Reasoning about Separation using Abstraction and Reification

Cliff B. Jones and Nisansala Yatapanage

School of Computing Science, Newcastle University, United Kingdom

Abstract. Showing that concurrent threads operate on separate portions of their shared state is a way of establishing non-interference. Furthermore, in many useful programs, ownership of parts of the state are exchanged dynamically. Reasoning about separation and ownership of heap-based variables is often conducted using some form of separation logic. This paper examines the *issue of separation* and investigates the use of abstraction to specify and to reason about separation in program design. Two case studies demonstrate that using *separation as an abstraction* is a potentially useful approach.

Keywords: concurrency, separation, ownership, abstraction

1 Introduction

Concurrent programs are difficult to reason about either formally or informally because of potential interference between threads; interference can be managed by separation of the parts of the state accessible to threads; separation arguments are often complicated by dynamic changes of ownership.

It is useful to distinguish the *issues* arising in the design of concurrent programs before fixing on specific notations — clearly, *separation/ownership* and *interference* constitute underlying issues. An obvious demarcation is to employ Separation Logic to tackle the first set of issues and something like Rely/Guarantee reasoning for the latter.

It has been shown elsewhere that 'pulling apart' the standard rely/guarantee notation throws light on the *issue of interference*. In [JHC15], the benefits of studying issues prior to choosing a notation are discussed. In particular, that paper takes a new look at specifying and reasoning about interference (the new presentation is more fully explained in [HJC14]).

In the same spirit, the current paper examines 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 provides this insight in [Hoa72]. Hoare shows that pre/post conditions of the code for separate threads can be conjoined providing the variables used by the threads are disjoint. He tackled normal (or 'scoped') variables where dynamic ownership might be controlled by something like monitors.

In comparison to scoped variables, it is more delicate to reason about separation over 'heap' variables whose addresses are computed by the programs in which they occur. Furthermore, exchange of *ownership* of heap addresses between threads is often disguised by intricate pointer manipulation.

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 only novelty with respect to standard data abstraction/reification is that the representation must be shown to preserve the separation property of the abstraction.

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 implementation uses (separate portions of) heap storage and the ownership of heap cells is exchanged between threads. It would be possible to object that the examples presented look like simple data reifications but that is, in fact, the main point. Using data abstraction, along with the one additional idea that separate abstract variables can be reified onto a shared data structure, throws light on the concepts of separation and ownership.

Of course, some notation has to be used for the specifications and requisite proof obligations but this is well-established and was not devised for concurrency. The authors happen to use ideas from VDM[1] but the same points could be made in Z or Event-B. In more complicated examples, it is useful to be explicit about 'framing' and VDM does offer ways of specifying read and write access to parts of the state. For framing, the ideas in [Bor00] or 'dynamic frames' [Kas11] would also be options.

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. However, as with the reported reformulation of rely/guarantee reasoning, focussing on the issue rather than a specific notation might give a new angle on notations for separation and/or reduce the need to develop new logics.

Hints for a top-down development of the list reversal algorithm are sketched in [JHC15]. 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 [JHC15], 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; the development of Concurrent Separation Logic [O'H07] came later. While Section 3 applies the idea of *separation as*

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

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 `i` initially points to the start of the list, `k` is a temporary place-holder and at termination of the algorithm, `j` points to the reversed list.

Reynolds used the separating conjunction of Separation Logic to develop a useful specification of the algorithm from the code. His specification demonstrates the ability of the separating conjunction operator to hide the details of the separation, such as showing that the two lists must remain separate and that they are separate from all other lists. While this is certainly a useful method for handling the complexities of separation, the following sections show how layered abstractions can offer a viable alternative.

```
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 (`*n` is the C-style pointer dereference of pointer `n`).

2.2 Abstract specification

The notion of reversing a sequence is expressed simply as a recursive function:

$$rev : Val^* \to Val^*$$

$$rev(list) \quad \triangleq \quad \textbf{if } list = [\,] \textbf{ then } list \textbf{ else } rev(\textbf{tl}\, list) \curvearrowright [\textbf{hd}\, list]$$

The initial step is to develop a program whose state is a pair of lists:

$$\Sigma_0 = (Val^* \times Val^*)$$

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_0 are implicitly separate — they are 'scoped' variables and, unless a language allows something like 'parameter passing by reference', there is no debate about a lack of separation.

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

$$post\text{-}REVERSE_0((s, r), (s', r')) \triangleq r' = rev(s)$$

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

```
$r \leftarrow [\,]$;
**while** $s \neq [\,]$ **do**
  $STEP_0$
**end while**

$pre\text{-}STEP_0((r, s)) \triangleq s \neq [\,]$
$post\text{-}STEP_0((r, s), (r', s')) \triangleq r' = [\mathbf{hd}\, s] \frown r \wedge s' = \mathbf{tl}\, s$
```

Fig. 2. Abstract list reversal program.

2.3 Representing sequences

The program in Figure 2 is based on abstract sequences and cannot address things like moving pointers to achieve in-place operation. To show how the list reversal can occur without moving the data, the abstract state needs to be represented as a heap:

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

(In VDM, maps ($D \xrightarrow{m} R$) are finite constructed functions; the fields of a pair $pr \in (Val \times [Ptr])$ are accessed here[2] by index, e.g. pr_1; the square brackets around Ptr indicate that it is optional and that $\mathbf{nil} \notin Ptr$ is a possible value.)

Such a heap might contain information for other threads and/or garbage discarded by processes. Section 2.4 completes the reification to just such a *Heap* but, here, an intermediate step is introduced which shows two scoped variables

[2] VDM aficionados would normally employ a 'record' construct here but using a pair and selecting by index reduces the potentially unfamiliar notation in this paper.

each containing a sub-heap that is precisely a sequence representation (*Srep*). (Although this intermediate representation could actually be elided, a significant advantage of its use is that *Srep* objects are also useful for the development of the concurrent program in Section 3.) One could define *Srep* using a datatype invariant but the proofs below benefit from defining the concept inductively as the least map $Srep \subseteq Heap$ containing:[3]

$$\begin{array}{l} \{\,\} \in Srep \\ sr \in Srep \wedge p \in Ptr \wedge p \notin \mathbf{dom}\, sr \;\;\Rightarrow\;\; (\{p \mapsto (v, start(sr))\} \cup sr) \in Srep \end{array}$$

Furthermore, a useful function that defines the start element can be defined over the recursive construction:

$$\begin{array}{l} start(\{\,\}) = \mathbf{nil} \\ start(\{p \mapsto (v, start(sr))\} \cup sr) = p \end{array}$$

The state for this intermediate development step contains two *Srep* objects which are required to have disjoint domains:[4]

$$\Sigma_1 = (Srep \times Srep)$$

where

$$inv\text{-}\Sigma_1((sr, rr)) \;\;\triangleq\;\; sep(sr, rr)$$

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

$$sep(sr, rr) \;\;\triangleq\;\; \mathbf{dom}\, sr \cap \mathbf{dom}\, rr = \{\,\}$$

On the Σ_1 representation, the specification of the operation corresponding to the body of the while loop in Figure 2 is:

$$\begin{array}{l} pre\text{-}STEP_1(sr, rr) \triangleq sr \neq \{\,\} \\ post\text{-}STEP_1((sr, rr), (sr', rr')) \triangleq \\ \quad \mathbf{let}\ p = start(sr)\ \mathbf{in} \\ \quad sr' = \{p\} \mathbin{⩤} sr \wedge rr' = rr \cup \{p \mapsto (sr(p)_1, start(rr))\} \end{array}$$

Lemma 1. *It is necessary to show that* $STEP_1$ *preserves the invariant of* Σ_1*.*

$$(sr, rr) \in \Sigma_1 \wedge pre\text{-}STEP_1((sr, rr)) \wedge post\text{-}STEP_1((sr, rr), (sr', rr')) \;\Rightarrow\; (sr', rr') \in \Sigma_1$$

The proof is by induction over *Srep*.[5]

[3] Of course, *Srep* and *start* are mutually recursive but it is clearer to separate their descriptions.

[4] So far, separation is a convenience that ensures transferring cells from one sequence to the other provides unused pointers; the restriction plays a bigger role in Section 2.4.

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

Proof obligations for data reification are standard in methods such as VDM (cf. [Jon90, Chap. 8]): retrieve functions are homomorphisms from the representation back to the abstraction.

$$retr_0 : \Sigma_1 \to \Sigma_0$$
$$retr_0((sr, rr)) \quad \triangleq \quad (gather(sr), gather(rr))$$

The *gather* function is again defined over the inductive construction of *Srep*:

$$gather\colon Srep \to Val^*$$
$$gather(\{\,\}) = [\,]$$
$$gather(\{p \mapsto (v, start(sr))\} \cup sr) = [v] \curvearrowright gather(sr)$$

VDM defines an 'adequacy' proof obligation which requires that, for each abstract state, there exists at least one representation state.

Lemma 2. *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 3. *$STEP_1$ models (under $retr_0$) the abstract $STEP_0$*

$$inv\text{-}\Sigma_1(\sigma_1) \wedge pre\text{-}STEP_0(retr_0(\sigma_1)) \wedge post\text{-}STEP_1(\sigma_1, \sigma_1') \;\Rightarrow\; post\text{-}STEP_0(retr_0(\sigma_1), retr_0(\sigma_1'))$$

The proof follows from unfolding the defined functions/predicates.

2.4 The heap

Although the two *Srep* variables in the preceding section are 'heap-like', each is used like a scoped variable. This section shows that the scoped variables can be represented in a single heap and that the behaviour on the heap remains as specified in Section 2.3.

This final representation uses a single heap (hp) and two pointers (i, j). The hp field of Σ_2 is essentially the heap underlying Figure 1.[6]

$$\Sigma_2 = (Heap \times Ptr \times Ptr)$$

where

$$inv\text{-}\Sigma_2((hp, i, j)) \quad \triangleq$$
$$\exists sr, rr \in Srep \cdot sr \cup rr \subseteq hp \;\wedge\; i = start(sr) \wedge j = start(rr)$$

[6] 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.

This is again an exercise in data reification. Here, it is mandatory that *sep* holds between the two sub-heaps because their union is used in $(sr \cup rr) \subseteq hp$; the fact that this is not an equality admits the possibility of other information in the heap. The retrieve function in this case is:

$$retr_1 : \Sigma_2 \to \Sigma_1$$

$$retr_1((hp, i, j)) \quad \triangleq \quad (trace(hp, i) \lhd hp, trace(hp, j) \lhd hp)$$

where:

$$trace : Heap \times Ptr \to Ptr\text{-}\mathbf{set}$$

$$trace(hp, p) \quad \triangleq \quad \begin{array}{l} \mathbf{if}\ p = \mathbf{nil} \\ \mathbf{then}\ \{\,\} \\ \mathbf{else}\ \{p\} \cup trace(hp, hp(p)_2) \end{array}$$

The definedness of *trace* for $Srep \subseteq Heap$ follows from $inv\text{-}\Sigma_2$.

Lemma 4. *The trace function applied to the start of an Srep returns exactly the pointers in that Srep; therefore, restricting the domain of a heap containing an Srep to such a trace yields the original Srep.*

$$sr \in Srep \wedge sr \subseteq hp \;\Rightarrow\; trace(hp, start(sr)) \lhd hp = sr$$

The proof is by induction over *Srep*.

The adequacy proof obligation for Σ_2 is:

Lemma 5. *There is at least one representation in Σ_2 for each Σ_1 state:*

$$\forall (sr, rr) \in \Sigma_1 \cdot \exists (hp, i, j) \in \Sigma_2 \cdot retr_1((hp, i, j)) = (sr, rr)$$

The proof creates a minimal *hp* that contains exactly the union of *sr*/*rr* which are disjoint.

On Σ_2, the specification of the operation corresponding to $STEP_1$ above is:

$$pre\text{-}STEP_2((hp, i, j)) \triangleq i \neq \mathbf{nil}$$

$$post\text{-}STEP_2((hp, i, j), (hp', i', j')) \triangleq$$
$$i' = hp(i)_2 \wedge j' = i \wedge hp' = hp \dagger \{i \mapsto (hp(i)_1, j)\}$$

for which the reification proof obligation is:

Theorem 1. *$STEP_2$ models (under $retr_1$) the abstract $STEP_1$*

$$inv\text{-}\Sigma_2(\sigma_2) \wedge pre\text{-}STEP_1(retr_1(\sigma_2)) \wedge post\text{-}STEP_2(\sigma_2, \sigma_2') \;\Rightarrow$$
$$post\text{-}STEP_1(retr_1(\sigma_2), retr_1(\sigma_2'))$$

The proof again follows from unfolding the defined functions/predicates.

Code (in C++) that satisfies $post\text{-}STEP_2$ is given in Figure 3. The final step in the correctness argument is to note that the loop in Figure 2 terminates when $s = [\,]$ and the loop on the representation terminates when $i = \mathbf{nil}$; under $retr_1/retr_0$, these conditions are equivalent.

```
Class Pair{
    Val v;
    Pair* p;
}
Pair* reverse(Pair* i){
  Pair* k;
  Pair* j = NULL;
  while (i != NULL) {
       // STEP
    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_0) of Section 2.2, the two variables are assumed to be distinct; standard data reification rules apply where that distinction is obvious; in the step to Σ_2, it must be established that the abstraction of separation holds in the representation as (changing) portions of a shared heap.

A valuable by-product 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: invariants (and the use of predicate restricted types) effectively provide pre conditions for 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.

This example is simple and, in fact, the development presented here is even clearer than that in an earlier draft. The point is that the important notion of separation has been tackled without any special notation. Section 3 employs the same approach on a program that uses parallelism.

3 Mergesort

The preceding 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 step, the argument list is divided into two parts (preferably, but not necessarily, of roughly equal sizes) which are recursively submitted to *mergesort*; 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:

$$\mathit{is\text{-}sort} : \mathit{Val}^* \times \mathit{Val}^* \to \mathbb{B}$$
$$\mathit{is\text{-}sort}(s, s') \quad \triangleq \quad \mathit{ordered}(s') \wedge \mathit{permutes}(s', s)$$

The *ordered* predicate tests that its argument is an ascending sequence.

$$\mathit{ordered} : \mathit{Val}^* \to \mathbb{B}$$
$$\mathit{ordered}(s) \quad \triangleq \quad \forall i \in \{1..\mathbf{len}\, s - 1\} \cdot s(i) \le s(i+1)$$

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

$$\mathit{permutes} : \mathit{Val}^* \times \mathit{Val}^* \to \mathbb{B}$$
$$\mathit{permutes}(s, s') \quad \triangleq \quad \mathit{bag\text{-}of}(s') = \mathit{bag\text{-}of}(s)$$

$$\mathit{bag\text{-}of} : \mathit{Val}^* \to (\mathit{Val} \xrightarrow{m} \mathbb{N}_1)$$
$$\mathit{bag\text{-}of}(s) \quad \triangleq \quad \{e \mapsto \mathbf{card}\, \{i \in \mathbf{inds}\, s \mid s(i) = e\} \mid e \in \mathbf{elems}\, s\}$$

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 selects the minimum head element from its two argument lists and recurses:

```
$merge : Val^* \times Val^* \to Val^*$
$merge(s1, s2) \quad \triangleq$
    if $s1 = [\,] \vee s2 = [\,]$
    then $s1 \curvearrowright s2$
    else if $(\mathbf{hd}\, s1 \le \mathbf{hd}\, s2)$
        then $[\mathbf{hd}\, s1] \curvearrowright merge(\mathbf{tl}\, s1, s2)$
        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:*

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

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:

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

```
$mergesort(s) \quad \triangleq$
    **if len** $s \leq 1$
    **then** $s$
    **else let** $s1, s2$ **be st** $s1 \frown s2 = s \wedge s1 \neq [\,] \wedge s2 \neq [\,]$ **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:*

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

Because of the arbitrary split, the proof uses course-of-values induction on s.

3.3 Representing sequences

Having dealt with the algorithmic ideas in Section 3.2, the method used in Section 2.3 can be followed by reifying the abstract sequences into *Srep* objects as defined in Section 2.3.

The implementation consists of two operations: $MSORT_1$ operates on S_1:

$S_1 = (Srep \times Srep)$,

while the $MERGE_1$ operation uses a state that contains three instances of *Srep*:

$M_1 = (Srep \times Srep \times Srep)$,

where the three fields are pairwise separate (*sep* cf. Section 2.3). As in Section 2.3, this notion of separation is used here only to simplify the exchange of ownership of cells between l, r and a. In Section 3.4, separation justifies the embedding of three *Srep* objects in a single heap.

Turning to the presentation of the (abstract) program, standard sequential program constructs (e.g. the while loop) were used in Section 2.2. This approach is not followed here because it would be a digression to derive a proof rule for the (non-tail) recursion needed in $MSORT_1$ (this construct is not covered in [Jon90]). Instead the recursion in both $MERGE_1$ and $MSORT_1$ is represented as predicates by 'quoting post conditions' (cf. [Jon90, Section 9.3]).

$$
\begin{array}{l}
\textit{post-MERGE}_1((l, r, a), (l', r', a')) \triangleq \\
\quad l = \{\,\} \wedge a' = r \wedge l' = r' = \{\,\} \vee \\
\quad r = \{\,\} \wedge a' = l \wedge l' = r' = \{\,\} \vee \\
\quad l \neq \{\,\} \wedge r \neq \{\,\} \wedge l(\textit{start}(l))_1 \leq r(\textit{start}(r))_1 \wedge \\
\qquad \textit{post-MERGE}_1((\{\textit{start}(l)\} \lhd l, r, a), (l', r', ma)) \wedge \\
\qquad a' = \{\textit{start}(l) \mapsto (l(\textit{start}(l))_1, \textit{start}((ma))\} \cup ma \vee \\
\quad l \neq \{\,\} \wedge r \neq \{\,\} \wedge l(\textit{start}(l))_1 > r(\textit{start}(r))_1 \wedge \\
\qquad \textit{post-MERGE}_1((l, \{\textit{start}(r)\} \lhd r, a), (l', r', ma)) \wedge \\
\qquad a' = \{\textit{start}(r) \mapsto (r(\textit{start}(r))_1, \textit{start}((ma))\} \cup ma
\end{array}
$$

Lemma 9. *$MERGE_1$ preserves separation:*

$$(l, r, a) \in M_1 \wedge \textit{post-MERGE}_1((l, r, a), (l', r', a')) \;\Rightarrow\; (l', r', a') \in M_1$$

The proof of this lemma is obvious from the form of the proof of Lemma 1.

Lemma 10. *The operation $MERGE_1$ mirrors the function merge*

$$
\begin{array}{l}
\forall l, r, a, l', r', a' \in \textit{Val}^* \cdot \\
\quad \textit{post-MERGE}_1((l, r, a), (l', r', a')) \;\Rightarrow \\
\qquad\qquad\qquad \textit{gather}(a') = \textit{merge}(\textit{gather}(l), \textit{gather}(r))
\end{array}
$$

Here again, the proof follows that of Lemma 3.

It is necessary to split an *Srep* into two separate values of that type. The function *split* recurses until the argument p is located in the representation:

$$\textit{split} : \textit{Srep} \times \textit{Ptr} \to (\textit{Srep} \times \textit{Srep})$$

```
split(sr, p)  △
    if $p = start(sr)$
    then $(\{\,\}, sr)$
    else let $(l, r) = split(\{start(sr)\} \lhd sr, p)$ in
        $(\{start(sr) \mapsto (sr(start(sr))_1, start(l))\} \cup l, r)$
pre $p \in \mathbf{dom}\, sr$
```

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

$$
\begin{array}{l}
sr \in \textit{Srep} \wedge p \in \mathbf{dom}\, sr \wedge (l, r) = \textit{split}(sr, p) \;\Rightarrow \\
\qquad\qquad\qquad l \in \textit{Srep} \wedge r \in \textit{Srep} \wedge \textit{sep}(l, r)
\end{array}
$$

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:*

$$
\begin{array}{l}
sr \in \textit{Srep} \wedge p \in \mathbf{dom}\, sr \wedge (l, r) = \textit{split}(sr, p) \;\Rightarrow \\
\qquad\qquad\qquad \textit{gather}(l) \curvearrowright \textit{gather}(r) = \textit{gather}(sr)
\end{array}
$$

This proof follows the structure of that of Lemma 11.

Whereas MERGE_1 is used sequentially (there are no concurrent threads), instances of MSORT_1 are to 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.

MSORT_1

ext wr $sr: \mathit{Srep}$

post $(sr = \{\,\} \vee sr(start(sr))_2 = \mathbf{nil}) \wedge sr' = sr \vee$
$\quad \exists p \in \mathbf{dom}\, sr, l, r \in \mathit{Ptr} \cdot$
$\quad\quad p \neq start(sr) \wedge$
$\quad\quad (l, r) = split(sr, p) \wedge$
$\quad\quad post\text{-}\mathit{MSORT}_1(l, l') \wedge post\text{-}\mathit{MSORT}_1(r, r') \wedge$
$\quad\quad post\text{-}\mathit{MERGE}_1((l', r', \{\,\}), (\{\,\}, \{\,\}, sr'))$

Theorem 2. *The final conclusion is that the operation MSORT_1 mirrors the function mergesort:*

$$post\text{-}\mathit{MSORT}_1(sr, sr') \;\Rightarrow\; gather(sr') = mergesort(gather(sr))$$

which follows from the lemmas.

3.4 The heap

It is almost as straightforward as in Section 2.4 to develop code for MSORT_2 and MERGE_2. There is one interesting addition required because of the concurrent execution of two instances of MSORT_2. The invariants follow the same pattern as with the sequence reversal example — for MERGE_2, the representation in the *Heap* is:

$M_2 = (\mathit{Heap} \times \mathit{Ptr} \times \mathit{Ptr} \times \mathit{Ptr})$

where

$inv\text{-}M_2((hp, x, y, z)) \;\triangleq$
$\quad \exists l, r, a \in \mathit{Srep} \cdot$
$\quad\quad l \cup r \cup a \subseteq hp \wedge x = start(l) \wedge y = start(r) \wedge z = start(a)$

and the corresponding representation for MSORT_2 is simply:

$S_2 = (\mathit{Heap} \times \mathit{Ptr})$

where

$inv\text{-}S_2((hp, p)) \;\triangleq\; \exists sr \in \mathit{Srep} \cdot sr \subseteq hp \wedge p = start(sr)$

The respective retrieve functions are:

$$retr\text{-}m_1 : M_2 \to M_1$$

$$retr\text{-}m_1((hp, x, y, z)) \quad \triangleq \\ \quad (trace(hp, x) \lhd hp, trace(hp, y) \lhd hp, trace(hp, z) \lhd hp)$$

$$retr\text{-}s_1 : S_2 \to S_1$$

$$retr\text{-}s_1((hp, p))) \quad \triangleq \quad (trace(hp, p) \lhd hp)$$

It is, however, necessary to establish non-interference between the concurrent threads. This can be done with a simple use of rely/guarantee reasoning:[7]

$rely\text{-}MSORT_2$: $p' = p \wedge trace(hp, p) \lhd hp' = trace(hp, p) \lhd hp$
$guar\text{-}MSORT_2$: $trace(hp, p) \ntriangleleft hp' = trace(hp, p) \ntriangleleft hp$

The code in Figures 4 and 5 satisfies the specifications of $MERGE_2$ and $MSORT_2$ respectively; a specific implementation of *split* is also provided.

```
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.

[7] A suitable formal proof rule is given in Section 4.

```
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.

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.

4 Discussion

The research reported in this paper is one vector of the ‘Taming Concurrency’ project in which it is hoped to identify and/or to develop apposite notations for reasoning about the underlying *issues* that make designing and justifying 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’. Of course, using existing notation is not precluded but ensuring that the issues are clear looks to be a prudent starting point.

The Rely/Guarantee (R/G) approach (of which more below) was devised for reasoning about the issue of *interference*. The R/G concept has been substantially recast in [HJC14] and the new version is summarised in [JHC15]. 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 shows their 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/ownership 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 representations (e.g. of separate scoped 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 the examples in this paper: 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 in 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].

Separation is, of course, a way of ruling out interference so it is interesting to understand those situations where a user can choose which approach to adopt. With scoped variables, there is a variety of ways to define the named variables (frame) of different threads. VDM allows state components to be marked as having **rd/wr** access; the keyword notation is rather heavy but serves the purpose and many alternatives could be considered. In the refinement calculus presentation of [HJC14,JHC15], write access is made clear but not access for reading. Section 3.4 above indicates the recording of read/write access to subsets of heap addresses. (There are, of course, occasions where read:write clashes require assumptions in the reading process and rely conditions are an obvious candidate for recording such assumptions.) One approach that is used with separation logics to handle such access constraints is to employ 'fractional permissions' [Boy03].

Technical connections between R/G and SL are considered in [VP07,Vaf07]). It might also 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 emphasises that complete separation is an extreme case of minimising interference.

One last comment on the similarities is that the importance of (data) abstraction in the proposed way of looking at separation nicely mirrors its key role in R/G methods [Jon07].

More narrowly, on the content of this paper, alternatives considered by the authors include:

– 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 examples, it would be possible to omit the intermediate representation and to move directly from the respective abstract states to the general *Heap*. As mentioned in Section 2.3, the fact that *Srep* is used in both examples is one argument for its separation — the other argument is the divorce of the algorithm design from the messy heap representation details.

For future work, it would be useful to develop a 'theory' of *Srep* objects. Another interesting avenue 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. Finally, 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 concurrent implementations of tree and graph representations.

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 programs 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.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In Roland Carl Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000, Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

[Boy03] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[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.

[Hoa72] C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[JHC15] Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27:475–497, 2015.

[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. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

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

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 375(1–3):109–119, 2007.

[JY15] Cliff B. Jones and Nisansala Yatapanage. Reasoning about separation using abstraction and reification. In R. Calinescu and B. Rumpe, editors, *SEFM*, volume 9276 of *LNCS*, pages 1–17. Springer, 2015.

[Kas11] Ioannis T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23(3):267–288, 2011.

[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 *LNCS*, pages 169–182. Springer, 2010.

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

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007.

[VP07] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. Vasconcelos, editors, *CONCUR 2007*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

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