \mathbb{B}$$

$$is\text{-}sort(s, s') \quad \triangleq \quad ordered(s') \wedge permutes(s', s)$$

The *ordered* predicate tests that the elements of its argument are in ascending order.

$$ordered : Val^* \to \mathbb{B}$$

$$ordered(s) \quad \triangleq \quad \forall i \in \{1..\mathbf{len}\, s - 1\} \cdot s(i) \leq s(i+1)$$

The *permutes* predicate tests that its two argument lists contain the same elements; here this is done by comparing the "bag" ("multiset") of occurrences:

$$permutes : Val^* \times Val^* \to \mathbb{B}$$

$$permutes(s, s') \quad \triangleq \quad bag\text{-}of(s') = bag\text{-}of(s)$$

$$bag\text{-}of : Val^* \to (Val \xrightarrow{m} \mathbb{N}_1)$$

$$bag\text{-}of(s) \quad \triangleq \quad \{e \mapsto \mathbf{card}\, \{i \in \mathbf{inds}\, s \mid s(i) = e\} \mid e \in \mathbf{elems}\, s\}$$

3.2 Algorithm

The basic idea of merge sorting can be established with a recursive function (*mergesort* defined below). This uses a *merge* function that joins the two given lists by comparing their head elements and inserting the smaller element into the result list:

```
$merge : Val^* \times Val^* \to Val^*$

$merge(s1, s2) \quad \triangleq$
    $\mathbf{if}\ s1 = [\,]$
    $\mathbf{then}\ s2$
    $\mathbf{else\ if}\ s2 = [\,]$
        $\mathbf{then}\ s1$
        $\mathbf{else\ if}\ (\mathbf{hd}\, s1 \le \mathbf{hd}\, s2)$
            $\mathbf{then}\ [\mathbf{hd}\, s1] \curvearrowright merge(\mathbf{tl}\, s1, s2)$
            $\mathbf{else}\ [\mathbf{hd}\, s2] \curvearrowright merge(s1, \mathbf{tl}\, s2)$
```

Lemma 6. *The merge function has the property that the final list is a permutation of the initial two lists conjoined:*

$$permutes(merge(s1, s2), s1 \curvearrowright s2)$$

The proof is by nested induction on the lists.

Lemma 7. *The merge function also satisfies the property that, if the argument lists are ordered, so is the resulting merged list:*

$$ordered(s1) \wedge ordered(s2) \;\Rightarrow\; ordered(merge(s1, s2))$$

The proof is identical in structure to that of Lemma 6.

The *mergesort* function itself is defined as follows:

```
$mergesort : Val^* \to Val^*$

$mergesort(s) \quad \triangleq$
    $\mathbf{if\ len}\, s \le 1$
    $\mathbf{then}\ s$
    $\mathbf{else\ let}\ s1, s2\ \mathbf{be\ st}\ s1 \curvearrowright s2 = s \wedge s1 \ne [\,] \wedge s2 \ne [\,]\ \mathbf{in}$
        $merge(mergesort(s1), mergesort(s2))$
```

Lemma 8. *The mergesort function ensures that the resulting list is both sorted and a permutation of the initial list:*

$$s' = mergesort(s) \;\Rightarrow\; is\text{-}sort(s, s')$$

The proof needs course-of-values induction on s.

3.3 Representing sequences

Having established the overall algorithmic ideas in Section 3.2, the method used in Section 2.3 can be followed by reifying the sequences into representations in heap storage. The type *Srep* is exactly as in Section 2.3. The implementation consists of two operations: *MERGE* and *MSORT*. *MSORT* operates on Σ_b (cf. Section 2.3), while the *MERGE* operation uses a state that contains three instances of *Srep*:

$$\Sigma_m = (Srep \times Srep \times Srep),$$

where the three fields are pairwise separate (*sep*).

In Section 2.2, a while loop was used for the (abstract) program. This approach is not followed here because it would be a digression to derive a proof rule for the (non-tail) recursion needed in *MSORT* (this construct is not covered in [Jon90]). Instead the recursion in both *MERGE* and *MERGESORT* is represented by "quoting post conditions" (cf. [Jon90, Section 9.3]).

$$\begin{array}{l}
post\text{-}MERGE((l, r, a), (l', r', a')) \ \underline{\triangle} \\
\quad l = \{\,\} \land a' = r \land l' = r' = \{\,\} \lor \\
\quad r = \{\,\} \land a' = l \land l' = r' = \{\,\} \lor \\
\quad l \neq \{\,\} \land r \neq \{\,\} \land l(start(l))_1 \leq r(start(r))_1 \land \\
\qquad post\text{-}MERGE((\{start(l)\} \lhd l, r, a), (l', r', ma)) \land \\
\qquad a' = \{start(l) \mapsto (l(start(l))_1, start((ma))\} \cup ma \lor \\
\quad l \neq \{\,\} \land r \neq \{\,\} \land l(start(l))_1 > r(start(r))_1 \land \\
\qquad post\text{-}MERGE((l, \{start(r)\} \lhd r, a), (l', r', ma)) \land \\
\qquad a' = \{start(r) \mapsto (r(start(r))_1, start((ma))\} \cup ma
\end{array}$$

Lemma 9. *MERGE preserves separation:*

$$(l, r, a) \in \Sigma_m \land post\text{-}MERGE((l, r, a), (l', r', a')) \ \Rightarrow \ (l', r', a') \in \Sigma_m$$

The proof of this lemma is obvious from the form of the proof of Lemma 2.

Lemma 10. *MERGE mirrors merge*

$$\begin{array}{l}
\forall l, r, a, l', r', a' \in Val^* \cdot \\
\quad post\text{-}MERGE((l, r, a), (l', r', a')) \ \Rightarrow \\
\qquad\qquad\qquad\qquad gather(a') = merge(gather(l), gather(r))
\end{array}$$

Here again, the proof follows that of Lemma 4.

It is necessary below to split an *Srep* into two separate values of that type. Since the result must represent two non-empty sequences, the argument *Srep* must represent a sequence whose length is at least two. The function *split* recurses until the argument p is located in the representation:

```
$split : Srep \times Ptr \to (Srep \times Srep)$

$split(sr, p) \quad \triangleq$
    $\mathbf{if}\ sr(start(sr))_2 = p$
    $\mathbf{then}\ (\{start(sr) \mapsto (sr(start(sr))_1, \mathbf{nil})\}, \{start(sr)\} \ntriangleleft sr)$
    $\mathbf{else\ let}\ (l, r) = split(\{start(sr)\} \ntriangleleft sr, p)\ \mathbf{in}$
        $((\{start(sr)\} \triangleleft sr) \cup l, r)$

$\mathbf{pre}\ p \in \mathbf{dom}\, sr \wedge p \neq start(sr)$
```

Lemma 11. *The split function yields two instances of Srep that are separate:*

$$p \in \mathbf{dom}\, sr \wedge p \neq start(sr) \wedge (l, r) = split(sr, p) \ \Rightarrow\ l, r \in Srep \wedge sep(l, r)$$

The proof is by induction on sr.

Lemma 12. *Under the gather function, concatenation of the two lists produced by split gives the argument list:*

$$p \in \mathbf{dom}\, sr \wedge p \neq start(sr) \wedge (l, r) = split(sr, p) \ \Rightarrow \\ gather(l) \frown gather(r) = gather(sr)$$

This proof follows the structure of that of Lemma 11.

The state for *MSORT* is precisely a (single) *Srep*:

$\Sigma_n = Srep$

Whereas *MERGE* is used sequentially (there are no concurrent threads), instances of *MSORT* can be run in parallel. The term "parallel" is used in preference to "concurrently" precisely because the instances are executed on separate parts of the heap.[5]

```
$MSORT$
$\mathbf{ext\ wr}\ \sigma\ :\ Srep$
$\mathbf{post}\ (\sigma = \{\,\} \vee \sigma(start(\sigma))_2 = \mathbf{nil}) \wedge \sigma' = \sigma\ \vee$
    $\mathbf{let}\ p \in \mathbf{dom}\, \sigma\ \mathbf{be\ st}\ p \neq start(\sigma)\ \mathbf{in}$
    $\mathbf{let}\ (l, r) = split(\sigma, p)\ \mathbf{in}$
    $post\text{-}MSORT(l, l') \wedge$
    $post\text{-}MSORT(r, r') \wedge$
    $post\text{-}MERGE((l', r', \{\,\}), (, , \sigma'))$
```

The final conclusion is that *MSORT* mirrors *mergesort*:

$$post\text{-}MSORT(\sigma, \sigma') \ \Rightarrow\ gather(\sigma') = mergesort(gather(\sigma))$$

which follows immediately from the lemmas.

[5] The alternative of using a simple form of rely/guarantee condition is discussed in Section 4.

3.4 Using pointers

As in Section 2.4, it is straightforward to develop code as in Figures 4 and 5 that satisfy the specifications of *MERGE* and *MSORT*; Section 4 raises the more interesting possibility of mechanically generating such code.

```
Class Pair{
    Val v;
    Pair* ptr;
}
Pair* merge(Pair* l, Pair* r){
  Pair* result;
  if (l == NULL){
    return r;
  }else if (r == NULL){
    return l;
  }else if (l->v <= r->v){
    result = merge(l->ptr, r);
    l->ptr = result;
    return l;
  }else{
    result = merge(l, r->ptr);
    r->ptr = result;
    return r;
  }
}
```

Fig. 4. C++ implementation of MERGE.

3.5 Observations

As in Section 2, the approach of viewing separation as an abstraction has benefits. As in the earlier example, aspects of VDM such as types restricted by predicates and relational post conditions play a small part in the development of merge sort. More significant is that the layered development makes it possible to divorce the reasoning about merging and sorting from details of how the abstract state is reified onto heap storage.

Although this example has used some aspects of VDM not needed in Section 2 — in particular, quoting post conditions — it is important to remember that these are long-standing ideas in VDM and are not specific to reasoning about the separation issue.

```
Pair* split(Pair* p){
  int midlen = getlength(p) / 2;
  int counter = 1;
  Pair* current = p;
  while (counter < midlen){
    current = current->ptr;
    counter++;
  }
  Pair* next = current->ptr;
  current->ptr = NULL;
  return next;
}

Pair* msort(Pair* p){
  if (p == NULL || p->ptr == NULL){
    return p;
  }
  Pair* mid = split(p);
  Pair* sortedp = msort(p);
  Pair* sortedmid = msort(mid);
  return merge(sortedp, sortedmid);
}
```

Fig. 5. C++ implementation of MSORT.

4 Discussion

The research reported in this paper is one vector of a project known as "Taming Concurrency" in which it is hoped to identify and/or develop apposite notations for reasoning about the underlying *issues* that make designing and reasoning about intricate concurrent programs challenging. In contrast, starting with a fixed notation might be seen as a version of "to a man with a hammer, everything looks like a nail".

The Rely/Guarantee (R/G) approach (of which more below) was devised to reason about the issue of *interference*. The R/G concept has been substantially recast in [HJC14] and summarised in [JHC14]. In contrast to the monolithic five-tuple approach of [Jon81,Jon83a,Jon83b] for R/G specifications, [HJC14] presents separate **rely** and **guar** constructs in a refinement calculus style and indicates that these constructs have an algebraic structure.

The current paper is written in the same spirit. *Separation* is also a key issue in thinking about parallel programs. One example of the importance of separation is the way in which storage is allocated between threads in an operating system. Separation Logic (SL) has a well-crafted collection of operators for reasoning about separation and an attractive feature is the pleasing algebraic properties of the operators.

This paper –with the help of examples previously tackled with SL– explores the option of reasoning about separation using predicates defined over heaps. The idea can be summarised with the motto that *separation is an abstraction.* A corollary of this point of view is that reifications (e.g. of separate variables into heap representations) have to preserve the separation property of the abstraction. Other than the twist of viewing separation as an abstraction, the method of data reification used here is long-established in the literature.

Analogous to the pulling apart of R/G specifications, an alternative view of SL might lead to different notational ideas than if the notation itself is taken as the fixed point.

Obviously, the fact that it is possible to reason about separation without the need to use SL itself, is not an argument against SL. One huge benefit of SL is the tool support that has been developed around the notation. These tools support a "bottom-up" approach that is advantageous with legacy software. The pleasing algebraic relationship between SL operators has been referred to above. These operators are also able to express some constraints in a succinct way (e.g. the use of separating conjunction with recursion to state that a chain of pointers has no loops).

A bonus from the top down approach can be seen in both of the examples above: the essence of each algorithm is documented and reasoned about on the abstraction and this is separated from arguments about the messy details of the (heap) representations. The hope is that seeing what can be done via a top-down view using abstraction could prompt new requirements for SL-like notations. The approach might, conceivably, also control the proliferation against which Matt Parkinson warns in [Par10].

To anyone familiar with the first author's work on R/G, the limited role of that approach in the current paper might come as a surprise. Of course, the principal cause is that this paper addresses the issue of *separation* whereas R/G was developed to tackle *interference.* It would, however, be possible to use R/G conditions for the specification of *MSORT* in Section 3.3 to show that there is really only one heap and that interference is limited to appropriate sub-heaps (e.g.
$\mathit{rely\text{-}MSORT}\colon \mathit{ptrs}(p, \mathit{hp}) \lhd \mathit{hp}' = \mathit{ptrs}(p, \mathit{hp}) \lhd \mathit{hp}$
$\mathit{guar\text{-}MSORT}\colon \mathit{ptrs}(p, \mathit{hp}) ⩤ \mathit{hp}' = \mathit{ptrs}(p, \mathit{hp}) ⩤ \mathit{hp}$
it would be worth defining a *ptrs* function but without that it is the domain of the corresponding *Srep*).

The preceding comment prompts some observations about alternative developments considered by the authors.

- It would simplify the notation to separate the *Heap* into two mappings (one for the *Val* and the other for the next *Ptr*) because it would remove the need to use subscripts to access the components of the pair.
- In both applications, it would be possible to omit the intermediate (Σ_b) representation and to jump directly from the respective abstract states to the general *Heap*. As mentioned in Section 2.3, the fact that *Srep* was used in both problems is one argument for separating it as an intermediate notion

— the other argument is the Wirth-like divorce of the algorithm design from the messy heap representation details.
– It would be useful to develop a "theory" of *Srep* objects and this might be the most telling testbed for choosing the role of separating conjunction.

Another avenue that would be interesting to explore is the extent to which recording the relationship between a clean abstraction and its representation (given here as "retrieve functions") could be used to generate code automatically from the abstract algorithm.

Related to the connections between R/G and SL (cf. [**?**,**?**]), it might be worth noting one of the Laws in [HJC14]:

$$[q_1 \wedge q_2] \sqsubseteq (\mathbf{guar}\ g_1 \bullet (\mathbf{rely}\ g_2 \bullet [q_1])) \parallel (\mathbf{guar}\ g_2 \bullet (\mathbf{rely}\ g_1 \bullet [q_2]))$$

which both handles the general case of interference and rather clearly shows that the attractive prospect of conjoining the post conditions of parallel threads can be achieved (only) if their respective guarantee conditions ensure sufficient separation. This makes clear that complete separation is just an extreme case of minimising interference.

For the future, the need to reason about both separation and interference will be discussed in another paper on which the current authors are working (together with Andrius Velykis) which covers the design of a concurrent implementation of DOM .

Acknowledgements

The research reported here is supported by (UK) EPSRC "Taming Concurrency" and "TrAmS-2" research grants. The authors would like to thank Andrius Velykis and our colleagues Ian Hayes, Larissa Meinicke and Kim Solin from the (Australian) ARC-funded project "Understanding concurrent programmes using rely-guarantee thinking" for their invaluable feedback.

References

BCJ84. H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, 1984.
HJC14. Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, July 2014.
Hoa69. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
Hoa71. C.A.R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.
Hoa72. C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.
JHC14. Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, (on-line), 2014.

Jon81. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

Jon83a. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

Jon83b. C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

Jon90. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

O'H07. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.

Par10. Matthew Parkinson. The next 700 separation logics. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.

Rey02. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, pages 55–74. IEEE, 2002.

Wir76. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

Appendix

This appendix (of the Technical Report version) contains detailed proofs of some of the lemmas in the submitted version of the paper. The proofs are laid out in a natural deduction style whose VDM use is described in [Jon90, Sect.1.2].

Lemma 1 states:

$$\forall sr \in \mathit{Srep} \cdot \forall c, d \in \mathbf{dom}\, sr \cdot \mathit{is\text{-}start}(sr, c) \wedge \mathit{is\text{-}start}(sr, d) \;\Rightarrow\; c = d$$

The proof follows simply by expanding the definitions.

	from $sr \in \mathit{Srep}; c, d \in \mathbf{dom}\, sr$	
1	**from** $\mathit{is\text{-}start}(sr, c) \wedge \mathit{is\text{-}start}(sr, d)$	
1.1	$\{sr(p)_2 \mid p \in \mathbf{dom}\, sr\} = (\mathbf{dom}\, sr - \{c\}) \cup \{\mathbf{nil}\}$	h1, *is-start*
1.2	$\{sr(p)_2 \mid p \in \mathbf{dom}\, sr\} = (\mathbf{dom}\, sr - \{d\}) \cup \{\mathbf{nil}\}$	h1, *is-start*
1.3	$(\mathbf{dom}\, sr - \{c\}) \cup \{\mathbf{nil}\} = (\mathbf{dom}\, sr - \{d\}) \cup \{\mathbf{nil}\}$	1.1, 1.2
1.4	$\{c\} = \{d\}$	h,1.3
	infer $c = d$	1.4
	infer $\mathit{is\text{-}start}(sr, c) \wedge \mathit{is\text{-}start}(sr, d) \;\Rightarrow\; c = d$	1

Lemma 2 states:

$$(s, r) \in \Sigma_b \wedge post\text{-}STEP_b((s, r), (s', r')) \;\Rightarrow\; (s', r') \in \Sigma_b$$

The proof again follows immediately from the definitions.

	from $(s, r) \in \Sigma_b \wedge post\text{-}STEP_b((s, r), (s', r'))$	
1	$inv\text{-}Srep(s)$	h, $inv\text{-}\Sigma_b$
2	$s \neq \{\,\}$	1, $pre\text{-}STEP_b$
3	$\exists b \in \mathbf{dom}\, s \cdot is\text{-}start(s, b)$	2, $inv\text{-}Srep$
4	$s' = \{start(s)\} ⩤ s$	1, 3, $post\text{-}STEP_b$
5	$start(s') = s(start(s))_2$	4, $start$
6	$\exists b \in \mathbf{dom}\, s' \cdot is\text{-}start(s', b)$	5, $is\text{-}start$
7	$\forall p, q \in \mathbf{dom}\, s \cdot p \neq q \;\Rightarrow\; s(p)_2 \neq s(q)_2$	h, $inv\text{-}\Sigma_b$
8	$\forall p, q \in \mathbf{dom}\, s' \cdot p \neq q \;\Rightarrow\; s'(p)_2 \neq s'(q)_2$	7, 4
9	$inv\text{-}Srep(s')$	6, 8
10	$inv\text{-}Srep(r)$	h, $inv\text{-}\Sigma_b$
11	$r' = r \cup \{start(s) \mapsto (s(start(s))_1, start(r))\}$	h, $post\text{-}STEP_b$
12	$start(r') = start(s)$	11, $start$
13	$\exists b \in \mathbf{dom}\, r' \cdot is\text{-}start(r', b)$	12, $is\text{-}start$
14	$\forall p, q \in \mathbf{dom}\, r \cdot p \neq q \;\Rightarrow\; r(p)_2 \neq r(q)_2$	10, $inv\text{-}\Sigma_b$
15	$sep(s, r)$	h, $inv\text{-}\Sigma_b$
16	$start(s) \notin \mathbf{dom}\, r$	15, sep
17	$\forall p, q \in \mathbf{dom}\, r' \cdot p \neq q \;\Rightarrow\; r'(p)_2 \neq r'(q)_2$	14, 16
18	$inv\text{-}Srep(r')$	13,17, $inv\text{-}Srep$
19	$sep(s', r')$	1, 4, 11, sep
	infer $(s', r') \in \Sigma_b$	9, 18, 19

Lemma 3 states:

$$\forall s \in Val^* \cdot \exists sr \in Srep \cdot gather(sr) = s$$

The proof is by a straightforward induction on sequences.

	from $s \in Val^*$	
1	**from** $s = [\,]$	
1.1	$\{\,\} \in Srep$	$inv\text{-}Srep$
1.2	$gather(\{\,\}) = [\,]$	$gather$
	infer $\exists sr \in Srep \cdot gather(sr) = s$	1.1, 1.2
2	**from** $s \in Val^*; \exists sr \in Srep \cdot gather(sr) = s$	IH
2.1	$p \notin \mathbf{dom}\, sr \wedge e \in Val$	assume
2.2	$sr' = \{p \mapsto (e, start(sr))\} \cup sr$	assume
2.3	$is\text{-}start(sr', p)$	2.2, $is\text{-}start$
2.4	$\forall q, q' \in \mathbf{dom}\, sr' \cdot q \neq q' \;\Rightarrow\; sr'(q)_2 \neq sr'(q')_2$	h2, 2.1, 2.2
2.5	$sr' \in Srep$	2.3, 2.4, $inv\text{-}Srep$
2.6	$gather(sr') = [e] \frown gather(sr)$	2.2, 2.3, 2.5, $gather$
	infer $\exists sr' \in Srep \cdot gather(sr') = [e] \frown s$	2.6, 2.7
	infer $\exists sr \in Srep \cdot gather(sr) = s$	indn(1, 2)

Lemma 4 states:

$$inv\text{-}\Sigma_b(\sigma_b) \wedge post\text{-}STEP_b(\sigma_b, \sigma_b') \;\Rightarrow\; post\text{-}STEP_a(retr\text{-}a(\sigma_b), retr\text{-}a(\sigma_b'))$$

The proof follows by expanding the definitions.

from	$inv\text{-}\Sigma_b(\sigma_b) \wedge post\text{-}STEP_b(\sigma_b, \sigma_b')$	
1	$(s, r) = \sigma_b; (s', r') = \sigma_b'$	assm
2	$s' = \{start(s)\} \lhd s$	1, $post\text{-}STEP_b$
3	$r' = r \cup \{start(s) \mapsto (s(start(s)_1, start(r)\}$	1, $post\text{-}STEP_b$
4	$gather(s') = gather(\{start(s)\} \lhd s)$	2, $gather$
5	$\quad = \mathbf{tl}\, gather(s)$	4, $gather$
6	$gather(r') = gather(r \cup \{start(s) \mapsto (s(start(s)_1, start(r))\}$	3, $gather$
7	$\quad = [\mathbf{hd}\, gather(s)] \curvearrowright gather(r)$	6, $gather$
infer	$post\text{-}STEP_a(retr\text{-}a(\sigma_b), retr\text{-}a(\sigma_b'))$	$post\text{-}STEP_a$, 5, 7

Lemma 5 states:

$$rel\text{-}b\text{-}c(\sigma_b, \sigma_c) \wedge post\text{-}STEP_c(\sigma_c, \sigma_c') \;\Rightarrow$$
$$\exists \sigma_b' \in \Sigma_b \cdot post\text{-}STEP_b(\sigma_b, \sigma_b') \wedge rel\text{-}b\text{-}c(\sigma_b', \sigma_c')$$

The proof requires the definitions of the functions/predicates; the existential introduction is by the one-point rule.

from	$rel\text{-}b\text{-}c(\sigma_b, \sigma_c) \wedge post\text{-}STEP_c(\sigma_c, \sigma_c')$	
1	$(s, r) = \sigma_b$	assm
2	$(hp, i, j) = \sigma_c$	assm
3	$(hp', i', j') = \sigma_c'$	assm
4	$hp' = hp \dagger \{i \mapsto (hp(i)_1, j)\}$	3, $post\text{-}STEP_c$
5	$i' = hp(i)_2$	3, $post\text{-}STEP_c$
6	$j' = i$	3, $post\text{-}STEP_c$
7	**from** $s' = \{start(s)\} \lhd s \wedge$ $r' = r \cup \{start(s) \mapsto (s(start(s)_1, start(r)\}$	assm
7.1	$hp' = r' \cup s'$	4, h7
7.2	$i' = start(s')$	5, h7
7.3	$j' = start(r')$	6, h7
7.4	$rel\text{-}b\text{-}c(\sigma_b', \sigma_c')$	7.1, 7.2, 7.3, h7, $rel\text{-}b\text{-}c$
	infer $post\text{-}STEP_b(\sigma_b, \sigma_b') \wedge rel\text{-}b\text{-}c(\sigma_b', \sigma_c')$	7.4, $post\text{-}STEP_b$
infer	$\exists \sigma_b' \in \Sigma_b \cdot post\text{-}STEP_b(\sigma_b, \sigma_b') \wedge rel\text{-}b\text{-}c(\sigma_b', \sigma_c')$	$\exists\text{-}I$, 7

Lemma 6 states:

$$permutes(merge(s1, s2), s1 \curvearrowright s2)$$

This proof uses nested induction on sequences. (In order to save (horizontal) space, *permutes* is abbreviated to p and *merge* to m in this proof.)

```
        from $s1 \in Val^* \land s2 \in Val^*$
1           from $s1 = [\,]$
1.1             $m(s1, s2) = s2$                                                          h1, $m$
1.2             $s1 \curvearrowright s2 = s2$                                             h1
            infer $p(m(s1, s2), s1 \curvearrowright s2)$                                  1.1, 1.2, $p$
2           from $p(m(s1, s2), s1 \curvearrowright s2)$                                   IH
2.1             from $s2' = [\,]$
2.1.1               $m([e] \curvearrowright s1, s2') = [e] \curvearrowright s1$           h2.1, $m$
2.1.2               $[e] \curvearrowright s1 \curvearrowright s2' = [e] \curvearrowright s1$   h2.1
                infer $p(m([e] \curvearrowright s1, s2'), [e] \curvearrowright s1 \curvearrowright s2')$   2.1.1, 2.1.2
2.2             from $p(m([e] \curvearrowright s1, s2'), [e] \curvearrowright s1 \curvearrowright s2')$    IH
2.2.1               $p(m(s1, [e'] \curvearrowright s2'), s1 \curvearrowright [e'] \curvearrowright s2')$   h2, $p$
2.2.2               from $e \leq e'$
2.2.2.1                 $m([e] \curvearrowright s1, [e'] \curvearrowright s2') = [e] \curvearrowright m(s1, [e'] \curvearrowright s2')$   h2.2.1, $m$
                    infer $p(m([e] \curvearrowright s1, [e'] \curvearrowright s2'), [e] \curvearrowright s1 \curvearrowright [e'] \curvearrowright s2')$   2.2.1, 2.2.2.1, $p$
2.2.3               from $e > e'$
2.2.3.1                 $m([e] \curvearrowright s1, [e'] \curvearrowright s2') = [e'] \curvearrowright m([e] \curvearrowright s1, s2')$   h2.2.2,$m$
                    infer $p(m([e] \curvearrowright s1, [e'] \curvearrowright s2'), [e] \curvearrowright s1 \curvearrowright [e'] \curvearrowright s2')$   2.2.1, 2.2.3.1, $p$
                infer $p(m([e] \curvearrowright s1, [e'] \curvearrowright s2'), [e] \curvearrowright s1 \curvearrowright [e'] \curvearrowright s2')$   $\lor$-$E$(2.2.2, 2.2.3)
            infer $p(m([e] \curvearrowright s1, s2), [e] \curvearrowright s1 \curvearrowright s2)$   indn(2.1, 2.2)
        infer $p(m(s1, s2), s1 \curvearrowright s2)$                                      indn(1, 2)
```

Lemma 7 states:

$$\mathit{ordered}(s1) \wedge \mathit{ordered}(s2) \;\Rightarrow\; \mathit{ordered}(\mathit{merge}(s1, s2))$$

The proof is identical in structure to that of Lemma 6

Lemma 8 states:

$$s' = \mathit{mergesort}(s) \;\Rightarrow\; \mathit{is\text{-}sort}(s, s')$$

This proof uses "course of values" induction so that the induction hypothesis can be applied to the arbitrary sub-parts of $s \in \mathit{Srep}$.

	from $s' = \mathit{mergesort}(s)$	
1	**from** **len** $s \leq 1$	
1.1	$\mathit{mergesort}(s) = s$	h1, *mergesort*
1.2	$\mathit{permutes}(s, s)$	*permutes*
1.3	$\mathit{ordered}(s)$	h, *ordered*
	infer $\mathit{is\text{-}sort}(s, s')$	*is-sort*, h, 1.1, 1.2, 1.3
2	**from** $s1' = \mathit{mergesort}(s1) \;\Rightarrow\; \mathit{is\text{-}sort}(s1, s1') \wedge$ $s2' = \mathit{mergesort}(s2) \;\Rightarrow\; \mathit{is\text{-}sort}(s2, s2')$	IH
2.1	$\mathit{ordered}(s1') \wedge \mathit{ordered}(s2')$	h2, *is-sort*
2.2	$\mathit{ordered}(\mathit{merge}(s1', s2'))$	Lemma 7, 2.1
2.3	$\mathit{permutes}(s1', s1) \wedge \mathit{permutes}(s2', s2)$	h2, *is-sort*
2.4	$\mathit{permutes}(\mathit{merge}(s1', s2'), s1 \curvearrowright s2)$	Lemma 6, *permutes*, 2.3
	infer $\mathit{is\text{-}sort}(s1 \curvearrowright s2, \mathit{merge}(s1', s2'))$	*is-sort*, 2.2, 2.4
	infer $\mathit{is\text{-}sort}(s, s')$	cov-indn(1, 2)

Lemma 9 states:

$$(l, r, a) \in \Sigma_m \wedge \mathit{post\text{-}MERGE}((l, r, a), (l', r', a')) \;\Rightarrow\; (l', r', a') \in \Sigma_m$$

The proof is obvious from the form of the proof of Lemma 2.

Lemma 10 states:

$$\begin{array}{l} \forall l, r, a, l', r', a' \in \mathit{Val}^* \cdot \\ \quad \mathit{post\text{-}MERGE}((l, r, a), (l', r', a')) \;\Rightarrow \\ \qquad\qquad\qquad \mathit{gather}(a') = \mathit{merge}(\mathit{gather}(l), \mathit{gather}(r)) \end{array}$$

The proof follows that of Lemma 4.

Lemma 11 states:

$$(l, r) = \mathit{split}(sr, p) \;\Rightarrow\; l, r \in \mathit{Srep} \wedge \mathit{sep}(l, r)$$

The proof is by induction on sr — remembering that the base case is for an *Srep* corresponding to two elements (see *pre-split*).

```
    from true
1         from $sr = \{p \mapsto (v, q), q \mapsto (v', \mathbf{nil})\} \wedge$
                    $q \in \mathbf{dom}\, sr \wedge q \neq start(sr)$                    base case
1.1            $split(sr, q) = (\{p \mapsto (v, \mathbf{nil})\}, \{q \mapsto (v', \mathbf{nil})\})$
1.2            $(l, r) = split(sr, q)$                                                   naming
          infer $l, r \in Srep \wedge sep(l, r)$                          1.1, 1.2, $Srep$, $sep$
2         from $(l, r) = split(sr, p) \wedge l, r \in Srep \wedge sep(l, r)$             IH
2.1            $sr' = \{o \mapsto (v', start(sr))\} \cup sr$                             naming
2.2            $(l', r') = split(sr', q)$                                                naming
2.3            from $q = start(sr)$
2.3.1               $(l', r') = (\{o \mapsto (v', \mathbf{nil})\}, sr)$             $split$, h2.3
               infer $l', r' \in Srep \wedge sep(l', r')$                     h2, $Srep$, $sep$
2.4            from $q \neq start(sr)$
2.4.1               $(l', r') = (\{o \mapsto (v', start(l))\} \cup l, r)$          $split$, h2.4
               infer $l', r' \in Srep \wedge sep(l', r')$                     h2, $Srep$, $sep$
          infer $l', r' \in Srep \wedge sep(l', r')$                           $\vee$-E(2.3, 2.4)
    infer $l, r \in Srep \wedge sep(l, r)$                                         indn(1, 2)
```

Lemma 12 states:

$$p \in \mathbf{dom}\, sr \wedge p \neq \mathit{start}(sr) \wedge (l, r) = \mathit{split}(sr, p) \;\Rightarrow\; \mathit{gather}(l) \frown \mathit{gather}(r) = \mathit{gather}(sr)$$

The proof follows the structure of that of Lemma 11.